A Preliminary Investigation into Parallel Routing on a Hypercube Computer *

O. A. Olukotun and T. N. Mudge

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

Abstract

This paper describes an experiment in which parallel routing is performed on a medium grained hypercube parallel processor having 64 processing elements. Each node is a complete 32-bit computer with 128 K-bytes of memory and is connected to the other nodes via a direct hypercube interconnection network. A new parallel routing algorithm was developed to exploit this parallel structure. It is a three step algorithm consisting of a global routing step, a boundary crossing placement step, and a detailed routing step. All steps can be performed in parallel. When applied to a standard benchmark it was able to route 95 % of the wires. The algorithm was also executed on a large mainframe computer using the same benchmark. The execution time was compared to that for the hypercube. The hypercube was about three times as fast.

1 Introduction

The routing of a collection of nets (sets of points) in a cellular grid, is one of the most computationaly difficult problems that one encounters in the design of VLSI circuits, PCB boards, and gate arrays. Maze routing, a technique originated by Lee in [1] to solve this problem, produces high quality results at the cost of long execution times and high storage requirements on serial computers. Channel routing is a routing algorithm that constrains the paths to predefined channels in an attempt to reduce the high computational requirements. It can also give good results if the number and location of the nets to be routed match the location and size of of the predefined channels. Usually channels are defined by the intervening space between previously placed modules. The quality of the final layout is dependant on this prior placement. The parallelism inherent in both of these routing algorithms could be exploited by a parallel computer to provide either shorter execution times or a higher quality result. This paper describes a hybrid channel-maze routing algorithm suitable for parallel processing. The algorithm is executed on a hypercube parallel processor and a large mainframe and the execution times are compared.

The problem of wire routing involves finding a path between members of a net on a cellular grid. The grid may represent the area of a VLSI chip, a gate array or a PCB board. Finding the best layout is, in general, a difficult (NP-complete) problem, that is further complicated by the constraints of a particular technology, such as restrictions in via position and adjacency of certain types of wires. Practical algorithms used to solve the wire routing problem use heuristic techniques with polynomial complexity that lead to near optimal solutions. The quality of the routing produced by such algorithms is evaluated by total wire length, maximum length of a net, number of vias used and overflow count (unroutable nets).

Maze routing, a simplification of the general wire routing problem, is a technique for finding the shortest path between two points in a rectilinear grid. The basic algorithm originated by Lee in [1] involves three main steps:

- **Wavefront Expansion**
  During this step each point in the grid is labeled with the length of the shortest path from the source to the point. At each expansion step the wavefront consists of labeled points, equidistant from the source, with adjacent free neighbors not yet labeled. The expansion consists of labeling these adjacent neighbors to create a new wavefront. Obstacles such as previously routed nets constrain the expansion of the wavefront. Wavefront expansion continues until the target point has been reached or there are no more grid points to expand.

- **Backtrace**
  This step traces the shortest path, along expanded grid points, from target to source. The path is labeled as a new obstruction to the routing of subsequent nets.

*This research was supported in part by a grant from Materials Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, Ohio 45433-6503.

- **Cleanup**
  In this step all expanded grid points that are not part of the new path are relabeled as free points to be used in the routing of other nets.

The maze routing algorithm is simple and offers two major advantages over other routing techniques. First, it guarantees that a source to target path will be found if one exists. Second, maze routing is flexible in the sense that routing paths can be optimized to achieve certain objectives, such as low via usage or low via congestion. These advantages come at the expense of long execution times and large storage requirements. Long execution times are due to the worst case $O(L^2)$ time complexity of the wavefront expansion step for wires of length L, and the fact that only a single net can be routed at a time. Large storage requirements are due to the grid representation. Furthermore, despite the maze routing algorithm's guaranteed path finding ability, there are no guarantees that all nets will be routed, as the routing of a net may be blocked by a previously routed net. Despite this shortcoming, maze routing lends itself well to the use of specialized parallel hardware to reduce execution time by exploiting the inherent parallelism of the wavefront expansion step. The following section examines some representative parallel hardware accelerators for maze routing.

2 Parallel Architectures for Maze Routing

Due to the two dimensional nature of the maze routing problem the most common processor configuration proposed for parallel maze routing hardware has been that of the two dimensional mesh of processors. However other architectures, such as raster pipeline subarrays have also been used [2]. Most approaches to the parallelization of maze routing aim to reduce the worst case time complexity of the time consuming wavefront expansion step from $O(L^2)$ to $O(L)$.

Examples of two dimensional meshes are the L-machine [3] and the SAM-machine [4]. Both of these machines are fine grained architectures in which a simple processor node, embedded in a large mesh of similar nodes, is assigned to mark each grid position. The maze routing is implemented with a high ratio of communication to computation at each processor and low average processor utilization. The Wire Routing Machine (WRM) [5] possesses a more powerful node processor and a smaller mesh and so can not afford the approach to maze routing taken by the L and SAM machines. Instead, to take advantage of the power of the node processor, the WRM uses a two phase routing algorithm. During the first phase, the routing grid is divided into cells which contain a number of grid points. A global routing from cell to cell is then determined. In the second phase, fine track wiring is made between grid points within the global route.

The NCUBE is an example of a general purpose parallel computer with a hypercube processor interconnection topology which can accomodate up to 1024 nodes in a 10-dimensional hypercube. A 4-dimensional hypercube is shown in Figure 1. Each node is a powerful custom 32-bit microprocessor with 128 K-bytes of memory and the ability to perform floating point arithmetic [6]. 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, the NCUBE/six. We have implemented a simple three step maze routing algorithm on the NCUBE/six with significant speedup over the same algorithm running on a large mainframe. Furthermore, due to the power and general purpose nature of the node processors, it is possible to implement a routing algorithm for the NCUBE with a considerable degree of sophistication in order to improve routing quality or to adapt the algorithm to a specific technology. The speed and flexibility make the NCUBE hypercube both a practical and a cost-effective solution to the maze routing problem.

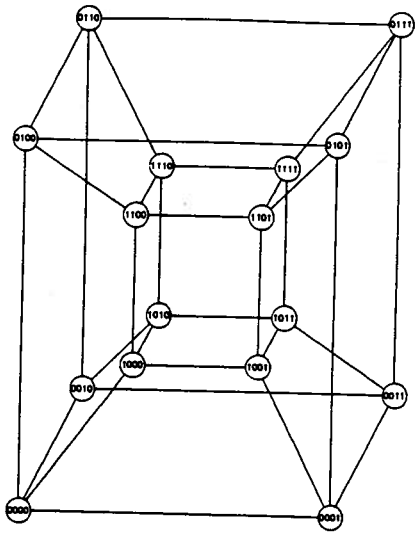

Figure 1: A 4-dimensional hypercube.

3 Hypercube Maze Routing Algorithm

One of the most important aspects in parallel algorithm design is the decomposition of the problem so that it maps efficiently onto the parallel architecture. Typically, this requires that the mapping does not result in the undue loading of a subset of the processing nodes, as this imbalance would degrade the overall performance of the algorithm. The decomposition should also minimize the ratio of interprocessor communication time to the computation time within the processors, as this will result in high efficiency for the overall computation. Unfortunately, there is no optimum solution to the decomposition of the maze routing problem.

To decompose the maze routing problem we have chosen the most obvious approach, which is to divide the grid into

as many square regions as there are node processors, N. If the grid is A square units in area, and there are $N = 2^d$ nodes, where d is the dimension of the hypercube used, then each node will be responsible for a partition or cell of the grid containing $\frac{A}{2^d}$ square units. At the edges of each cell there are crossings which connect the cell to its adjacent cells. Each edge has a fixed capacity of crossings in which nets can be routed. A node keeps the number of crossings available at the eastern and southern boundaries of the cell assigned to it. Given this decomposition of the routing problem the parallel routing algorithm consists of the following steps:

1. Global routing.
2. Boundary crossing placement.
3. Detailed maze routing.

Each of these steps will be described in detail in the sections that follow. The speedup of the algorithm over a conventional maze routing algorithm rests on the fact that the above steps can all be performed in parallel.

As noted, a 2-dimensional mesh may be mapped onto a hypercube connected parallel computer such as the NCUBE, and the parallel routing algorithm described above implemented. The algorithm does not make any use of the hypercube interconnection topology beyond that of a 2-dimensional mesh, however the preliminary work described here is intended to set the stage for further experimentation to develop algorithms that utilize the full power of the hypercube interconnection scheme.

3.1 Global Routing

The first phase of the routing algorithm is a global routing to assign nets to routing regions (cells) rather than detailed grid points. A global routing phase should increase the routing quality by globally optimizing net placement [7]. Global routing also serves as the first step in the decomposition of the routing problem into N independent routing problems. This phase requires communication among adjacent processors to perform the routing and to update the values of the edge capacities.

The global routing phase is performed under the following assumptions:

- all nets consist of two points (source-target points),
- all nets can be wired within their minimum global bounding rectangle.

The first assumption is a simplification of the general multipoint net routing problem, and will be relaxed in later work. The second assumption relies on the fact that in practice most nets are routable near their minimal lengths, and do not detour much beyond the minimum bounding rectangle [8]. In all cases the area enclosed by the global minimum bounding rectangle is always greater than or equal to that of the actual minimum bounding rectangle (see Figure 2).

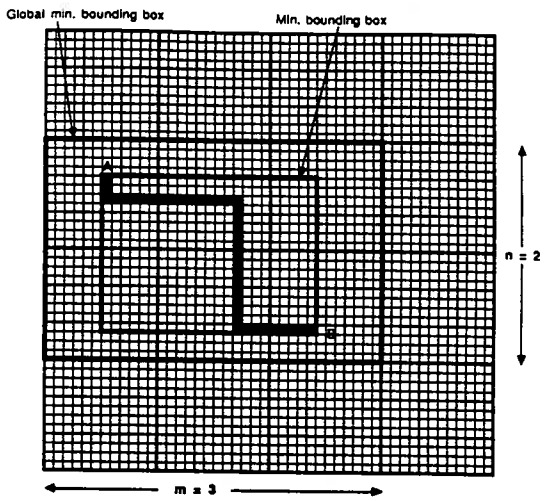

Figure 2: Routing grid showing cell boundaries.

If we wish to connect grid points A and B , shown in Figure 2, that reside in a global rectangle of n units in the x-direction and m units in the y-direction, there are N_R possible minimum global length routes from the cell containing point A to the cell containing point B, where G is given by

$$G = \begin{pmatrix} n+m-2 \\ n-1 \end{pmatrix}. \quad (1)$$

To explore these routes and find the one with the least boundary crossing cost, a global expansion phase is performed, using global routing messages, that is analogous to the wavefront expansion step of maze routing. A global routing message consists of seven fields, these are:

- The *net number* identifies which net a global route message belongs to.
- The *source* processor indicates where the message originated from.
- The *destination* or target processor indicates where an expansion message should terminate.
- The *direction* gives the direction of the destination processor in relation to source processor. The direction has a field to specify the x-direction, which can have a value of east or west, and a field for the y-direction, which can have a value of north or south.
- The *route bit vector* maintains a record of the direction, x or y, in which the message is advanced at each processor. The bit indicated by the *bit position* portion of the message is set in the route bit vector to "1" if the message is advanced in the x-direction and "0" if the message is advanced in the y-direction.
- The *bit position* identifies the current position in the route bit vector, it is also a measure of how far, in global cells, a message is from its origin.

- The *route cost* keeps a running total of the cost incurred by the message at each boundary crossing. The cost of crossing a cell boundary during global expansion is computed as an inverse exponential function of the edge capacity of the boundary.

A message also has a *type* associated with it. The two types of messages used in the global routing phase are expansion and backtrace.

The processor responsible for the cell in which the source point of the net resides initiates the global expansion phase by initializing the fields of a global expansion message and sending the message to its neighboring processors contained within the the minimum global bounding rectangle. On the reception of an expansion message a processor compares the source portion of the message with its processor location to determine if it is the target processor. If the processor is not the intended destination of the message, it adjusts the fields of the message before advancing the expansion message to its neighbors within the global bounding rectangle. If a processor receives an expansion message for a net whose source does not have either an x or y coordinate in common with the processor's location it waits for another message for that net to arrive from the other direction, x or y, before advancing the message with least cost. Once the target processor has been reached by an expansion message from both the x and y directions, the backtrace phase is initiated for the least cost message.

The global backtrace phase consists of retracing the path taken by the least cost expansion message from the source to the target processor. To accomplish this, the direction portion of the message is replaced with the direction of the source processor in relation to the the target processor. The route bit vector, scanned in the opposite direction to that used during expansion, is used to determine in which direction, x or y, to move in at each processor. The backtrace message traverses the path from the target processor to the source processor utilizing the route bit vector, the direction and the bit position to direct its course. At each cell boundary along the backtrace path crossings used by the net are claimed by reducing the value of the edge capacity of the boundary by one.

The global expansion phase generates M_E messages given by,

$$M_E = 2mn - m - n. \tag{2}$$

The backtrace phase generates M_B messages given by,

$$M_B = m + n - 2. \tag{3}$$

The total number of messages, M_G, generated for the global routing of a two point net is, thus

$$M_G = M_E + M_B = 2(mn - 1). \tag{4}$$

Besides the transmission time, each message has associated with it a certain amount of computation time which causes its transmission and arises from its reception. If we assume that any computation time that does not arise from the transmitting and receiving of messages is negligible in comparison with computation time that does, the number of messages generated is a relative measure of the time spent during the global routing phase.

Each processor is responsible for the global routing of nets whose source points are contained within the processor's cell. A processor initiates the global routing of a net until all nets for which it is responsible have been routed. This happens simultaneously in all processors. Although significant speedup is achieved by this parallel global routing scheme, it is possible that during the backtrace phase a backtrace message may arrive at a particular processor to discover that all the available crossing crossings had been claimed by other nets since the expansion phase. If this situation occurs a completely new global routing could be performed for the net, or a new path could be found from the cell where the block occurred to the source cell. Intelligent assignment of boundary crossing costs can minimize the occurrence of this problem. In our present algorithm there is no facility to recover from blocked boundary crossings as this problem did not occur while running the benchmark used to test this routing algorithm.

After a processor has completed the global routing of all the nets with source points within its cell it reports to the host. When all the processors have completed the global routing phase the host initiates the next phase of the routing algorithm.

3.2 Crossing Placement

In order to decompose the routing problem into a set of independent routing problems that can be solved in parallel, it is necessary for a processor and its four neighbors to decide on the placement of nets that cross their common boundaries. In a multilayer extension to this algoritm it would also be necessary to asign a crossing layer to each net. The crossing placement algorithm used to achieve this end embodies some ideas from [9]. Although fixing the crossing points is necessary in order to carry out the detailed routing without any interprocessor communication, it can restrict the solution space so that certain nets become unroutable. The crossing placement algorithm is designed to minimize this problem.

At the completion of the global routing phase each processor contains a list of nets with portions within its cell, including nets that are completely contained within its cell, and the borders across which these nets pass. A *strand* is defined, as in [9], to be a connected portion of a net within a cell together with the boundary crossings with which this portion connects. These strands represent the input to the crossing placement algorithm.

Initially all crossings have an undefined value outside the allowed range of crossing values. The crossing placement routine uses an iterative refinement method, in which each processor calculates the position of a crossing, on either its southern or eastern borders, based on a weighted average of the current position of the crossing, if it is defined, and the positions of the crossings as projected on the crossing border of strands to which the crossing is connected. The closer a crossing is to the one being placed the more weight it is

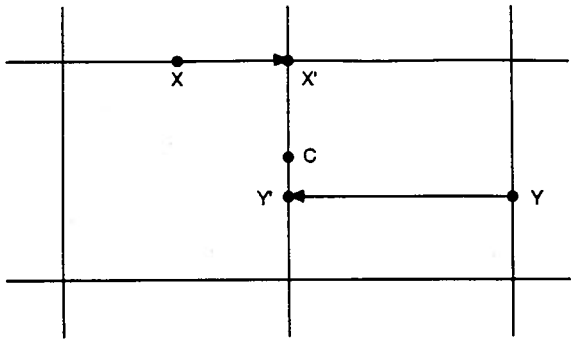

Figure 3: An example of crossing placement.

given. Crossings on an opposite border to the crossing being placed are given much more weight and source and target points even more weight. A crossing can become defined if a strand connects this crossing to one or more already defined crossings. Initially only source and target points are defined. This scheme has the effect of giving the source and target endpoints even more weight in determining the crossing placement of strands connected to them. Figure 3 shows an example of crossing placement. In this example crossing C, which is to be placed, will be connected to crossings 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 Y' is given more weight than X' as it is opposite C.

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 towards the east, while the second step begins with processors on the northern border and proceeds towards the south. After each net has been placed, message is sent to the appropriate neighbor processor with the new crossing position. At the completion of each step each procesor checks for strands that occupy the same crossing point. Any conflicts are resolved by moving the crossing of the strand belonging to the longer net to another position. The number of iterations the crossing placement algorithm executes is predetermined by the user. In practice, convergence occurs quickly, and weight given to crossings on opposite boundaries of a cell tend to cause the strands to form straight lines as one would expect. Howerver, conflicts may cause jogs in the wiring paths.

Each iteration of the crossing placement algorithm generates three messages for every crossing placement iteration. Thus the number of messages generated by a net, N_C, during crossing placement is given by,

$$M_C = 3I(m+n-2), \tag{5}$$

where I is the number of crossing placement iterations.

3.3 Detailed Routing

In the final step of the parallel routing algorithm each processor performs the detailed routing of its cell. This is done using a software maze router following the algorithm outlined in the Introduction and using the additional idea from [10] of storing the location of the wavefront grid points in a stack. This step is executed in parallel on all processors and requires no interprocessor communication.

4 PCB Benchmark Results

To test the parallel routing algorithm described above we have implemented it on a NCUBE/six hypercube with 64 processor nodes. The problem used is a single-layer Printed Circuit Board (PCB) benchmark developed by Blank in [11]. The benchmark consists of routing 200 nets with an average length $\approx$ 170 grid points in a 512 $\times$ 512 grid.

The completed PCB routing is shown in Figure 4. Table 1 shows the the execution times for the PCB benchmark run on hypercubes of order four (16 processors) and order six (64 processors). The time taken to load the processors with the routing information and the time to retrieve the finished result is included in the total time. From a comparison of the routing times (ignoring I/O time) we observe that the time for the order six hypercube is a roughly a factor of four reduced from that of the order four hypercube, this is in keeping with the domination of the total routing time by the detailed routing phase.

Using Equations 4 and 5 with estimates for m and n to scale the the times for steps one and two we can extrapolate the results from Table 1 to obtain the routing times on higher order hypercubes. Estimated routing times excluding I/O time are shown in Table 2. Timing results for the summation of all three steps are shown in Figure 5. From Figure 5 we see that the optimum size hypercube for the PCB benchmark using our routing algorithm is that of order six. For hypercubes of larger order, the detailed routing phase time becomes insignificantly small while the global routing phase, which now dominates the total routing time, grows by a factor of four for each factor of four increase in cube size. Furthermore, due to the reduced crossing capacity of each cell with increasing hypercube size there is greater possibility for boundary crossing blockage, as described in Section 4.1. To recover from this problem, should it arise, would add more time to the global routing phase. The times in Table 2 are best case and do not take into account time lost due to boundary crossing blockage. If a larger problem in terms of routing grid size were used we would continue to see speedup with larger orders of hypercube while at the same time reducing the possibility of boundary crossing blockage.

Figure 6 is a comparison of the hypercube router and the Amdahl 5860 executing a serialized version of the parallel routing algorithm. The Amdahl had roughly 100 users when this benchmark was performed. The CPU times represent the time spent in actual computation where as the elapsed time includes is wall-clock time and includes time spent performing

Hypercube Dimension	Routing Time (sec)			
	Step 1	Step 2	Step 3	Total
4-cube (16)	0.3	0.8	48.8	55.3
6-cube (64)	1.6	1.7	9.1	22.4

Table 1: NCUBE Routing Times for the PCB Benchmark.

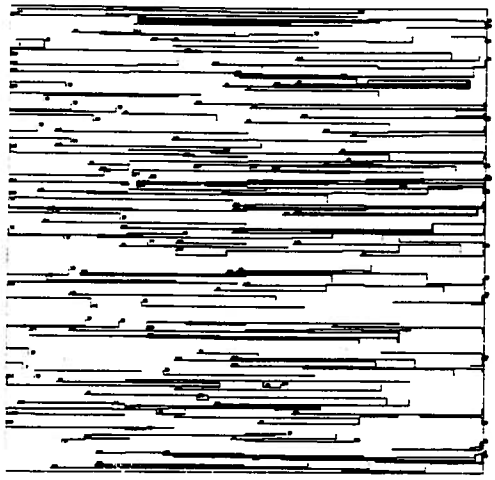

Figure 4: The PCB Benchmark.

I/O. The elapsed time and CPU time of the parallel version of the routing algorithm executing on the NCUBE/six is superior to the serial version of the algorithm executing on the Amdahl 5860 for both 16 and 64 cells. Notice, that the case for 64 cells executes faster in serial. This is because the worst case complexity of the wavefront expansion in the detailed routing has been reduced by a factor of 16 and the increase in complexity in the other two steps has not made up for this. Increasing the number of cells further will eventually result in an increase in execution time.

The hypercube routing algorithm was unable to complete ten nets yielding a completion rate of 95 % while running on 64 nodes. In all cases the nets were not completed due to poor crossing point assignment. Potential solutions to this problem would involve a more complex crossing placement algorithm or the ability to move crossings during the detailed routing phase. Of these two, the former is more appealing as it retains the separation between crossing placement and detailed routing. It is evident from Table 1 that more time could be devoted to crossing placement without greatly affecting the impressive speed of the hypercube router.

5 Conclusions

This paper has presented an algorithm for parallel routing on a hypercube parallel processor. The performance of this algorithm has been evaluated using a synthetically produced single layer, two-point net benchmark on an NCUBE/six hy-

Hypercube Dimension	Routing Time (sec)		
	Step 1	Step 2	Step 3
8-cube (256)	6.7	3.4	2.6
10-cube (1024)	26.8	6.8	0.6

Table 2: Estimated NCUBE Routing Times for the PCB Benchmark.

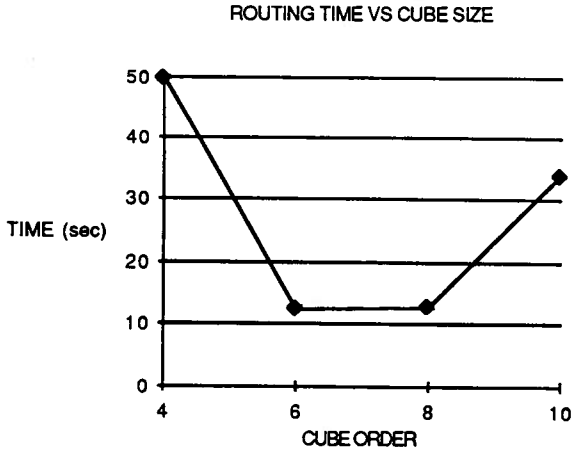

Figure 5: Hypercube Routing Times for the PCB Benchmark.

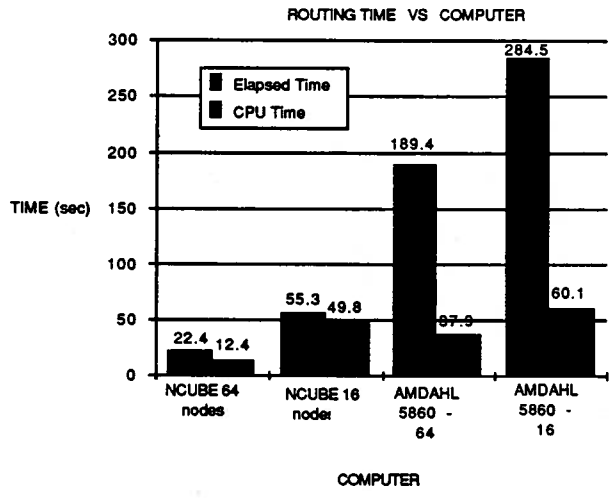

Figure 6: Comparison of Routing Times for the PCB Benchmark.

percube computer. The results are encouraging and suggest that a more general router based on this technique should be investigated.

References

[1] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Tran. on Electronic Computers*, vol. EC-10, Sep. 961, pp. 346–358, 1961.

[2] R. A. Rutenbar, T. N. Mudge and D. E. Atkins, "A class of cellular architectures to support physical design automation," *IEEE Tran. CAD of IC's and Systems*, vol. CAD-3, no. 4, pp. 264–278, Oct. 1984.

[3] M. A. Breuer and K. Shamsa, "A hardware router," *Jour. of Digital Systems*, vol. IV, issue 4, pp. 393–408, 1981.

[4] T. Blank, M. Stefik and W. van Cleemput, "A parallel bit map processor architecture for DA algorithms," *Proc. 18-th Design Automation Conf.*, pp. 837–845, June 1981.

[5] S. J. Hong and R. Nair,"Wire routing machines—New tools for VLSI physical design," *Proc. of the IEEE*, vol. 71, no. 1 pp. 57-65, Jan 1983.

[6] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley and J. Palmer, "Architecture of a Hypercube Supercomputer," *Proc. of Int. Conf. on Parallel Processing*, pp. 653–660, Aug. 1986.

[7] R. A. Rutenbar, "*A Class of Cellular Computer Architectures To Support Design Automation*," Ph. D. thesis, Dept of CICE, University of Michigan, Ann Arbor, MI., Sep. 1984.

[8] R. A. Rutenbar, "Systolic routing Hardware: performance evaluation and optimization," submitted to *IEEE Tran. CAD of IC's and Systems.*

[9] R. L. Rivest, "The PI (placement and interconnect) System," *Proc. 19-th Design Automation Conference*, pp. 418–424, 1982.

[10] J. H. Hoel "Some variations of Lee's algorithm," *IEEE Trans. Computers*, vol. C–25, pp. 19–24, Jan. 1976.

[11] T. Blank, *A Bit Map Architecture and Algorithms for Design Automation*, Ph. D. Thesis, Dept. of E.E., Stanford Univ., Stanford CA, Sep. 1982.