JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 8, 313-324 (1990)

Hierarchical Gate-Array Routing on a Hypercube Multiprocessor *

O. A. OLUKOTUN AND T. N. MUDGE

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109

Gate-arrays are the most common design style for semicustom VLSI integrated circuits. An important part of the gate-array design process is the routing of wires between the logic elements, which is an extremely compute-intensive operation. This paper presents an algorithm for routing gate-arrays that uses a hypercube connected parallel processor to provide the necessary computation power. In order to make optimal use of the hypercube, the routing algorithm is organized so that interprocessor communication is kept to a minimum. It occurs only during the global routing and crossing placement phases of the algorithm, which constitute less than 15% of the total running time of the algorithm. On the basis of the results of executing the algorithm on two gate-array benchmarks the case is made for using hypercube multiprocessors as accelerators for compute-intensive CAD operations. © 1990 Academic Press, Inc.

1. INTRODUCTION

The use of gate-arrays in the design of semicustom VLSI integrated circuits is common in the semiconductor industry today. At present, gate-arrays outsell standard cell designs, the other major semicustom design methodology, by a margin of over 4 to 1 [13]; moreover, they are available in a variety of technologies that span the design space of speed, power, and integration density.

Gate-arrays are designed using a predefined library of well-characterized cells which have been supplied by a vendor. After the cells and their interconnections have been specified the gate-array design is ready for automatic layout. Automatic layout maps a structural description of a circuit into a physical description consisting of geometric coordinates for all the circuit elements and interconnection wiring [30]. Automatic layout consists of placement, or positioning the circuit cells on the layout surface, and routing, or interconnecting the cells. A good placement is essential for high-quality routing. In this paper we will assume that placement has been performed and focus on the routing phase of gate-array design.

As the density and size of gate-arrays increase, routing becomes less tractable and consumes larger amounts of computer time. Parallel computers, particularly those with large numbers of processors, have the potential to provide the necessary computational power with which to accelerate gate-array layout and thus facilitate the design of larger chips than it is possible to do at present. However, in order to realize this potential, it is necessary to develop efficient parallel algorithms for these computers. In this paper we describe an algorithm for a particular type of parallel machine, the hypercube multiprocessor.

Parallel processors can be separated into two major types on the basis of their memory architecture: shared-memory machines or distributed-memory machines. A variety of terms have been coined for these two types of multiprocessors, including "tightly coupled" for shared-memory and "loosely coupled," "message-based," or "disjoint-memory" for distributed-memory. In shared-memory multiprocessors, the processors can access a common memory which they use for interprocessor communications, as well as the usual function of storing the program and data. In contrast, distributed-memory multiprocessors normally comprise a set of processors, each with its own private memory, that communicate among themselves over a fixed interconnection network. Recently, a wide selection of commercial parallel processors of both types has become available. The shared-memory type has been the most widely accepted because it represents a natural evolution from the conventional uniprocessor. However, the congestion that can occur at the shared-memory limits the number of processors to a few dozen [37]. By comparison, the distributed-memory machines appear to offer a greater potential for increased performance because they can readily scale to thousands of processors [8]. Indeed, in [18] supercomputer performance levels were reported for Monte Carlo simulations of particle transport using the 64-processor commercial hypercube NCUBE/ten, and in widely publicized experiments a 1024-processor version of the NCUBE/ten was used by researchers at Sandia National Laboratories to obtain a fixed-size speedup of 500-fold on problems in beam stress, baffled wave simulation, and unstable fluid flow [7].

Although distributed-memory machines allow greater numbers of processors to be usefully connected, they pre-

* This work was supported in part by the NSF under Grant MIP-8802771.

0743-7315/90 $3.00
Copyright © 1990 by Academic Press, Inc.
All rights of reproduction in any form reserved.

sent a greater challenge to the programmer and algorithm designer. Given the current immature state of parallel programming and parallel algorithm design, it is only cost effective to employ massively parallel distributed-memory machines when the considerable effort required to develop an efficient parallel algorithm can be amortized over a long period of time through repeated use. Many applications associated with physical CAD, in particular routing, fall into this category. It follows that massively parallel distributed-memory machines are candidates for hardware accelerators for compute-intensive physical CAD applications. This notion was the point of departure for the work described here.

Gate-array routing is usually implemented in two steps: global routing followed by detailed routing. The purpose of global routing is to determine which of the routing channels that lie between the cells of the gate-array will be used for each connection. A number of global routers [31, 17, 23, 35] which are variations of the Lee–Moore or the Dijkstra algorithm [16, 20, 6] have been proposed. In detailed routing wires are assigned to their final tracks according to a set of design rules. Detailed routing is usually performed with some type of channel router [38] or Lee–Moore router [16, 20] and the routing region is constrained by the channels and feed-throughs specified for the net in the global routing step. Two aspects of gate-array routing make it amenable to parallelization on a distributed-memory multiprocessor. First, global routing partitions the routing region into independent subregions which can be distributed among the processors for detailed routing. Second, much more time is spent in the detailed routing step than in the global routing step. Since the detailed routing step requires no interprocessor communication, the overall parallel routing algorithm can be made very efficient.

In this paper, we present a hierarchical routing algorithm designed for implementation on a massively parallel distributed-memory multiprocessor. This two-step algorithm is intended for routing multiterminal nets of a gate-array that exhibits near-uniform wiring density. It is based on our earlier work described in [22] but uses the pattern routing ideas presented in [3] for the global step. Global routing, since it involves global information spread over the processors, can produce excessive amounts of interprocessor communication. The use of a modified version of the first algorithm in [3] keeps this interprocessor communication to a minimum. Multiterminal nets are connected as a Steiner tree using a heuristic algorithm implemented during the global step. Detailed routing is performed by a modified Lee–Moore router. In our experiment two layers of interconnect are used for routing with vias serving as connections between the layers. This is not a limitation and, in principle, the algorithm could be extended to routing problems that have any number of layers. The parallel routing algorithm maps directly onto a hypercube topology and, as we will show, achieves good speedup.

To date, Lee–Moore routing has dominated the approaches taken for application of special-purpose parallel hardware to accelerate routing [34, 2, 32]. These single instruction multiple data (SIMD) array processor architectures are designed specifically to accelerate grid-search based algorithms. One of the first uses of a multiple instruction multiple data (MIMD) computer for routing was the Wire Routing Machine developed by Hong *et al.* [14]. This special-purpose architecture has 64 microprocessors configured as an 8×8 mesh on which a two-phase algorithm (global routing followed by detailed routing) is implemented. The global routing phase is based on Lee–Moore routing and uses an exponential channel density metric to avoid overflows. The speedup due to parallelism is reported as modest. This is due in part to the low power of the processor nodes and to the fact that the global routing phase does not make efficient use of the processors because only one net is globally routed at a time. Other approaches intended for general-purpose parallel computers make use of algorithms (e.g., simulated annealing and channel routing) that are best suited to a few tightly connected processors, because they require large amounts of interprocessor communication to keep a global data structure up to date in order to achieve good routing quality [28, 4, 39, 26]. In contrast, our approach, which is intended for many loosely coupled processors, reduces expensive communication traffic by keeping the global information local to each processor throughout the hierarchical decomposition of the global routing step. While this approach results in a low efficiency for the global routing step, the overall algorithm achieves high efficiency and produces good-quality routing. We have not seen any other routing algorithms that both have high efficiency and produce good-quality routing running on large distributed-memory multiprocessors.

The paper is organized as follows. In the next section our gate-array routing algorithm is presented in two parts: the global routing phase and the detailed routing phase. In Section 3 we describe how the router can be implemented on a hypercube multiprocessor. Section 4 contains experimental results obtained by using the gate-array router to route two gate-array benchmarks with a 64-processor version of the NCUBE/ten. Section 5 contains some concluding remarks.

2. HIERARCHICAL ROUTING

2.1. Global Routing

We assume a uniform gate-cell array in which each gate-cell is surrounded by four boundaries that form a rectangle (see Fig. 1). Each boundary has a channel capacity, which is the number of detailed wiring tracks that can cross it. These detailed wiring tracks may be feed-throughs that run between gate-cells in a row or routing channels adjacent to the boundary. The terminals of each gate are located on the top

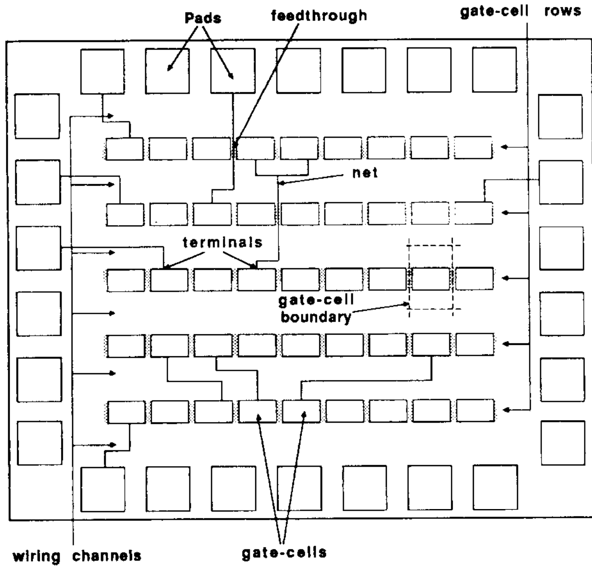

FIG. 1. Gate-cell array.

and bottom boundaries of a gate-cell. A *net* consists of a list of gate-cell terminals to be interconnected. There are two interconnection layers for routing, one for vertical and another for horizontal wire segments. Vias are introduced to make connections between the layers and *jogs* (wrong way wires) are allowed. The global routing of a single net is a set of gate-cells that contains all terminals of the net such that there is a path through adjacent gates-cells between all pairs of net terminals. At each gate-cell boundary that the path crosses, a wiring track is consumed, and the number of gate-cell boundaries the path crosses is taken as the global wire length. A feasible global routing for all nets is a global routing for each net that does not exceed the channel capacity at any boundary.

Hierarchical global routing begins by dividing the routing region, which consists of an $m \times n$ array of gate-cells, into four "macro"-gate-cells (see Fig. 2). Each macrocell contains a quarter of the $m \times n$ gate-cells. If a global routing between these macrocells is performed, then the quadrants can be separated into independent routing problems. After further subdividing each of the new routing problems into four subregions another global routing creates 16 independent routing problems. This process of quadrisection followed by global routing continues until there are as many subregions as there are processors (see Fig. 2). The detailed routing is performed by each processor on its subregion in parallel. This hierarchical approach reduces a problem of routing in an $m \times n$ array of cells to the problem of routing in a 2×2 grid at each level of the hierarchy. Our solution to this problem uses a modified form of the first approach described in [3] and is explained in the next section. Unlike [3] we do not restrict a wire route from crossing a vertical or horizontal boundary twice; i.e., all the routing possibilities that can occur in a 2×2 grid are allowed. This modification introduces an extra step called boundary crossing placement (explained in Section 2.1.1) into the global routing phase but allows a clean parallel decomposition of the routing problem into four separate routing problems at each step in the hierarchy. In the following section we explain further 2×2 routing among macrocells, and in Section 2.1.2 we give the details of extending the 2×2 solution to the $m \times n$ case using the hierarchical approach sketched above.

2.1.1. *Routing in a 2 × 2 Grid*

In the case of routing in a 2×2 grid it is possible to enumerate all the *types* of nets that may occur. The net types, T_i, for two, three, and four terminal nets are shown in Fig. 3. Each net is assigned a type on the basis of the position of its terminals within the 2×2 grid. There are a small number of ways in which each net type may be routed within the grid. These routing possibilities for each net type are also shown in Fig. 3, to the right of their corresponding types. We use $P_{i,j}$ to denote possibility j of net type T_i. Thus the original problem of routing a 2×2 grid is now a problem of partitioning the nets of each type among the type's possibilities, or determining the values of $x_{i,j}$, where $x_{i,j}$ denote the number of nets routed using $P_{i,j}$. The values of variables $x_{i,j}$ can be found by solving the following integer programming problem.

To formulate the problem we start with a set of linear restrictions. If t_i is the number of nets of type T_i, then a valid solution to the routing problem should route all nets of each type. This can be expressed as

$$\sum_j x_{i,j} = t_i \quad \text{for} \quad 1 \leqslant i \leqslant 11. \tag{1}$$

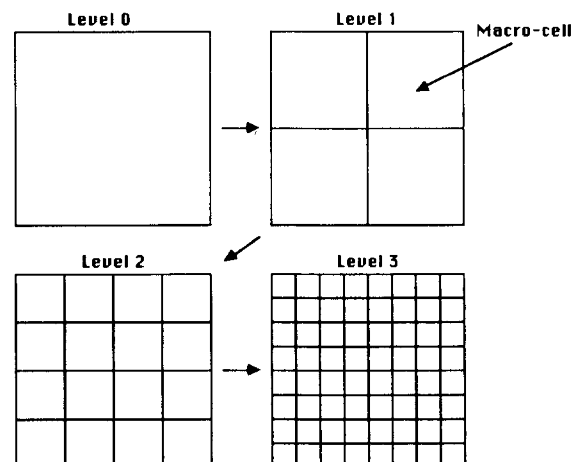

FIG. 2. Macrocell hierarchy.

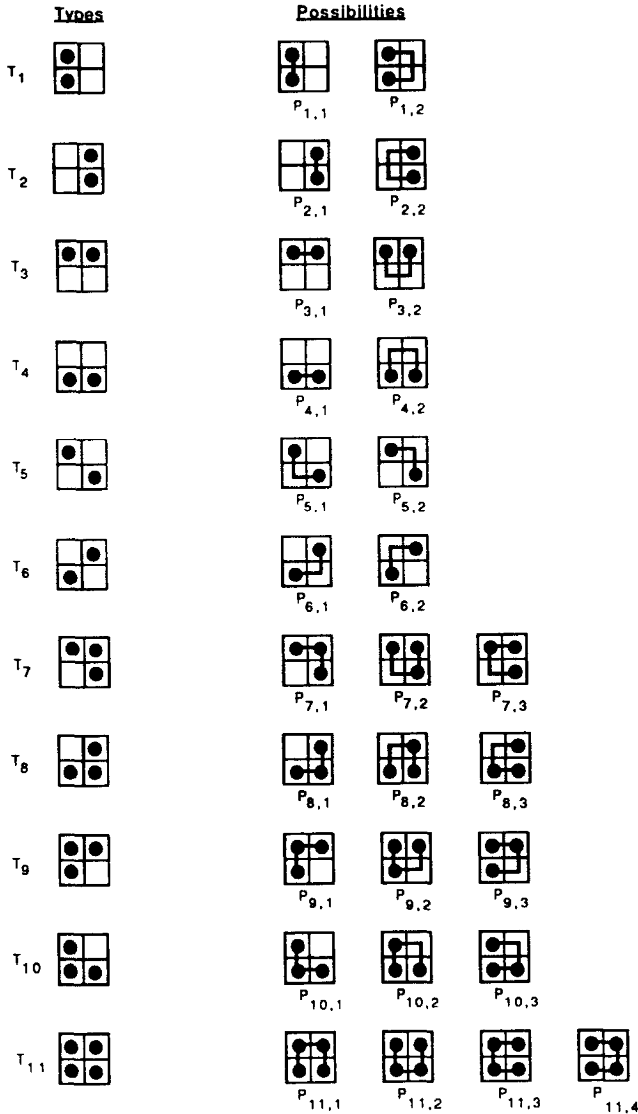

FIG. 3. Routing types and possibilities of a 2 × 2 grid from [3].

The channel capacity limitations imply another set of restrictions. Referring to Fig. 4, let

$$\mathcal{V}_1 \triangleq \{P_{i,j} \mid P_{i,j} \text{ crosses the upper vertical boundary } v_1\}.$$

$\mathcal{V}_1$ is a set of possibilities that can be enumerated by examining Fig. 3. Then to satisfy capacity limitations we have the restriction

$$\sum_{(i,j)\ni P_{i,j}\in\mathcal{V}_1} x_{i,j} \leq C_{v_1}, \tag{2}$$

where C_{v_1} is the crossing capacity of the vertical boundary v_1, and the domain of the summation is all pairs of indices (i, j) such that possibility $P_{i,j}$ is an element of the set $\mathcal{V}_1$. Let $\mathcal{V}_2$, $\mathcal{H}_1$, and $\mathcal{H}_2$ be sets, defined in a way similar to that in which $\mathcal{V}_1$ is defined, for the other three crossing boundaries; then we also have the following restrictions due to capacity limitations,

$$\sum_{(i,j)\ni P_{i,j}\in\mathcal{V}_2} x_{i,j} \leq C_{v_2} \tag{3}$$

$$\sum_{(i,j)\ni P_{i,j}\in\mathcal{H}_1} x_{i,j} \leq C_{h_1} \tag{4}$$

$$\sum_{(i,j)\ni P_{i,j}\in\mathcal{H}_2} x_{i,j} \leq C_{h_2}, \tag{5}$$

where C_{v_2}, C_{h_1}, and C_{h_2} are the crossing capacities of the remaining boundaries. The objective is to minimize total wire length W, which is equivalent to minimizing the number of boundary crossings, i.e.,

$$\text{minimize } W = \sum_{(i,j)\ni P_{i,j}\in\mathcal{V}_1} x_{i,j} + \sum_{(i,j)\ni P_{i,j}\in\mathcal{V}_2} x_{i,j} + \sum_{(i,j)\ni P_{i,j}\in\mathcal{H}_1} x_{i,j} + \sum_{(i,j)\ni P_{i,j}\in\mathcal{H}_2} x_{i,j}. \tag{6}$$

The size of this integer programming problem is 28 variables ($x_{i,j}$) and 15 constraint equations, (1)–(5). A satisfactory approach for solving this integer programming problem is to pose it as a linear programming problem, use the Simplex method [25] to find a solution, and then round the results. This approach is possible because the $x_{i,j}$ are typically large integers. They are large integers because there are many terminals in each quadrant [11].

The solution of this fixed-size integer programming problem yields values of $x_{i,j}$. However, to complete the global routing problem, we must decide (1) which nets of each net type will be routed in each net type possibility and (2) exactly where on each boundary each net will cross. These issues are addressed next.

Assigning possibilities to net types. Once the above integer programming problem has been solved, there may be more than one nonzero $x_{i,j}$ for each net type T_i. Since each of the nets of T_i can be routed in only one of the type's routing possibilities, we must assign each net to a particular possibility $P_{i,j}$. To guide this assignment, we make a heuristic attempt to minimize overall net wire length and reduce

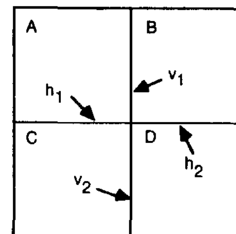

FIG. 4. Quadrants and boundary crossings of a 2 × 2 grid.

congestion at the crossing boundary. For example, if we have t_1 nets of type T_1, and $x_{1,1}$ and $x_{1,2}$ are both nonzero, we must assign $x_{1,1}$ of the nets to $P_{1,1}$ and $x_{1,2}$ to $P_{1,1}$; i.e., $x_{1,1}$ of the nets must be routed straight across the h_1 boundary, and $x_{1,2}$ nets must be detoured via the h_2 boundary (see Fig. 3). To determine the assignment, we sort the nets by the sum of the distance of each net's terminals from the vertical boundary. The first $x_{1,1}$ nets which have terminals that are farthest from the vertical boundary are routed in fashion $P_{1,1}$, and the rest of the nets ($x_{1,2}$) are routed in fashion $P_{1,2}$. Ties between nets that arise using this heuristic are broken by considering distance from the horizontal boundary. If we consider a net type with more than two routing possibilities, such as T_7, the situation becomes more complicated. Here, we may have to partition nets among possibilities $P_{7,1}$, $P_{7,2}$, and $P_{7,3}$ (see Fig. 3). Using the same heuristics, we sort the nets by the sum of the distance of the nets' terminals from the v_1 and h_2 boundaries. The $x_{7,1}$ nets with the greatest total distance are assigned route $P_{7,1}$. This heuristic ensures that those nets which span the greatest distance are routed in the most direct way so as to minimize overall wire length. To determine the assignments to $P_{7,2}$ and $P_{7,3}$, we note that both possibilities have quadrant C in common. If the nets are sorted by their terminal's distance from the h_2 boundary the $x_{7,2}$ nets with least distance would be routed in fashion $P_{7,2}$, and the rest of the $x_{7,3}$ nets would be assigned route $P_{7,3}$. Here, we attempt to minimize both the length of wire running parallel to the crossing boundary and the total wire length.

Boundary crossing placement. Finally, in order to fully decompose the routing at each level of hierarchy into four independent routing problems at the next level down in the hierarchy we must decide where to route each net that crosses a boundary. Our goal is to place the crossing points of a net so that the resulting route is a rectilinear Steiner tree interconnecting the net's terminals [1]. In general this is an NP-complete problem and we must again rely on heuristics to obtain an approximate solution. At each level of the global hierarchy the solution of the 2 × 2 routing problem creates four subproblems. These subproblems are intended for solution on different processors; therefore we must fix the crossings and introduce new terminals into the routing region so that at any level lower down in the hierarchy no interprocessor communication will be necessary for a processor to obtain terminal position information. In essence, we require that a crossing be placed to an accuracy that is within the smallest subregion created at the lowest level of the global hierarchy. Our heuristic uses the positions of terminals to place crossings at gate-cell resolution.

The bounding line segment of a net is formed by connecting the projections of all terminal points onto a quadrant boundary. Examples of bounding line segments for a five-terminal net are shown in Fig. 5. If a quadrant contains a single terminal the bounding line segment will be the point projection of this terminal. For each net for which a crossing must be placed there will be at least one bounding line segment and at most two bounding line segments. If there are two line segments we take their intersection and place the crossing point at a location on the boundary opposite the terminal that is closest to the boundary. If there is no intersection of the line segments the crossing is placed opposite the terminal which is nearest to both the boundary and the other line segment. If one line segment is a single point, the crossing is placed opposite this point. If both line segments are points, the crossing is placed opposite the point closest to the boundary. Figure 6 shows three scenarios and the crossing placements that would result from using these heuristics.

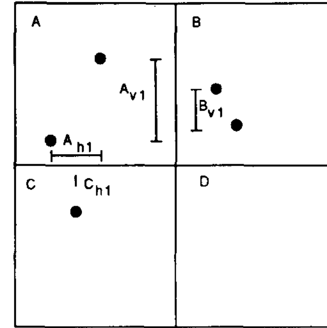

FIG. 5. Bounding line segments of a net. Each bounding line is labeled by quadrant name and boundary crossing name.

2.1.2. *Routing in an n × m Grid*

We now describe a hierarchical router for the $n \times m$ global routing problem based on 2 × 2 solutions. We begin at each level of the hierarchy by dividing the routing area into quadrants defined in terms of gate-cells. If possible, the horizontal and vertical divisions are made so that there are an equal number of gate-cells in each quadrant. Next we perform net classification. Net classification determines the values t_i for each net type T_i from the net list. Nets are classified by checking the location of each terminal of the net with the quadrant boundaries. Nets for which all terminals lie within a single quadrant are not considered for global routing. The values of t_i are used to set up the 2 × 2 grid routing integer programming problem, as discussed in Section 2.1.1. After the solution to the integer programming problem has been found, we assign possibilities to net types and then perform boundary crossing placement. The boundary crossing placement introduces new terminal cells into what are now four independent routing subproblems. These four subproblems can now be solved, in the same manner, simultaneously in parallel. The hierarchy stops when there are as many independent routing problems as there are processors. As we will see in the next section this

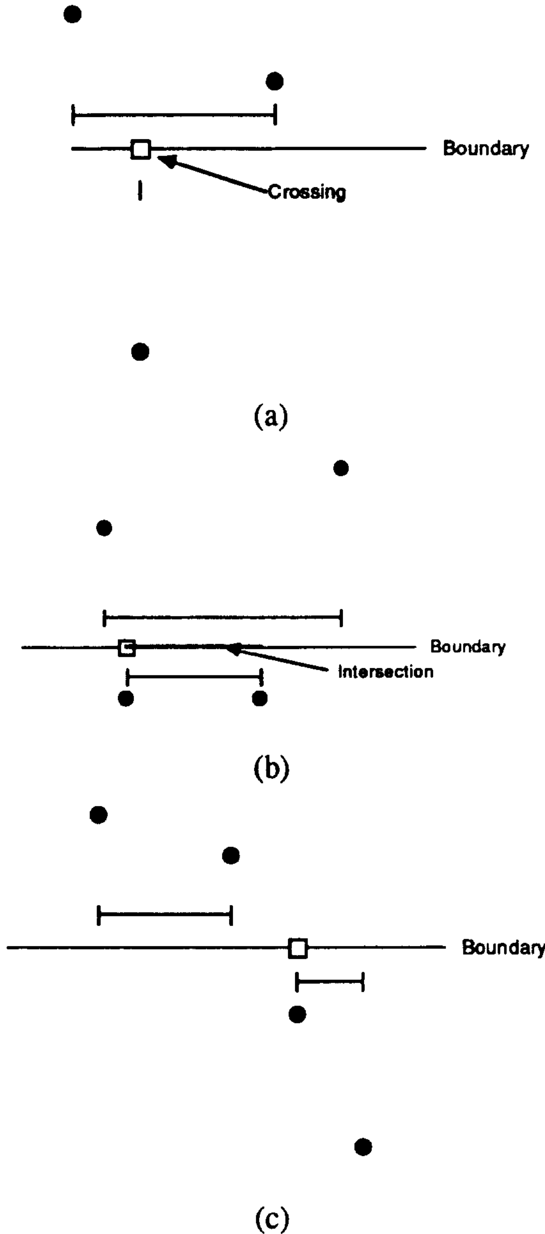

FIG. 6. Bounding lines, intersections, and crossing placements. (a) The crossing point is placed at the intersection point. (b) The crossing is placed at a point on the intersection closest to a terminal. (c) Because there is no intersection, the crossing is placed at a point opposite the closest terminal that is also closest to an end-point on the opposite bounding line segment.

final level of routing problems is handled using a high-quality Lee–Moore router.

We note that in our formulation of global routing we do not reuse boundaries at any time during global routing and so do not need the restrictive 2×2 routing of [3] that would eliminate the use of possibilities $P_{1,2}$, $P_{2,2}$, $P_{7,3}$, $P_{8,3}$, $P_{9,3}$, $P_{10,3}$, $P_{11,3}$, and $P_{11,4}$ as well as reduce the number of routing subproblems created at each level to two. Figure 7 shows an example of hierarchical global routing on a 4×4 array of macro-gate-cells. Each terminal cell may represent several actual terminals; however, we may assume we have only 16 processors and so we are only interested in following the hierarchy to a level at which 16 independent routing problems have been created. Figure 7a shows the terminal cells of the net as they would appear at the lowest level of the global hierarchy. In Fig. 7b we solve the initial 2×2 problem and place the crossings, which are represented by boxes, on the quadrant boundary using the techniques of Section 2.1.1. The introduction of new terminal cells at the next level down in the hierarchy produces the four 2×2 subproblems shown in Fig. 7c. Once these four subproblems have been solved the global routing is complete, as shown in Fig. 7d.

2.2. Detailed Routing

Once global routing is complete each processor contains an independent routing problem. However, before detailed routing can commence each processor must agree with its four neighboring processors on where, in terms of the detailed grid, nets that cross their common boundaries will be placed. The method of detailed crossing placement is a modified version of a technique described in [22] and involves communication among neighboring processors. The goals of detailed crossing placement are to evenly distribute the crossings along the boundary in order to reduce congestion, and to place the crossings so that jogs in the wiring paths will not be necessary. Each processor is responsible

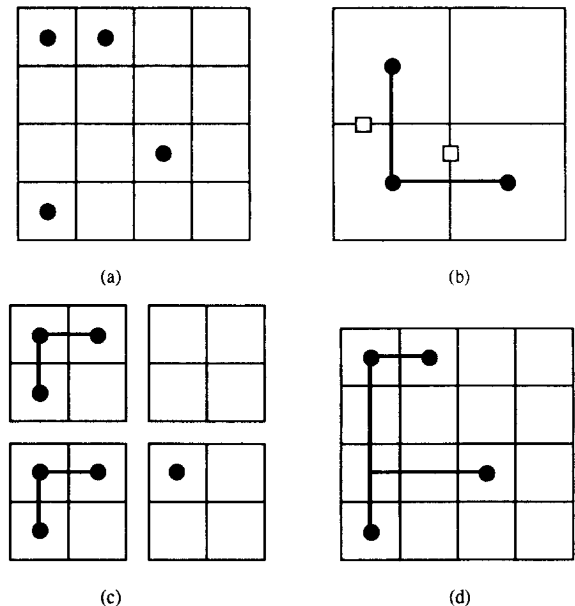

FIG. 7. 4×4 hierarchical global routing. (a) Terminal gate-cells. (b) A solution to the initial 2×2 problem with crossing placements. (c) Solutions to the four 2×2 subproblems. (d) Final global routing.

for the crossings on the southern and eastern borders of the routing region that it works on.

The crossing placement technique uses an iterative refinement method, in which each processor calculates the preferred position of a crossing, on either its southern or eastern border, on the basis of a weighted average of the current position of the crossing, if it has been established, and the positions of the other crossings and terminals of the net as projected onto the crossing boundary. The closer a crossing or terminal is to the one being placed the more weight it is given. Once a preferred crossing position for all nets on both boundaries has been calculated the crossings are evenly distributed along the crossing boundary. Figure 8 shows an example of detailed crossing placement. In this example crossing C, which is to be placed, will be connected to terminals X and Y in the detailed routing step. X' is the projection of X on the crossing border of C, and Y' is the projection of Y. A new position for crossing C is computed as a weighted average of the positions of X', Y', and C. Position X' is given more weight than Y' because X is closer to the boundary. Each iteration of the crossing placement algorithm consists of two steps. First, place all the eastern crossings and second, place all the southern crossings. The first step starts from the processors on the western border of the mesh and proceeds toward the east, while the second step begins with processors on the northern border and proceeds toward the south. At each step after crossings have been determined, a message is sent to the appropriate neighbor processor with the updated crossing positions. The number of iterations the crossing placement algorithm executes is predetermined by the user. In practice, convergence occurs quickly, and the relative weights given to terminals opposite the crossings being placed tend to cause the wiring paths to form straight lines as one would like (see ahead to Fig. 12). More complex heuristics for detailed crossing placement are employed if one or both of the terminals are also crossings on another boundary.

The technique used for detailed routing incrementally routes the nets in the routing region. In this discussion a *subnet* refers to some subset of the terminals of a net and a *connected* subnet is a subset of net terminals that have been connected by a marked-path within the detailed grid. Detailed routing begins by repeatedly starting a two-layer variable cost Lee–Moore router from each terminal or crossing of each net in the routing region. For each net the routing grid is assumed to be empty, so that any routes that are found are independent of all other net connections in the routing region. The Lee–Moore router is implemented using some ideas from [10, 12]. The cost metrics used to guide the search include the use of a preferred layer for horizontal and vertical wires. Expansions on a layer that are in the nonpreferred direction are penalized with a cost higher than that of expansion in the preferred direction. Expansions that change layers to produce vias are also penalized with a higher cost. Only the minimum cost connected subnet of each net is placed on a list of subnets ordered by cost. Once all nets have an entry in the list, the minimum cost connected subnet is selected and is kept by permanently marking its path in the routing grid. This process of finding the minimum cost subnet as a starting point for the detailed routing process minimizes a problem associated with Lee–Moore routers, namely, that the quality of routing is highly dependent on the order in which nets are routed, and, in particular, on which net is chosen for routing first.

After the initial subnet is picked, the remaining subnets are examined in order and kept if their path does not conflict with the path of a previously kept subnet. If the path does conflict, the Lee–Moore router is invoked again and a new connected subnet is sought that avoids the conflict. This new connected subnet is placed on the ordered list of subnets in the appropriate place. If no connected subnet is found its associated net is marked as "unroutable." For every connected subnet that is kept the Lee–Moore router is invoked on the net containing it to find a new connected subnet to add to the ordered list. Minimum cost subnets are taken from the list and processed until the list is empty, in which case, all nets have been completely routed, or no more connected subnets can be found. Separate detailed routing problems are executed in parallel on all processors and require no interprocessor communication.

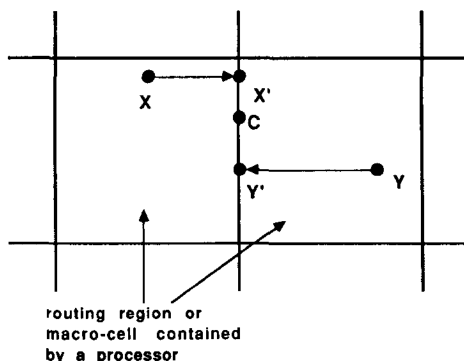

FIG. 8. An example of detailed crossing placement.

3. IMPLEMENTING HIERARCHICAL ROUTING

The hierarchical global routing algorithm described in the previous section can be mapped in an obvious way onto a pyramid computer. A pyramid computer of size P is a distributed-memory parallel machine having $P^{1/2} \times P^{1/2}$ mesh connected processors at its base, and $\log_4 P$ levels of mesh connected processors above [19]. Each processor at level k is connected to four neighbors at level k, four children at level $k - 1$, and a parent at level $k + 1$ (see Fig. 9). Execution of the global routing algorithm would begin in the processor at the apex of the pyramid. This processor

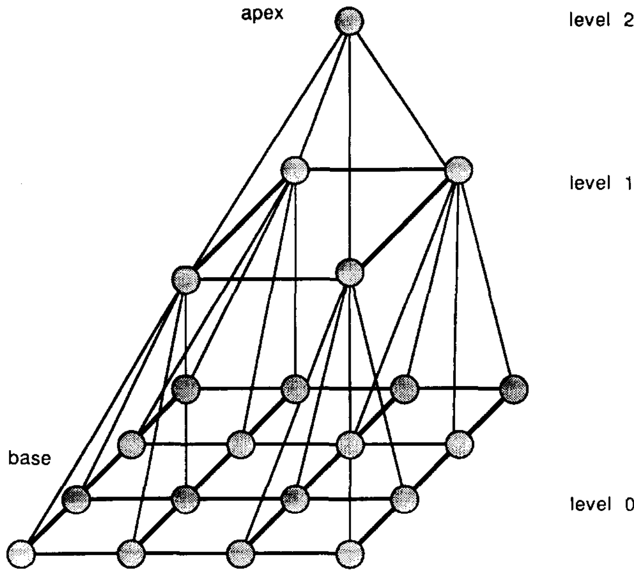

FIG. 9. A pyramid of size 16.

would solve the 2×2 routing problem over the entire routing area and then pass each of its four children at level $(\log_4 P) - 1$ the data for one of the independent routing problems. Each child then solves its own 2×2 routing problem and passes on four smaller routing problems to its children. This process continues until the base of the pyramid is reached. We note, in the execution of this algorithm on a pyramid computer, that all processors work independently and in parallel, except when data are being passed from one level to the next. However, only processors from one level are in use at any point in time; therefore, although a pyramid computer may execute the global routing algorithm in a natural way, it does not do it very efficiently. In particular, the detailed routing which would be performed in the base mesh of the pyramid would not use

$$\sum_{k=0}^{(\log_4 P)-1} 4^k \simeq \frac{P}{3}$$

processors, which is more than 25% of the total available $(\simeq 4P/3)$. Furthermore, the detailed phase takes much longer than the global phase, and it is trivially parallelizable; therefore, it is important to use all available processors for this phase. Nevertheless, the pyramid paradigm is useful for understanding how hierarchical routing maps onto hypercube computers.

3.1. Hypercube Computers

Hypercube computers have 2^d processors connected in the topology of a d-dimensional binary hypercube. Figure 10 shows the familiar example of a four-dimensional hypercube. Each processor of a d-dimensional hypercube is labeled with a unique d-bit address, such that each bit position corresponds to a coordinate along one of d-dimensions. Because the pyramid algorithm discussed above is the most natural one for our hierarchical global routing algorithm, we would like to map it onto a hypercube computer so that neighboring processors in the pyramid algorithm are also neighbors in the hypercube, i.e., so that the mapping has a dilation of one. Unfortunately, pyramids cannot be embedded in hypercubes with a dilation of one [33]. However, hypercube computers can still efficiently execute the hierarchical routing algorithm using a less than optimal mapping, because, among other things, the mesh used for the detailed routing phase, the dominant phase of the algorithm, can be embedded with a unit dilation.

The steps of the pyramid algorithm make use of two-dimensional meshes that trace out the levels of a pyramid. Two-dimensional meshes may be embedded in hypercubes so that mesh neighbors are also hypercube neighbors. In such an embedding each processor is uniquely labeled by a d-bit binary address denoted xy (x and y concatenated), where x and y are two $d/2$ bit gray codes, one for each of the two dimensions of the mesh. An example of this mapping is shown in Fig. 11 for a hypercube of dimension $d = 6$ and a 8×8 mesh [21]. The gray code of a mesh cell corresponds to its hypercube address. If we take this two-dimensional mesh to be the base of a pyramid of size $P = 2^d$, we can also map the rest of the meshes used in the pyramid as follows. Take the processor whose address is all zeros to be the apex of the pyramid. To find the processor addresses of the apex's four children (c_0, c_1, c_2, c_3) we use the expressions

$$c_0 = xy$$
$$c_1 = (x \oplus 2^l)y$$
$$c_2 = x(y \oplus 2^l)$$
$$c_3 = (x \oplus 2^l)(y \oplus 2^l),$$

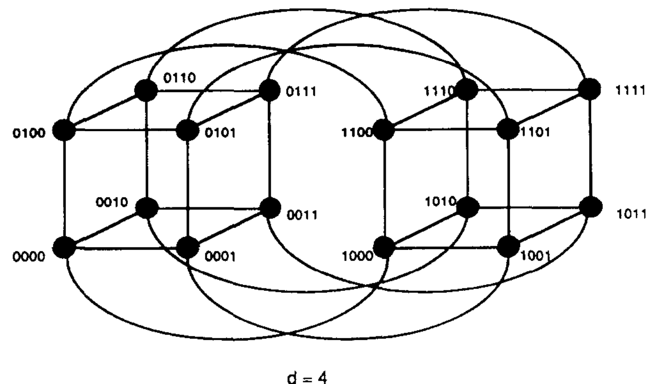

FIG. 10. A hypercube of dimension $d = 4$.

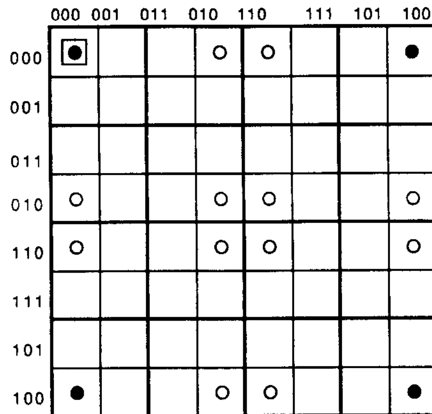

FIG. 11. The mapping of an 8×8 mesh onto a hypercube combined with the mapping of the pyramid algorithm onto a hypercube. The apex processor at level 3 is labeled by a filled circle enclosed within a square; its four children at level 2, including itself, are labeled with a filled circle. The processors at level 1 are labeled with a filled or unfilled circle. The complete mesh is the base level 0 of the pyramid.

where l is the pyramid level. For the apex processor $l = d/2 - 1$ and $xy = 00\cdots0$. Each child processor calculates the address of its children in the same way using an l value which is reduced by one. This process continues until the base mesh is reached and $l = 0$. In this mapping every parent processor has one child which is a nonneighbor processor. Each nonneighbor parent–child pair is at most two interconnection links away. This mapping ensures that at each level, adjacent processors in the pyramid mesh at that level are also adjacent in the hypercube. Furthermore, it allows hypercube processors to be reused at each level of the pyramid; therefore the mapping scheme is also suitable for the global routing algorithm, because only the processors on one pyramid level are used at a time and it allows the mapping of a pyramid of size equal to the number of processors in the hypercube.

4. EXPERIMENTAL RESULTS

The NCUBE/ten is an example of a general-purpose parallel computer with a hypercube interconnection topology which can accommodate up to 1024 nodes in a 10-dimensional hypercube. In our experiments we used an early version of the NCUBE/ten in which each node is a custom 32-bit microprocessor with 128 kbytes of memory and the ability to perform floating point arithmetic [9]. The nodes are capable of a peak performance of 2 MIPS and 0.5 MFLOPS. The connection between the nodes is by dedicated point-to-point bit-serial DMA channels. The hypercube is managed by a host processor (an Intel 80286). Our experiments were performed using a 64-node version of the NCUBE/ten.

To measure the performance of our parallel routing algorithm we executed two gate-array benchmarks. The first gate-array benchmark involves routing a 735-gate circuit in a 900-gate gate-array organized as 25×18 circuit blocks [27]. As a measure of performance we use fixed-size speedup [5]. This is the ratio of the time it takes to solve a problem in serial on a single processor to the time it takes to solve the same problem in parallel using P processors. The term fixed-size refers to the fact that the size of the problem does not scale with the number of processors. The scaled speedup is normally a much higher figure. A more accurate measure of the benefits obtained from parallel processing would be to compare the running time of the parallel version with the time taken by the best serial algorithm on one node. However, because many of the subproblems needed to perform a routing are NP-complete [15] and heuristics are used to solve them, there is no good basis on which to compare different routers and thus to identify the "best" serial algorithm. Therefore, we will use the serial time of our algorithm to determine speedup, while recognizing this limitation in the interpretation of our results.

Due to memory constraints on the NCUBE processor it was possible to route this benchmark only on a hypercube with 64 nodes—smaller cubes lacked the aggregate memory. To estimate the fixed-size speedup of our algorithm we ran a serial version of our parallel algorithm on an Apollo DN 570 computer which had sufficient memory. We then normalized the resulting execution time by multiplying it by the ratio of the execution time for the Dhrystone on a single NCUBE node to that on the DN 570 [36]. Table I shows the comparison in terms of execution time for the gate-array benchmark. In comparing Dhrystone benchmark results, it was found that the DN 570 has an execution rate which is 6.9 times that of an NCUBE processor. From this figure we derive an overall fixed-size speedup of 35.2 for the parallel routing algorithm over the serial version; see Table I. Our algorithm was able to fully route 97% of the nets using 10 tracks per vertical channel and 12 tracks per horizontal channel. This percentage of nets routed is higher than that of the approach taken in [27], where only 94% of

TABLE I

Comparison of Gate-Array Benchmark Routing Times

Computer	Routing time (s)			
	Global routing	Crossing placement	Detailed routing	Overall
(1) DN 570	15.2	24.7	1,899.1	1,939.0
(2) DN 570 (normalized)	104.9	170.4	13,103.8	13,379.1
(3) NCUBE (64 processors)	18.0	35.7	327.5	380.2
Speedup (3) vs (2)	5.8	4.8	40.0	35.2

the nets were routed. For the next benchmark, as we shall see, we were able to route all the nets.

For the parallel global routing the speedup S of a pyramid computer of size $P = 4^l$ is given by

$$S = \frac{T_s}{T_p} = \frac{1}{l} \sum_{i=0}^{l-1} 4^i$$

if we assume the time taken at each level of the pyramid is approximately constant. Thus for a pyramid of size 64, such as the one we are simulating with the hypercube, the maximum speedup for parallel global routing is 7. This speedup neglects the time it takes for a processor at each level to send data to its four children at the level below it. We have achieved a speedup of 5.8 (Table I), with the inclusion of communication time, which gives an efficiency exceeding 80% of the available parallelism for the global routing step executed on a hypercube. Detailed crossing placement exhibits a speedup of 4.8 using all 64 processors. This low value is due to the limited inherent parallelism and the communication-intensive nature of the crossing placement step. Detailed routing is completely parallel; however, even this step does not achieve a perfect speedup of 64 due to the uneven load distribution among processors. Uneven load distribution is inevitable, because gate-arrays rarely partition evenly and net connection density is not uniform across all gate-cells.

The second benchmark we used to evaluate the performance of our parallel router is the Primary1 circuit from the benchmark suite of the International Workshop on Placement and Routing 1988 [24]. This circuit is intended for either a standard cell or a gate-array layout style. It consists of 904 nets and 752 cells. The cells are organized as 26, 6000-μm rows with 13-track horizontal routing channels between rows. The internal utilization of the array is 86%. The vertical routing channels were composed of the built-in feed-troughs and the gaps between cells. The placement was obtained by simulated annealing using the TimberWolf3.2 standard-cell placement and global routing package [29]. Interestingly, the global routing produced by TimberWolf for this benchmark exceeded the number of tracks available (13) for some channels in the gate-array. In contrast, using the placement information from TimberWolf, we were able to completely route the gate-array using no more than 13 tracks per channel.

TABLE II

A Comparison of Primary1 Gate-Array Benchmark Routing Times on 16 Processors

Computer	Routing time (s) Global routing	Detailed routing	Overall
(1) DN 4000	5.0	4,892.0	4,897.0
(2) DN 4000 (normalized)	25.5	24,949.2	24,974.7
(3) NCUBE (16 processors)	5.0	2,116.0	2,121.0
Speedup (3) vs (2)	5.1	11.8	11.8

TABLE III

A Comparison of Primary1 Gate-Array Benchmark Routing Times on 64 Processors

Computer	Routing time (s) Global routing	Detailed routing	Overall
(1) DN 4000	12.0	1897.0	1909.0
(2) DN 4000 (normalized)	61.2	9674.7	9735.9
(3) NCUBE (64 processors)	7.0	219.0	226.0
Speedup (3) vs (2)	8.7	44.2	43.1

This second experiment was performed on an upgraded NCUBE/ten with 512 kbytes of memory per node. The algorithm was optimized for speed instead of space to take advantage of the increase in memory. We also eliminated the detailed crossing placement step by fixing the crossings on the detailed grid during global crossing placement. Tables II and III show the results for 16 and 64 processors, respectively. Furthermore, we were still not able to fit the whole routing problem in one node of the NCUBE, so a DN 4000 was used to estimate speedup.

It is not possible to compute an optimum routing for circuits as complex as the ones we have used, so absolute figures of routing quality cannot be obtained. Instead, we compare our results with those of other routers. Unfortunately, we were not able to obtain the results of other routers for this gate-array benchmark because all participants at the International Workshop on Placement and Routing 1988 chose to lay out the benchmark as a standard cell. However, we were able to compare the relative quality of routing between the 16- and 64-processor cases and the estimated wire length found by the TimberWolf global router. We have done significantly better than the TimberWolf estimations (Table IV). As one would expect the quality of the 64-pro-

TABLE IV

Quality of Primary1 Gate-Array Benchmark Routing

Measure	16 Processors	64 Processors	Δ%	Timberwolf (estim.)
Horizontal wire length (μm)	608,860	625,710	2.7	647,469
Vertical wire length (μm)	712,970	717,080	0.5	883,753
Total wire length (μm)	1,321,830	1,342,790	1.5	1,531,222
Number of vias	4,521	5,028	11.2	NA

cessor routing is degraded compared to the 16-processor case because global information is lost. However, the amount of degradation in routing quality is relatively small (Table IV), but the reduction in routing time is almost a factor of 10 (Tables II and III).

The completed Primary1 circuit without the pad cells is shown in Fig. 12. From this figure it is clear that the density of wires is not uniform. The nonuniform distribution density of wires means that some processors have more work to do than others. The uneven distribution of work may be severe; in the 64-processor case the ratio between the finishing times of the last and first processor is 5. The resulting efficiency of the detailed routing step is 70% instead of close to 100% as one would predict for a parallel computation that does not involve communication (Table III). However, correcting this load imbalance would involve extra computation to estimate wire density or extra communication to distribute the work. Exploring the trade-offs between the execution time reduction that can be achieved by good load balancing and the extra computation and communication time necessary to achieve it is a subject of further research.

5. CONCLUSIONS

We have presented a parallel hierarchical routing algorithm for routing gate-arrays and have mapped it onto a hypercube multiprocessor to route two modestly sized gate-arrays. The results show that the hypercube computer can be used to obtain high-quality routing and to achieve a reasonable speedup. In fact, this fairly simple machine—one large board of components—was over eight times faster, in absolute terms, than a high-performance workstation composed of many more parts (Table III). If other compute-intensive CAD tasks could be ported to hypercubes and achieve similar levels of performance, then an argument could be made for using them as CAD compute servers. However, the modest benchmarks (less than 1 kgate) stretched the memory capacity of our version of the NCUBE/ten to its limit. Therefore, it is clear, that for these computers to be effective on large gate-arrays (20 kgates and larger) much more memory per processor node is necessary. We close by noting that recent developments in commercial hypercube machines—nodes with many megabytes of memory, and processors that are much faster than our machine—have more than satisfied this requirement, and it should now be possible to use hypercubes to significantly reduce the computation time of critical CAD operations such as routing even for the largest circuits.

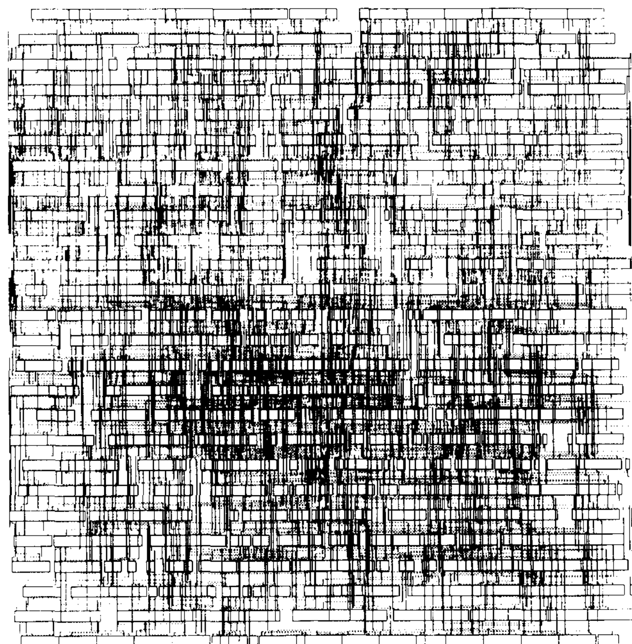

FIG. 12. Core of the Primary 1 gate-array benchmark.

REFERENCES

1. Aho, A., Garey, M. R., and Hwang, F. K. Rectilinear Steiner trees: Efficient special case algorithms. *Networks* 7 (1977), 37–58.
2. Breuer, M. A., and Shamsa, K. A hardware router. *J. Digital Syst.* **IV**, 4 (1981), 392–408.
3. Burstein, M., and Pelavin, R. Hierarchical wire routing. *IEEE Trans. Computer-Aided Design IC Syst.* **CAD-2**, 4 (Oct. 1983), 223–234.
4. Chung, M. J., and Rao, K. K. Parallel simulated annealing for partitioning and routing. *Proc. IEEE International Conference on Computer Design*, 1986, pp. 238–242.
5. Denning, P. J. Speeding up parallel processing. *Amer. Sci.* **76** (Jul.–Aug. 1988), 347–349.
6. Dijkstra, E. A note on two problems in connection with graphs. *Numer. Math.* **1** (1959), 269–271.
7. Gustafson, J. L., Montry, G. R., and Benner, R. E. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Statist. Comput.* **9**, 4 (July 1988), 1–32.
8. Hayes, J. P., Mudge, T. N., Stout, Q. F., Colley, S., and Palmer, J. Architecture of a hypercube supercomputer. *Proc. International Conference on Parallel Processing*, Aug. 1986, pp. 653–660.
9. Hayes, J. P., Mudge, T. N., Stout, Q. F., Colley, S., and Palmer, J. A microprocessor-based hypercube supercomputer. *IEEE MICRO* (Oct. 1986), 6–17.
10. Hightower, D. The Lee router revisited. *Proc. ACM/IEEE Conference on Computer-Aided Design*, 1983, pp. 136–139.
11. Hiller, F. S., and Lieberman, G. J. *Operations Research*. Holden–Day, San Francisco, 1974.
12. Hoel, J. H. Some variations of Lee's algorithm. *IEEE Trans. Comput.* **C-25**, 1 (Jan. 1976), 19–24.
13. Hollis, E. E. *Design of VLSI Gate Array ICs*. Prentice–Hall, Englewood Cliffs, NJ, 1987.
14. Hong, S. J., and Nair, R. Wire routing machines—New tools for VLSI physical design. *Proc. IEEE.* **71**, 1 (Jan. 1983), 57–65.
15. Johnson, D. S. The NP-completeness column: An ongoing guide. *J. Algorithms* **3** (1983), 381–395.
16. Lee, C. Y. An algorithm for path connections and its applications. *IRE Trans. Electron. Comput.* **EC-10** (Sept. 1961), 346–358.

17. Marek-Sadowska, M. Global router for gate-array. *Proc. IEEE International Conference on Computer Design,* 1984, pp. 332–337.
18. Martin, W., Wan, T.-C., Poland, D., Mudge, T., and Abdel-Rahman, T. Monte Carlo photon transport on the NCUBE. In Heath, M. (Ed.). *Proc. 1986 Conference on Hypercube Multiprocessors.* Society for Industrial and Applied Mathematics, 1987, pp. 454–463.
19. Miller, R., and Stout, Q. F. Data movement techniques for the pyramid computer. *SIAM J. Comput.* **16,** 1 (Feb. 1987), 38–60.
20. Moore, E. Shortest path through a maze. *Ann. Comput. Lab. Harvard Univ.* **30** (1959), 282–292.
21. Mudge, T. N., and Abdel-Rahman, T. S. Vision algorithms for hypercube machines. *J. Parallel Distrib. Comput.* **4** (1987), 79–94.
22. Olukotun, O. A., and Mudge, T. N. A preliminary investigation into parallel routing on a hypercube computer. *Proc. 24th ACM/IEEE Design Automation Conference,* 1987, pp. 814–820.
23. Preas, B., and VanCleemput, W. Routing algorithms for hierarchical IC layout. *Proc. International Symposium on Circuits and Systems,* 1979, pp. 482–485.
24. Preas, B. T. Benchmarks for cell-based layout systems. *Proc. 24th ACM/IEEE Design Automation Conference,* 1987, pp. 319–320.
25. Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical Recipes—The Art of Scientific Computing.* Cambridge Univ. Press, London/New York, 1986.
26. Rose, J. LocusRoute: A parallel global router for standard cells. *Proc. 25th ACM/IEEE Design Automation Conference,* 1988, pp. 189–195.
27. Rutenbar, R. A. A class of cellular computer architectures to support design automation. Ph.D. thesis, Department of CICE, University of Michigan, 1984.
28. Rutenbar, R. A. Simulated annealing algorithms: An overview. *IEEE Circuit Devices Mag.* **1** (1989), 19–26.
29. Sechen, C., and Sangiovanni-Vincentelli, A. TimberWolf3.2: A new standard cell placement and global routing package. *Proc. 1986 Design Automation Conference,* June 1986, pp. 432–439.
30. Soukup, J. Circuit layout, *Proc. IEEE.* **69,** 10 (Oct. 1981), 1281–1304.
31. Soukup, J., and Royle, J. C. On hierarchical routing. *J. Digital Syst.* **5,** 3 (Mar. 1981).
32. Spiers, T. D., and Edwards, D. A. A high performance routing engine. *Proc. 24th ACM/IEEE Design Automation Conference,* 1987, pp. 793–799.
33. Stout, Q. F. Hypercubes and pyramids. In Cantoni, V., and Levialdi, S. (Eds.). *Pyramidal Systems for Image Processing, NATO ASI Series ARW.* Springer-Verlag, Berlin/New York, 1986.
34. Blank, T., Stefik, M., and van Cleemput, W. A parallel bit map processor architecture for DA algorithms. *Proc. 18th ACM/IEEE Design Automation Conference,* 1981, pp. 837–845.
35. Ting, B., and Tien, B. Routing techniques for gate arrays. *IEEE Trans. Computer-Aided Design IC Syst.* **CAD-3** (Oct. 1983), 301–312.
36. Weicker, R. P. Dhrystone: A synthetic systems programming benchmark. *Comm. ACM* **27,** 10 (Oct. 1984), 1013–1016.
37. Winsor, D. C., and Mudge, T. N. Analysis of bus hierarchies for multiprocessors. *Proc. IEEE 15th Annual International Symp. Computer Architecture,* May 1988.
38. Yoshimura, T., and Kuh, E. S. Efficient algorithms for channel routing. *IEEE Trans. Computer-Aided Design IC Syst.* **CAD-1,** 1 (Jan. 1982), 25–35.
39. Zargham, M. R. Parallel channel routing. *Proc. 25th ACM/IEEE Design Automation Conference,* 1988, pp. 123–132.

Received February 15, 1989; revised October 16, 1989

OYEKUNLE OLUKOTUN received the B.S. degree in electrical engineering in 1985 and the M.S. degree in computer engineering in 1987 from the University of Michigan. Currently, he is pursuing a Ph.D. also at the University of Michigan. His research interests include parallel algorithms for computer aided design of integrated circuits and tools for the analysis, design, and verification of high-speed digital systems.

TREVOR MUDGE received the B.Sc. degree in cybernetics from the University of Reading, England, in 1969, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively. While at the University of Illinois he participated in the design of several special purpose computers and did research in computer architecture. Since 1977, he has been on the faculty of the University of Michigan, Ann Arbor, where he has taught classes on logic design, CAD, computer architecture, and programming languages. He is currently an associate professor of electrical engineering and computer science and director of the Advanced Computer Architecture Lab. He is author of more than 80 papers on computer architecture, programming languages, VLSI design, and computer vision, and he holds a patent in computer aided design of VLSI circuits. In addition to his position as a faculty member, he is a consultant for several computer companies in the areas of architecture and languages. Trevor Mudge is a senior member of the IEEE, a member of the ACM, and a member of the British Computer Society.