The Journal of Supercomputing, 2, 55–79 (1988)
© 1988 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

# Maze Routing on a Hypercube Multicomputer

YOUNGJU WON and SARTAJ SAHNI
*Computer Science Department, 136 Lind Hall, University of Minnesota, Minneapolis, MN 55455*

(Received May 1987; final version accepted March 1988)

**Abstract.** The implementation of Lee's maze routing algorithm on an MIMD hypercube multiprocessor computer can follow several plausible mappings and synchronization strategies. These are evaluated experimentally on an NCUBE/7 hypercube computer with 64 processors. Different grid partitioning and mapping strategies result in a different balance between computation and communication time. The total routing time is significantly impacted by the synchronization and termination detection scheme used. Further, by rearranging the computation, it is possible to overlap much of the interprocessor communication with the computation and realize a significant reduction in the overall run time. By choosing the right partitioning and synchronization scheme and by overlapping computation and communication, a good speedup is obtained on large routing grids.

## 1. Introduction

Lee's maze router is a popular wire routing algorithm. In the single layer case, the wiring surface is represented by a grid, as shown in Figure 1. Some of the cells are blocked (shown as shaded cells in Figure 1a) while others are available for routing. There are two specially designated cells: *s*, the source cell, and *t*, the target. These cells are the end points of a wire that is to be routed using only those cells that are not blocked. The wire begins at *s*, passes through available cells, and finally reaches *t*. A wire can be routed from one cell to the next by crossing a cell boundary (but not through a cell corner). One may assume that cells *s* and *t* are not blocked. The objective is to find a shortest route from *s* to *t*. In this paper, we consider the single layer case only.

Lee's algorithm for maze routing is a three-phase algorithm. These phases are *front wave expansion, path recovery,* and *sweeping*. During *front wave expansion,* a breadth-first search beginning at *s* is performed. Cells that are one unit from *s* are labeled, then those two units from *s* are labeled, then those three units from *s* are labeled, and so on. This labeling continues until the target cell *t* is reached. Blocked cells are not labeled during *front wave expansion.* Figure 1b shows the effect of the *front wave expansion* phase on the initial configuration. We use the four labels →, ←, ↓, and ↑ to point to the cell from which we reached the current cell. Thus, all four cells adjacent to *s* have a label that points to *s*.

If the front wave expansion reaches the target cell *t*, the *path recovery* phase begins. This involves backtracking from *t* to *s* by simply following the arrow labels from *t* to *s* (see Figure 1b). Now the wire path has been identified. Before the next wire can be

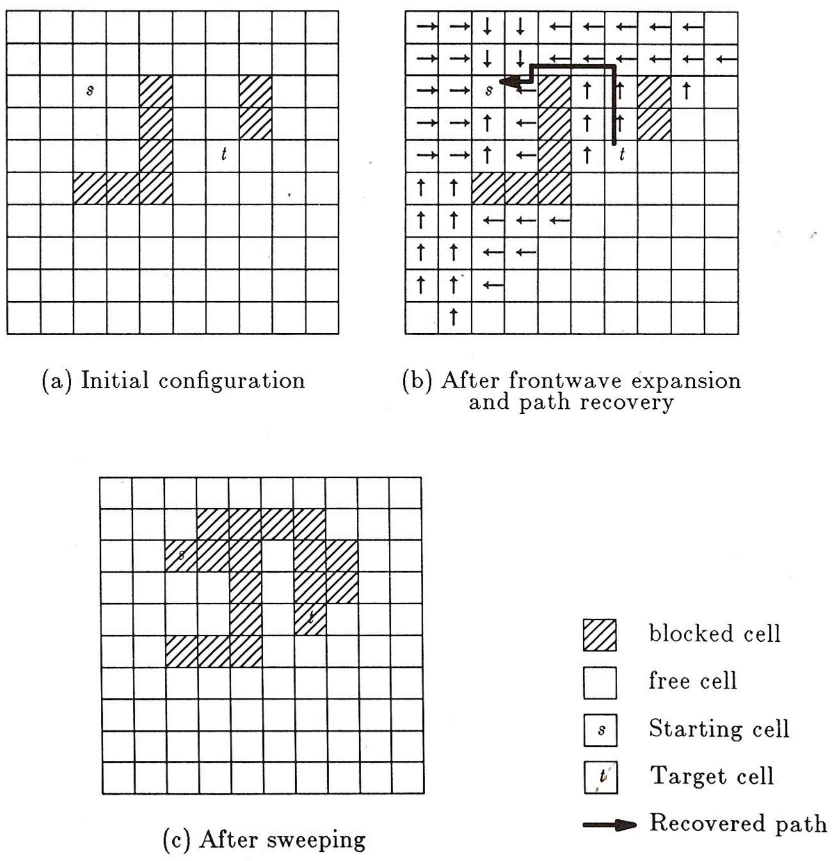


*Figure 1.* Maze routing phases.

routed, this wire path must be blocked and all arrow labels cleared from the grid. This is done in the *sweeping* phase, which is similar to the *front wave expansion* phase. Figure 1c shows the configuration after *sweeping*.

The complexity of Lee's router lies in the front wave expansion and sweeping phases. Since these are similar, we discuss the former only.

Since grids that represent realistic routing surfaces are quite large and since many wires have to be routed, Lee's maze router consumes a large amount of computer time in practice. Hence, it is desirable to find a suitable parallel implementation. Earlier attempts at fast realizations of Lee's router have focused on the development of special purpose hardware. For example, a cellular mesh connected processor array is proposed in Blank, Stefik, and van Cleemput [1981]; a processor pipeline is proposed in Mudge et al. [1982]; an iterative processor array has been developed in Iosupovici [1986]; a new method to map the grid onto a multiprocessor aray is developed in Suzuki et al. [1986]; and an architecture consisting of three processor pipelines is proposed in Won, Sahni, and El-Ziq [1987].

This paper explores the possibility of using a commercially available multiprocessor computer for routing. Successful implementation of Lee's router on our target computer, an MIMD hypercube computer, requires us to consider the following: mapping the routing grid onto the hypercube and synchronization.

Section 2 gives a brief overview of the architecture of the NCUBE hypercube computer and Section 3 provides a high-level description of the multiprocessor Lee's router. The next section considers different possibilities for mapping the routing grid onto a hypercube. Sections 5 and 6 examine the synchronization issue. Finally, we present experimental results that allow us to compare the various plausible implementations of Lee's router.

## 2. Architecture of the NCUBE Hypercube

A detailed description of the architecture of NCUBE's hypercube appears in Palmer et al. [1986]. Here, we review only those features that are relevant to the development of the remainder of this paper. The hypercube multiprocessor is an MIMD computer consisting of a host processor with local memory, node processors with their local memory, and external memory (Figure 2).

Each node has a custom 32-bit, 2-MIP, 0.5-MFLOP processor and a local memory of either 128K bytes or 512K bytes. The node processors are interconnected using the binary hypercube topology. Figure 3 shows this for a four-node and an eight-node hypercube. The NCUBE/AT (hypercube for IBM AT) supports up to 16 processor nodes, the NCUBE/7 supports up to 128 nodes, and the NCUBE/10 may have up to 1024 nodes.

The hypercube of node processors operates essentially as a peripheral attached to the host processor. Currently, the high-level programming language support includes FORTRAN and C. Both languages have been extended to allow for host-to-node and node-to-node communication. Neither language has compilers that perform automatic parallelism detection or multiprocessor problem decomposition. The programmer must provide a program for the host as well as one for each node. Typically, all the

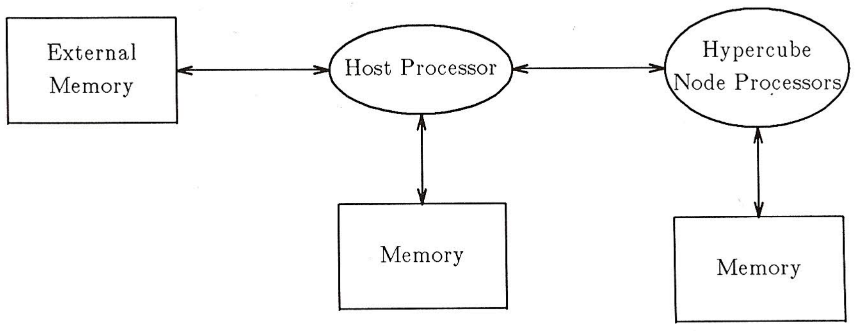
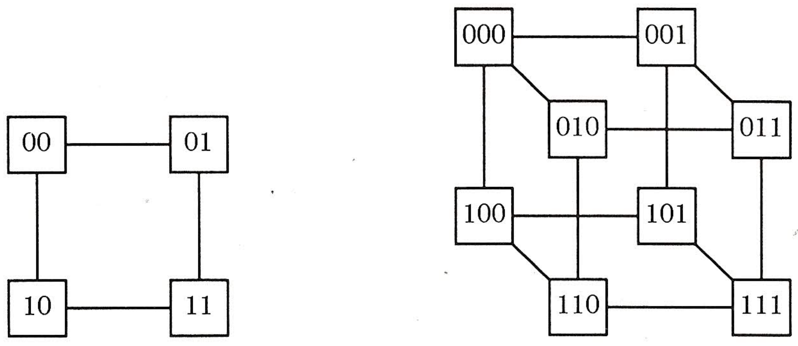


*Figure 2.* Hypercube multiprocessor.



(a) 4 node hypercube (b) 8 node hypercube

*Figure 3.* Hypercube topology.

hypercube nodes run the same program, though this is not necessary. In addition, all required synchronization must be done explicitly by the programmer. This is accomplished by message passing. It takes approximately $447 + 2.4\,L\,\mu s$ to transfer $L$ bytes between adjacent hypercube processors [Dunigan 1987]. By comparison, two 2-byte integers can be added in $4.3\,\mu s$.

The host program reads the node programs and data from the external memory (disk), farms these out to the appropriate node processors, communicates with the node processors while they are computing (if necessary), receives the results from the node processors upon completion, and writes out to the printer or to the disk. In a multidisk environment, it is possible for the node processors to directly access the disks.

## 3. Hypercube Implementation

A high-level description of a multiprocessor version of the front wave expansion phase of Lee's router is given in Figure 4. As can be seen, the basic strategy is to partition the $n \times n$ routing grid into $k$ parts where $k$ is the number of node processors. Each grid partition is assigned (or mapped) to a node processor. Each node processor performs the front wave expansion for the cells in the grid partition assigned to it.

To facilitate this front wave expansion, each processor maintains a queue of front wave cells in its grid partition. During front wave expansion, each cell in the queue is expanded. This involves examining the cells to its north, south, east and west on the routing grid. Some of these cells are in the processor's grid partition while others are in grid partitions assigned to other processors. For those in the local partition, we may complete the front wave expansion by labeling the unblocked cells and placing them in the internal queue (IQ) for later expansion. Cells not in the local partition are stored

**Step 1** : **[Grid partitioning and mapping]** Partition the $n \times n$ routing grid into $k$ parts and assign one partition to each of the $k$ node processors.

**Step 2** : **[Front wave expansion]** Each processor that has a grid cell on the current front wave expands the front wave. This expansion may require communicating with other processors as the cells adjacent to the front wave cell being expanded may be in different processors. All communication requests are saved.

**Step 3** : **[Inter processor communication]** Each processor sends its communication packets to the destination processor.

**Step 4** : **[Process communication packets]** Each processor examines the packets it receives and labels the front wave cells contained in these packets.

**Step 5** : Repeat steps 2, 3, and 4 until either the target cell is reached or the new front wave has no cells in it.

*Figure 4.* Multiprocessor front wave expansion.

in a send queue (SQ) for later transmission to the proper processor. Because of the high start-up time associated with a message transfer, it is faster to send a long message rather than several short ones. Once the front wave cells have been examined in this way, each node processor transmits the cells in its send queue to the processors assigned to them. These are received by the destination processors and stored in their receive queues (RQ). If the cells received are unlabeled and unblocked, they are then labeled and added to the internal queue (IQ).

It is well known that every $2^d$ node hypercube has embedded in it a $2^{\lceil d/2 \rceil} \times 2^{\lfloor d/2 \rfloor}$ two-dimensional mesh [Saad and Schultz 1985]. Figure 5 shows such an embedding for 4-, 8-, and 16-node hypercubes.

As we will later see, our mapping of the grid partition into the hypercube processors will require only interprocessor communication corresponding to a mesh. Keeping this in mind, we view a node processor as in Figure 6. The RQ and SQ are actually four separate queues each: one for each of the four transmit neighbors. Other interconnection patterns may be required by some of our synthronization schemes. The required queue size is a function of the size of the local front waves.

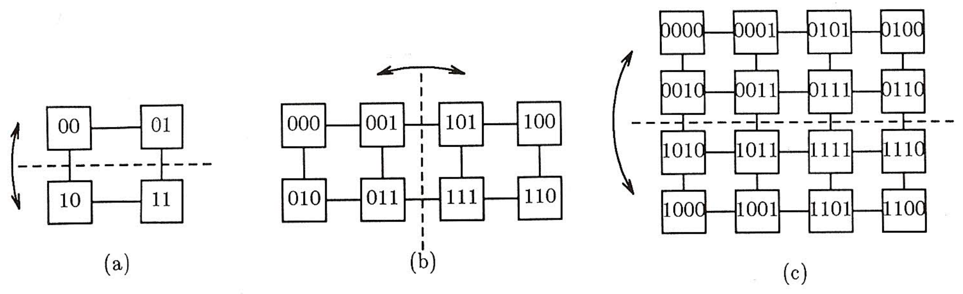
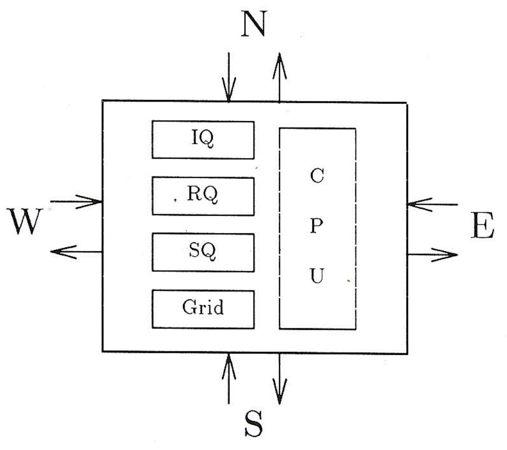


*Figure 5.* Meshes embedded in a hypercube (showing growth with dimension).



| | |
|---|---|
| IQ | : queue of internal front wave cells |
| RQ | : queue of front wave cells received from neighbors |
| SQ | : queue of front wave cells to be sent to neighbors |
| Grid | : local grid partition |
| ↑,↓ | : node to node connections for interprocessor communication |

*Figure 6.* Local view of node processor.

## 4. Grid Partitioning and Mapping

As described in the previous section, we utilize only an embedded mesh of the hypercube. Further, we assume that the number of grid cells $n^2$ is significantly greater than the number of node processors $k$. Since the node processors form a hypercube, $k$ is a power of 2. $k = 2^d$ where $d$ is called the *dimension* of the hypercube.

In discussing our grid partitioning and mapping strategy, for simplicity we assume that $n$ is a power of 2, $n = 2^p$ for some integer $p$ where $p \geqslant d/2$. Suppose that $n = 8$

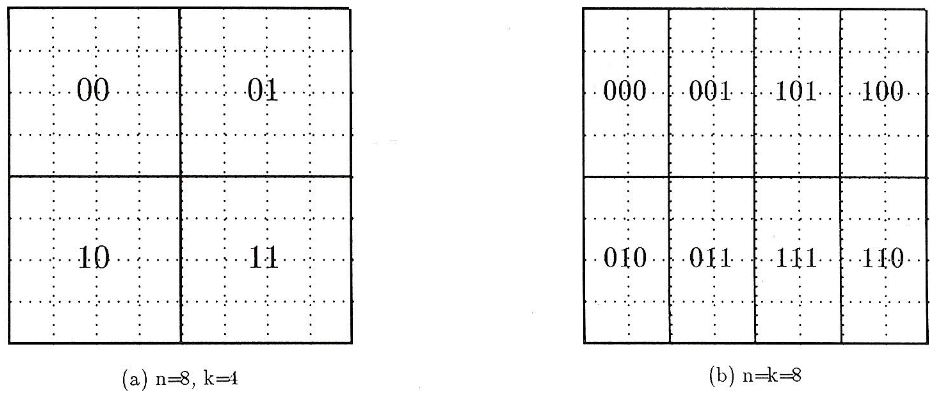


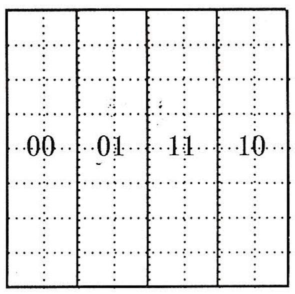
*Figure 7.* Grid partitioning.

and $k = 4$. The $8 \times 8$ grid may be partitioned and mapped onto the four processors, as shown in Figure 7a. Figure 7b shows a possible partitioning and mapping when $n = k = 8$. Each partition is labeled with the processor to which it is assigned. As can be seen, the neighbor partitions of any partition are assigned to node processors that are adjacent in the hypercube connection. This partitioning strategy may be formally defined as follows:

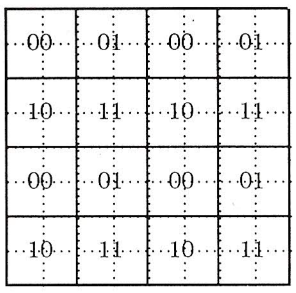
*Partitioning Strategy 1*: Cover the $n \times n = 2^p \times 2^p$ grid with rectangles of size $2^{\lfloor d/2 \rfloor} \times 2^{\lceil d/2 \rceil}$. Each rectangle in the cover defines a partition of the grid. The partitions are mapped to processors in such a way that partitions that are adjacent in the grid are mapped to processors that are adjacent in the hypercube.

A grid cell is said to be on a partition boundary if at least one of its immediate neighbor cells is in a different partition. It is easy to see that only boundary cells have a potential to cause interprocessor communication during front wave expansion. To reduce interprocessor communication one may attempt to reduce $B$, the number of boundary cells. For example, in Figure 7a, $B = 7 \times 4 = 28$ whereas in Figure 7b, $B = 5 \times 4 + 8 \times 4 = 52$. Some other ways to partition and map an $8 \times 8$ grid onto $k = 4$ processors are shown in Figure 8. The values for $B$ are $16 \times 3 = 48$ (Figure 8a) and $64 - 4 = 60$ (Figure 8b). It is easy to see that partitioning strategy 1 results in a partition with the lowest $B$ value.

Although $B$ is smallest when partitioning strategy 1 is used, this may not yield the best performance. To see this, consider Figure 7a. Suppose that $s$ is at the top left corner grid cell and $t$ at the bottom right corner grid cell. Assume there are no blockages. For the first several cycles, the front wave is confined to processor 00, so the remaining processors are idle. During the last several cycles, the front wave is confined to processor 11; once again, the remaining processors are idle. However, with the partitioning and mapping shown in Figure 8b, the idle time for processors is reduced. Hence, there is a trade-off between processor utilization and interprocessor communication which may be studied experimentally. For this, partitioning strategy 1 is generalized as below:



(a) n=8, k=4



(b) n=8, k=4

*Figure 8*. Other grid partitionings.

*Generalized Partitioning Strategy:* Let $h$ and $w$ be the height and width, respectively, of a covering rectangle. Cover the $n \times n$ grid with rectangles of dimension $h \times w$ beginning at the top left corner and proceeding from left to right, top to bottom.

For simplicity, we confine ourselves to the case where both $h$ and $w$ are a power of 2 and $h = w$ when the hypercube dimension $d$ is even and $h = 2w$ when $d$ is odd. As before, we assume that $h \leqslant n$. Because of these assumptions, the width $w$ of the covering rectangle completely characterizes it. As the width is increased, the number of boundary cells decreases, and hence interprocessor communication is expected to decrease. However, as the width is decreased, processor utilization is expected to increase. While other partitionings are possible, we do not expect much gain on random grids from partitionings that are not nearly square. For grids with known distribution of blockages a multigrid partitioning will work better.

Note that since we require the number of rectangles in the cover to be at least equal to the number $k$ of processors, $w$ must be in the range

$$1 \leqslant w \leqslant \frac{n}{\lfloor\sqrt{k}\rfloor} = 2^{p-\lceil d/2\rceil}.$$

With these bounds, the number of rectangles in the cover is in the range $[k, n^2]$.

## 5. Synchronization

A front wave expansion cycle consists of one execution of lines 2, 3, and 4 of the algorithm of Figure 4. When all the processors have completed the first cycle, the front wave consists of cells that are distance 1 from $s$; when the second cycle is complete, the front wave is made up solely of cells distance 2 from $s$; when cycle $q$ is complete, the front wave consists solely of cells that are $q$ units from $s$. One way to ensure that the multiprocessor algorithm has found a shortest path to $t$ whenever an $s$ to $t$ path exists is to perform a global synchronization at the end of each cycle. That is, no

| Host | Node |
|---|---|
| 1. Enable one cycle of *front wave expansion*; | 1. **If** enable received **then** perform steps 2, 3, and 4 of Figure 3.1 ; |
| 2. **If** path found **then** disable node processors and proceed to *path recovery*; | 2. Report result (i.e., target reached, local front wave queue is empty) to host; |
| 3. **If** new front wave is empty **then** terminate; | 3. Go to step 1; |
| 4. Go to step 1; | |

*Figure 9.* Synchronization by host.

processor begins cycle $q$ until all have completed cycle $q - 1$. This global synchronization may be performed by the host or by the node processors themselves.

## *5.1. Synchronization by Host*

When the host processor is used to perform global synchronization at the end of each cycle, the host and node processor programs take the form given in Figure 9.

For each cycle of front wave expansion, $k = 2^d$ enable messages are sent from the host to the node processors and $k$ completion messages are sent form the node processors to the host. If the shortest path is of length $l$, then approximately $2kl$ messages are transmitted. Since in this case the host transmits and receives messages serially, this method could produce a bottleneck. Hence, we need to consider ways to reduce the synchronization overhead.

Step 4 of the *front wave expansion* phase (Figure 4) performs local synchronization. A processor cannot begin cycle $q$, $q > 1$, expansion until it receives the results of the cycle $q - 1$ expansion from its north, south, east, and west processors. Consequently, global synchronization is not required to ensure correctness. However, without some form of synchronization, it is difficult to terminate the algorithm when there is no $s$ to $t$ path. When such a path is detected, the node processor that contains the $t$ cell signals the host. However, when there is no such path, it is necessary for all processors to notify the host that their front waves are empty. There are two ways to accomplish this. One is to do a global synchronization, as shown in Figure 9, but less frequently, as described below.

Let the grid coordinates of $s$ and $t$ be $(x_s, y_s)$ and $(x_t, y_t)$, respectively. The Manhattan distance $M$ between $s$ and $t$ is $|x_s - x_t| + |y_s - y_t|$. No $s$ to $t$ path is shorter than $M$. Hence the first synchronization could be done after $M$ cycles. That is, each processor does $M$ cycles of front wave expansion. Upon completion, it signals

Host

```
1. counter = 1;
2. Enable front wave expansion;
3. If path found then disable node processors and proceed to path recovery;
4. If signal(+1/-1) received then
    4.1. counter = counter + signal;
    4.2. If counter = 0 then
            wait Δ time;
            if no signal then terminate;
5. Go to step 3;
```

Node

```
1. If enable received then perform steps 2, 3, and 4 of Figure 3.1 ;
2. If target reached then report it to host;
3. If front wave changes from non empty to empty then send -1 signal to host;
4. If front wave changes from empty to non empty then send +1 signal to host;
5. Go to step 2;
```

*Figure 10.* Counter termination.

| Host | Node |
|---|---|
| 1. Enable *front wave expansion* in processor 0; | 1. **If** this is node 0 **then** wait for *enable* signal from host; |
| 2. **If** path found **then** enable *path recovery* **else** terminate; | 2. **repeat** |

```
Node
1. If this is node 0 then
     wait for enable signal from host;
2. repeat
2.1.    If this is node 0 then enable remaining
        processors else wait for enable signal
        from node 0;
2.2.    Perform steps 2, 3, and 4 of Figure 3.1;
2.3.    If this is node 0 then receive result from
        other nodes else report result to node 0;
2.4.    If this is node 0 then
          examine results from all nodes;
          If path has been found or if all local
          front waves are empty then signal
          host and terminate else enable all
          nodes for next cycle;
3. until false;
```

*Figure 11.* Global cycle synchronization by node 0.

the host. The host, upon receiving signals from each of the $k$ processors, then enables another $\delta$ cycles, unless a path is already found or all processors have an empty front wave. Following synchronization after these $\delta$ cycles, another $\delta$ cycles may be enabled and so on. We will call this *$\delta$–synchronization*, with $\delta$ designated as a user-selected parameter. However, if $\delta$ or $M$ is too large, then unsuccessful attempts to route from $s$ to $t$ may take more time than necessary. In a successful route only a small part of the last $\delta$ cycles may be useful. If $\delta$ is too small, the synchronization overhead slows the algorithm.

Another solution to the termination detection problem is to have each processor send a $+1$ to the host when its local front wave changes form empty to nonempty and a $-1$ when it changes form nonempty to empty. (The local front wave consists of cells in IQ and unlabeled/unblocked cells in RQ; its size may be determined after step 4 of Figure 4.) The host begins with a counter of 1 since initially only the processor with cell $s$ has a nonempty local front wave. When the host receives a $+1$ or $-1$ from a node processor, it adds it to the counter. When the counter is 0, it is possible that messages in transit between node processors could cause the previously empty front wave of a processor to become not empty. So, it is necessary for the host to wait $\Delta$ time units to allow for a node processor to receive its messages and transmit a $+1$ to the host. If the counter remains 0 for at least $\Delta$ time, then every node processor has an empty front wave and there are no messages in transit. At this time we know there is no $s$ to $t$ path. This termination scheme is called *counter termination.* The host and node programs for *counter termination* are given in Figure 10. $\Delta$ is a characteristic of the particular multiprocessor computer in use.

### *5.2. Synchronization by Node Processors*

*Global cycle synchronization* (Figure 5.1), *δ–synchronization*, and *counter termination* can also be implemented using the node processors rather than the host. When global cycle synchronization and node 0 are used to perform synchronization, the host and node programs are as given in Figure 11.

When node 0 is to send an enabling signal to the remaining nodes, it may do so using a *tree expansion*. That is, node 0 sends the message to node 1; nodes 0 and 1 then send it to nodes 10 and 11, respectively; nodes 0, 1, 10, and 11 send it to nodes 100, 101, 110, and 111, respectively; and so on. This scheme takes $\log_2 k = d$ message cycles to transmit the enable signal. Another possibility is for node 0 to transmit the enable signal directly to each of the remaining processors. This requires $k - 1$ signal transfers, most of which will be to nodes not directly connected to node 0. We will call this the *direct transfer* method.

Nodes may transmit messages to node 0 either directly or by using the *tree expansion* in reverse. The *direct transfer* results in a bottleneck at node 0 when all nodes finish their work at approximately the same time. The *reverse tree expansion* will not perform well if the nodes at the tree leaves are the last to finish. The execution of *δ–synchronization* and *counter termination* is similar to that of *global cycle synchronization* (Figure 11).

## 6. Overlapping Communication and Computation

After sending its data packets (Step 3 of Figure 4), a processor is ready to do more work. However, the next step causes it to wait since there is a delay between data leaving a source processor and arriving at its destination. We can rearrange the algorithm, as shown in Figure 12, by introducing a step between the send data and

**Step 1** : **[Grid partitioning and mapping]** Partition the $n \times n$ routing grid into $k$ parts and assign one partition to each of the $k$ node processors.

**Step 2** : **[Inter processor communication]** Each processor sends its communication packets to the destination processor (front waves of distance $d$)

**Step 3** : **[Front wave expansion]** Each processor that has a grid cell on the current front wave expands the front wave ( of distance $d$). This expansion may require communicating with other processors as the cells adjacent to the front wave cell being expanded may be in different processors. All communication requests are saved for the next iteration.

**Step 4** : **[Process communication packets]** Each processor examines the packets it receives and labels and expands (as in step 3) the distance $d$ front wave cells contained in these packets.

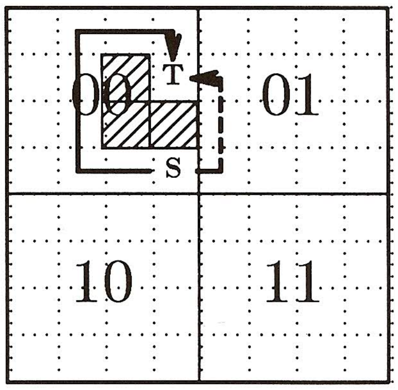
**Step 5** : Repeat steps 2, 3, and 4 until either the target cell is reached or the new front wave has no cells in it.

*Figure 12.* Modified multiprocessor front wave expansion.

receive data steps. In the first iteration, each processor sends distance 0 (i.e., null packets) front wave cells in Step 2, processes distance 0 front wave cells in Step 3, and receives and processes remaining distance 0 packets in Step 4. In the next iteration, distance 1 packets are sent in Step 2, the local distance 1 front wave cells are processed in Step 3, and the remaining distance 1 front wave cells are received from the neighbor processors and processed in Step 4 and so on.

## 7. Asynchronous Operation

The local synchronization being done in Step 4 (Figure 4) may be eliminated. This step may be modified so that when Step 4 is reached by a processor, it consumes all cells received so far but does not wait for messages that haven't yet arrived from its neighbors. Following this, it proceeds to Step 1. As a result of this modification, it is possible that the first time *t* is reached it is not reached by a shortest path (see Figure 13). To ensure termination only when a shortest path is reached, it is necessary to label cells by path length from *s* rather than by the directions $\rightarrow$, $\leftarrow$, $\uparrow$, and $\downarrow$. Now, whenever a cell is reached by a shorter path than it was previously reached by, the cell joins the front wave. Front wave expansion terminates when there is no front wave cell with a label less than that at *t*. A slightly better terminating condition can be obtained by utilizing the mesh distance from a node processor to the node processor containing *t* and the difference in the label at *t* and the smallest front wave label in the node processor.



$\longrightarrow$ Path found first time in *asynchronous* operation
$\dashrightarrow$ The shortest path

*Figure 13.* Example for *asynchronous* method.

Host

```
1. last path = empty; counter = 1;
2. Enable front wave expansion;
3. If new path found then
       last path = new path;
       send new path length to node processors;
       Go to step 3;
4. If signal(+1/-1) received then
   4.1. counter = counter + signal;
   4.2. If counter = 0 then
           wait Δ time;
           if signal then go to step 3;
5. shortest path = last path;
6. Disable node processors and exit;
```

Node

```
1. spath = $\infty$
2. If disable received then terminate;
3. If new path length received then spath = new path length
4. Perform steps 2, and 3 of Figure 3.1 with front waves of distance < spath
5. If packet received (at step 4 of Figure 3.1) then process front waves in communication packet of distance < spath
6. Report result (i.e., target reached, signal change of status of local front wave queue) to host.
7. Go to step 2;
```

*Figure 14.* Asynchronization method.

When this asynchronous scheme is used, the host and node processors take the form given in Figure 14. The additional space required by this scheme for the cell labels may make it impractical.

## 8. Experimental Results

Each strategy described above has been programmed in FORTRAN and run on an NCUBE/7 hypercube multiprocessor. We experimented with randomly generated grids which were obtained by placing a reasonable number of components (processors, memory, I/O devices, etc.) and introducing blockages to represent existing connections. The blockages represented 45 to 50% of the grid. The *nets* that were used were generated randomly, but were required to have an approximate length of

$$(\textit{row length of grid} + \textit{column length of grid})/4.$$

Each computing time reported below represents the average time to route 10 nets.

The first experiment studied the effect of the partition width $w$ using generalized partitioning. Figure 15 shows the average run time for different values of $w$ using a four-dimensional (i.e., 16-processor) hypercube, a $256 \times 256$ grid, and the *counter termination* scheme. Figure 16 shows the relative performance of different syn-

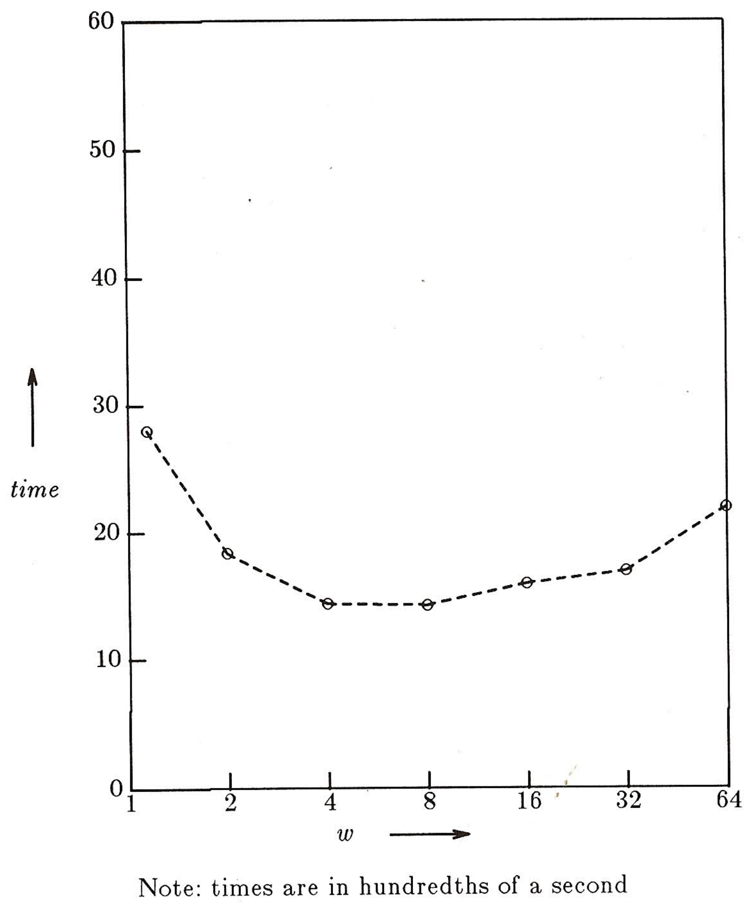


*Figure 15. Counter termination* with different partition widths ($w$) for a 256 × 256 grid using a four-dimensional hypercube ($h = w$).

chronization methods on a 512 × 512 grid using a five-dimensional hypercube. As is evident, the performance of *δ–synchronization*, the *counter termination*, and the *asynchronous* method is quite sensitive to the choice of $w$, whereas the performance of *global synchronization* is relatively insensitive to $w$. For each of the methods, there is an optimal value of $w$ such that using a smaller or larger value increases the run time. For the case of the 512 × 512 grid reported in Figure 16, these optimal $w$'s are 4 (*δ–synchronization*), 8 (*global synchronization* and *counter termination*), and 32 (*asynchronous* method). For the *asynchronous* method, using a $w$ of 4 rather than the optimal $w$ of 32 results in a run time that is more than four times the minimum. In the case of *counter termination* and *δ–synchronization* a bad choice of $w$ (in the tested range) could result in run times more than double the minimum. The run time of *global synchronization* varied less than 20% as $w$ was changed. Choosing the optimal

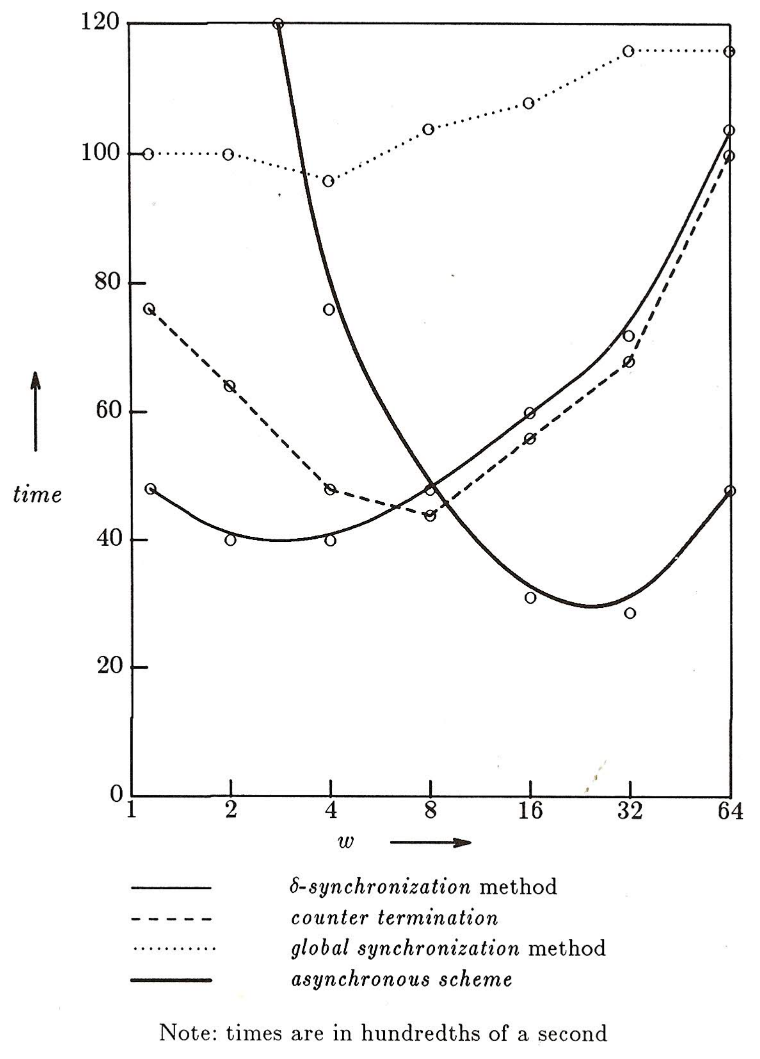


*Figure 16.* Behavior of each synchronization method with different partition widths (*w*) for a random 512 × 512 grid using a five-dimensional hypercube (*h* = 2*w*).

*w* is critical to good performance. The best value of *w* to use for different size grids is plotted in Figure 17. This best value is a nondecreasing function of the grid size.

The run times shown in Figure 18 were obtained using the best *w* values for each method and grid size. For *δ–synchronization*, node 0 is used as the synchronizing node since this is always quicker than using the host for this purpose. Also, the *tree expansion* method is used here because this was generally faster than *direct transfer*. However, for *counter termination* and the *asynchronous* method, the host processor is used along with *direct transfer*. After experimenting with several δ values, it was

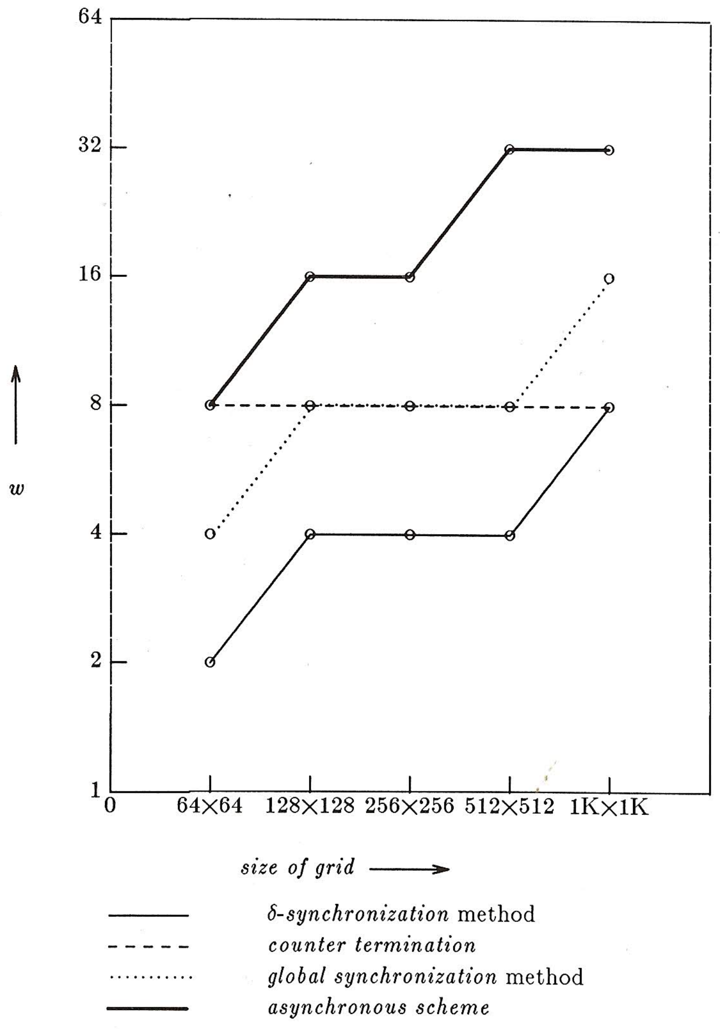


*Figure 17.* Optimal *w* value for each method.

determined that a $\delta$ in the range $[0.2M, 0.3M]$ ($M$ is the Manhattan distance between the wire end points) gave the best performance.

In all the experiments shown in figure 18, we used the nonoverlapping version of our algorithms. Larger grids could not be run on smaller hypercubes for lack of sufficient memory. The *global synchronization* method shows virtually no speedup with increase in hypercube dimension. In fact, for smaller grids, the run time actually increases with an increase in the hypercube dimension. This is due to the immense synchronization overhead associated with this method. By controlling this overhead, in *δ–synchronization*, we are able to obtain a significant reduction in the run time

| *data size* | hypercube dimension | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 64 × 64 | 6.0 | 6.5 | 7.0 | 7.8 | 8.5 | 9.5 | 10.6 |
| 128 × 128 | 9.6 | 9.5 | 9.3 | 10.4 | 11.1 | 11.3 | 12.7 |
| 256 × 256 | - | - | 35.4 | 30.7 | 30.5 | 32.0 | 36.1 |
| 512 × 512 | - | - | - | - | 110.9 | 102.5 | 100.2 |
| 1024 × 1024 | - | - | - | - | - | - | 276.3 |

**Table 1** *Global synchronization method*

| *data size* | hypercube dimension | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 64 × 64 | 1.5 | 2.2 | 2.3 | 2.2 | 2.3 | 2.3 | 2.4 |
| 128 × 128 | 7.3 | 7.2 | 6.8 | 6.5 | 6.2 | 6.0 | 6.8 |
| 256 × 256 | - | - | 26.3 | 18.4 | 15.6 | 10.9 | 11.2 |
| 512 × 512 | - | - | - | - | 65.2 | 40.5 | 30.1 |
| 1024 × 1024 | - | - | - | - | - | - | 84.0 |

**Table 2** *$\delta$ synchronization scheme*

| *data size* | hypercube dimension | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 64 × 64 | 1.6 | 2.1 | 2.2 | 2.2 | 2.3 | 2.6 | 3.7 |
| 128 × 128 | 7.3 | 7.0 | 6.5 | 6.0 | 6.4 | 6.7 | 7.5 |
| 256 × 256 | - | - | 23.1 | 16.9 | 14.3 | 12.8 | 13.8 |
| 512 × 512 | - | - | - | - | 60.1 | 41.2 | 41.5 |
| 1024 × 1024 | - | - | - | - | - | - | 89.1 |

**Table 3** *Counter termination*

| *data size* | hypercube dimension | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 64 × 64 | 2.1 | 2.3 | 2.3 | 2.4 | 2.3 | 2.5 | 3.8 |
| 128 × 128 | 9.3 | 9.2 | 6.3 | 5.5 | 6.1 | 6.9 | 9.3 |
| 256 × 256 | - | - | 22.2 | 14.3 | 10.9 | 11.2 | 15.3 |
| 512 × 512 | - | - | - | - | 56.3 | 35.8 | 35.4 |
| 1024 × 1024 | - | - | - | - | - | - | 80.1 |

**Table 4** *Asynchronous scheme*

Note : all times are in hundredths of a second

*Figure 18.* Run times for different control schemes.

relative to that of *global synchronization*. Further, on larger grids, say 512 × 512, we are able to get good speedup. When dimension is 4, the run time is 652 ms. So, with a dimension 5 hypercube the best we can expect is 326 ms. The observed time is 405 ms. The speedup is 1.6 rather than the theoretical maximum of 2. The theoretical maximum speedup in going from dimension 4 to dimension 6 is 4. The speedup

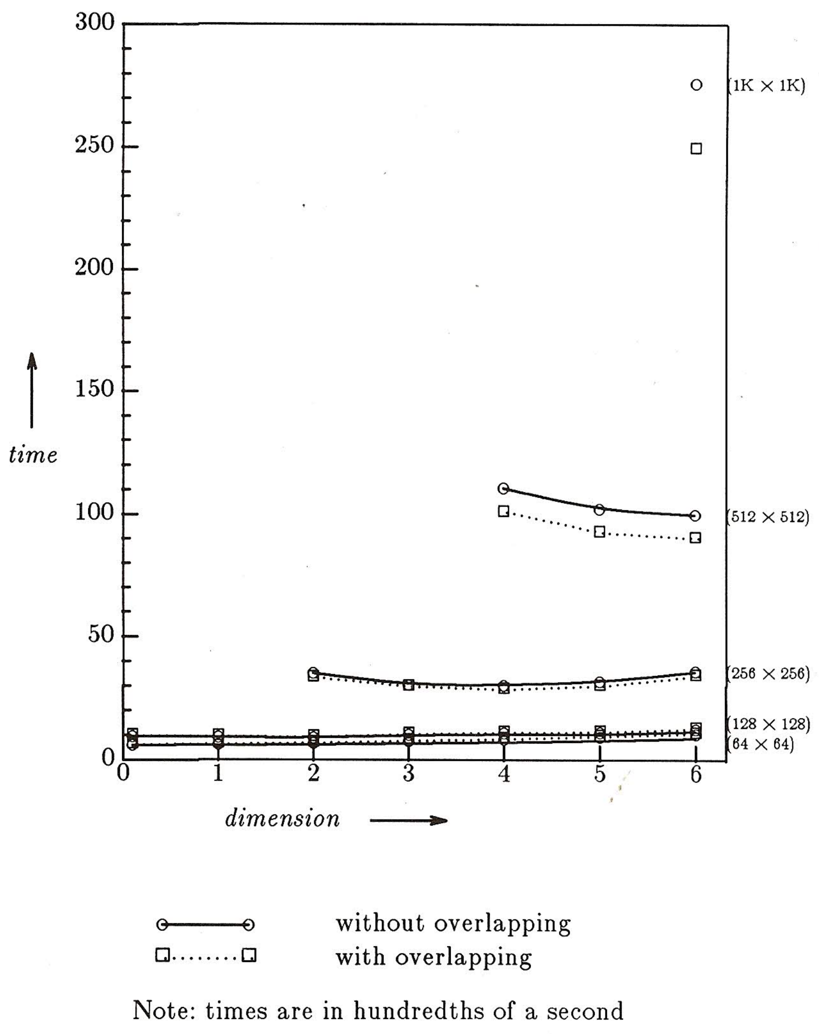


*Figure 19a.* Run time of *global synchronization* method.

achieved by *δ–synchronization* is 652/301 = 2.17. This is still quite good when one considers the wave front is not evenly distributed over the processors. The performance of *counter termination* is comparable to that of *δ–synchronization.* The *asynchronous* scheme was the fastest on 1K × 1K grids.

We next tried to ascertain what benefits, if any, would result from using the overlapping strategy outlined in Section 6. Figure 19 compares, with the exception of the *asynchronous* scheme, the run times obtained using overlapping with those tabulated in Figure 18. The *asynchronous* scheme automatically overlaps computation and communication since it does not wait for data to arrive. Rather, it consumes the

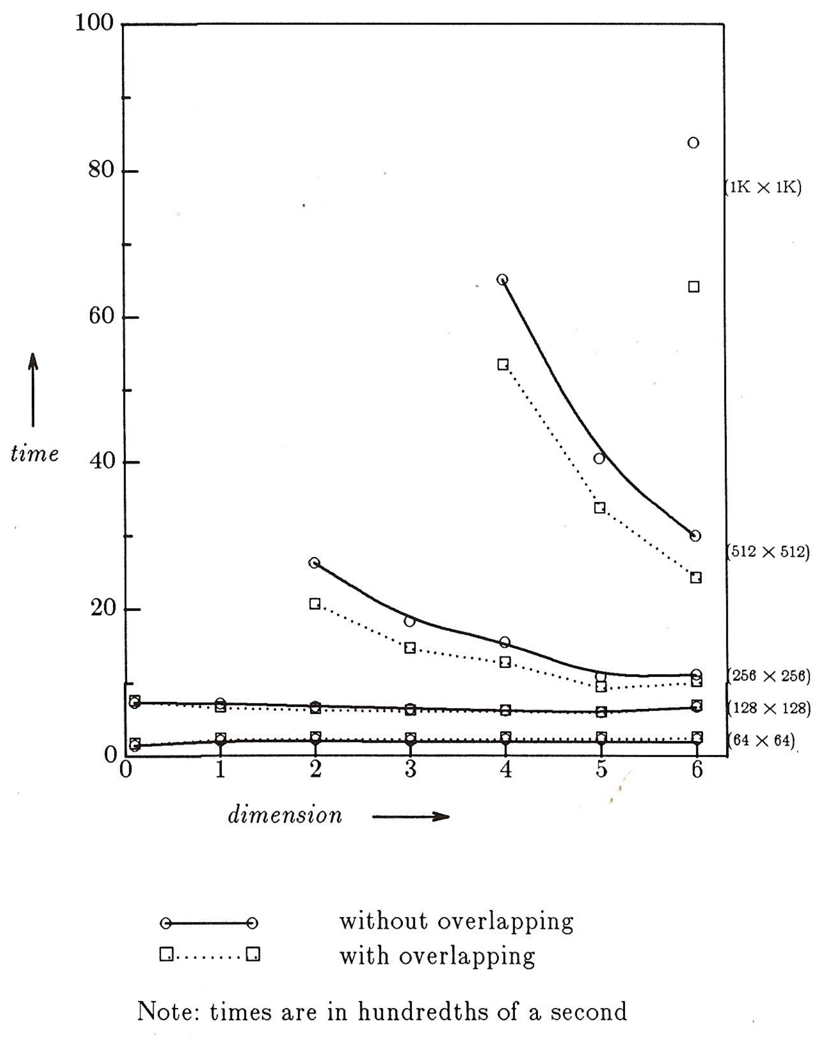


*Figure 19b.* Run time of *δ–synchronization* method.

data as it arrives. For smaller grids, there was no notable difference in performance. However, for larger grids, overlapping reduced run times significantly. The overlapping strategy reduced the run time of *global synchronization* by approximately 11% on a 1K × 1K grid and 10.5% on a 512 × 512 grid. For the *δ–synchronization* scheme, these reductions were approximately 26% and 18.5%, respectively. In the case of the *counter termination* method, the reductions were 22% and 15%, respectively.

Figure 20 plots the run times of all the methods on one graph for two grid sizes 256 × 256 and 512 × 512. The run times for the overlapping strategy are used except

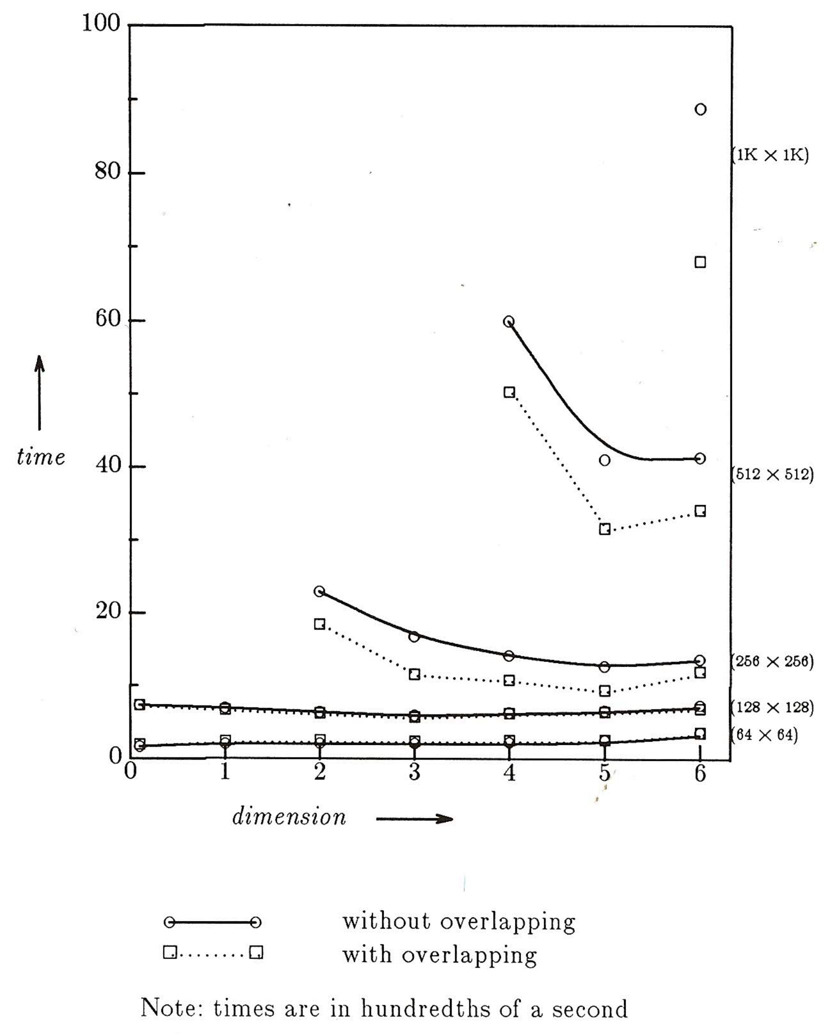


*Figure 19c.* Run time of *counter termination* method.

in the case of the *asynchronous* scheme where this strategy does not apply. As is evident, the multiprocessor algorithms are useful only on larger grids (and this is when we need them). Further, the $\delta$*-synchronization* method outperforms the remaining methods. This contrasts with the data of Figure 18 where the *asynchronous* scheme outperformed the others in the absence of the overlapping strategy.

Figure 21 shows how the communication and computation time changes as the hypercube dimension is changed. While the processing time drops as the cube dimension increases, the communication time remains steady (except when going from

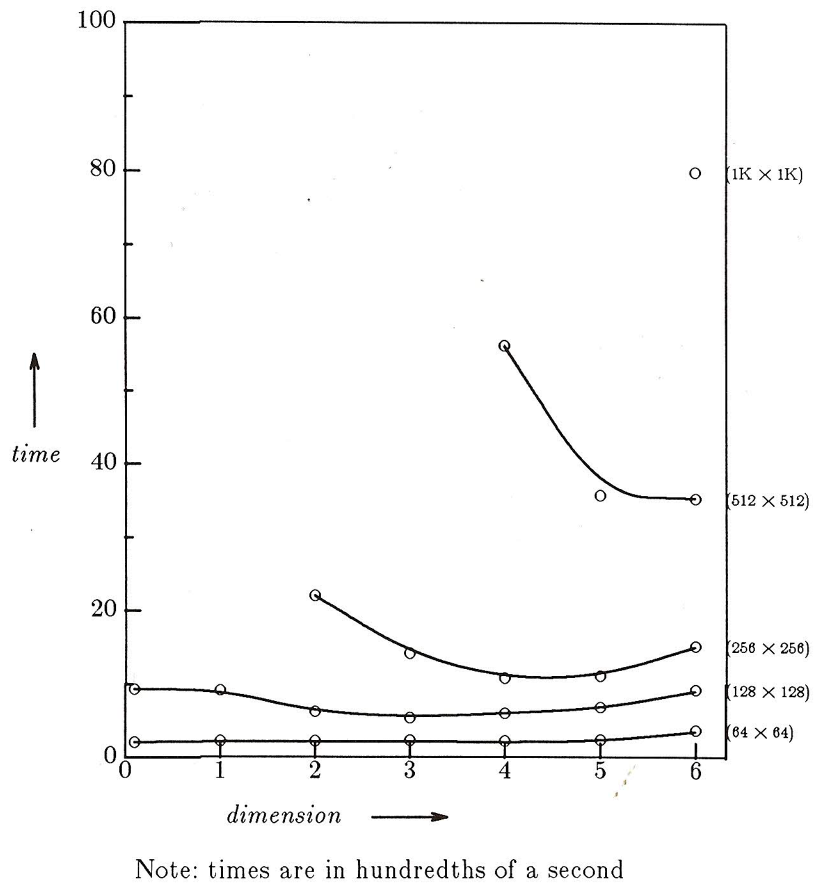


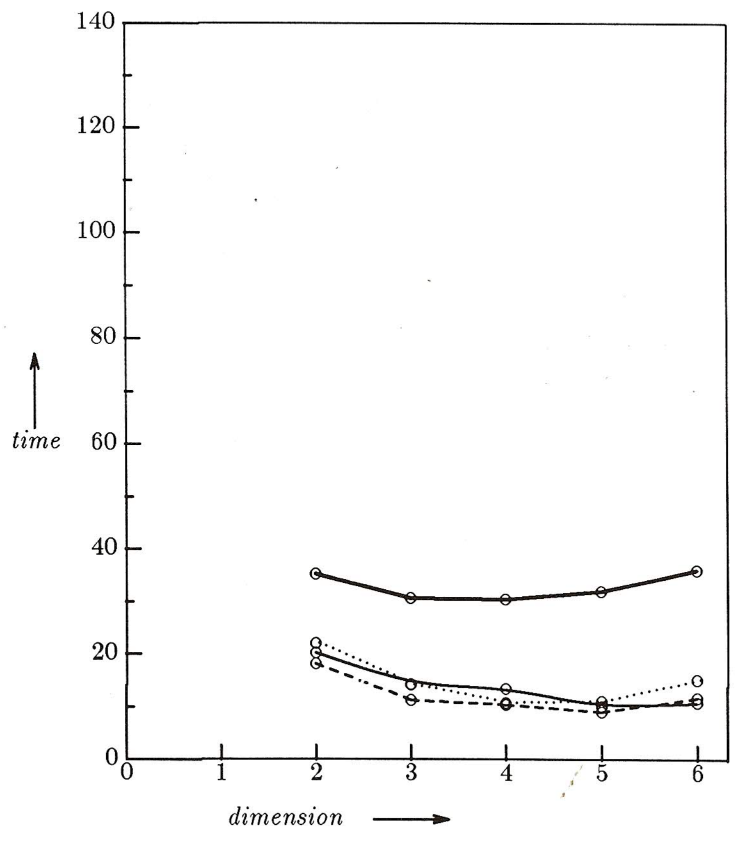
*Figure 19d.* Run time of *asynchronous* scheme.

dimension 0 to 1). This indicates that for any routing grid, increasing the hypercube dimension will reduce the run time only to a certain point. After this, the run time is limited by the communication overhead, and further reduction in run time cannot be obtained by increasing the number of processors.

Our experiments support the use of the *δ-synchronization* scheme when the hypercube dimension is large and when the grid size is large. The *asynchronous* scheme, even though it exploits the MIMD nature of the machine, is plagued by the overhead of maintaining the path length for each cell and the cost of termination detection.

## 9. Conclusions

We have implemented Lee's maze router on a hypercube multiprocessor computer.



(a) *256X256 grid*

| | |
|---|---|
| ——— | *δ-synchronization* method |
| - - - - - | *counter termination* |
| .......... | *asynchronous scheme* |
| ━━━ | *global synchronization method* |

Note: times are in hundredths of a second

*Figure 20a.* Run time of each method with overlapping.

Several different partitioning and control strategies were developed and investigated. *δ–synchronization* with overlapping gives the best performance on large grids and hypercubes. Further, one may expect substantial speedups on very large grids, though not on small ones. In our experiments a speedup of 1.6 was observed using *δ–synchronization* with overlapping on a 512 × 512 grid when the hypercube dimension was increased from 4 to 5. The speedup when the dimension increased from 4 to 6 was

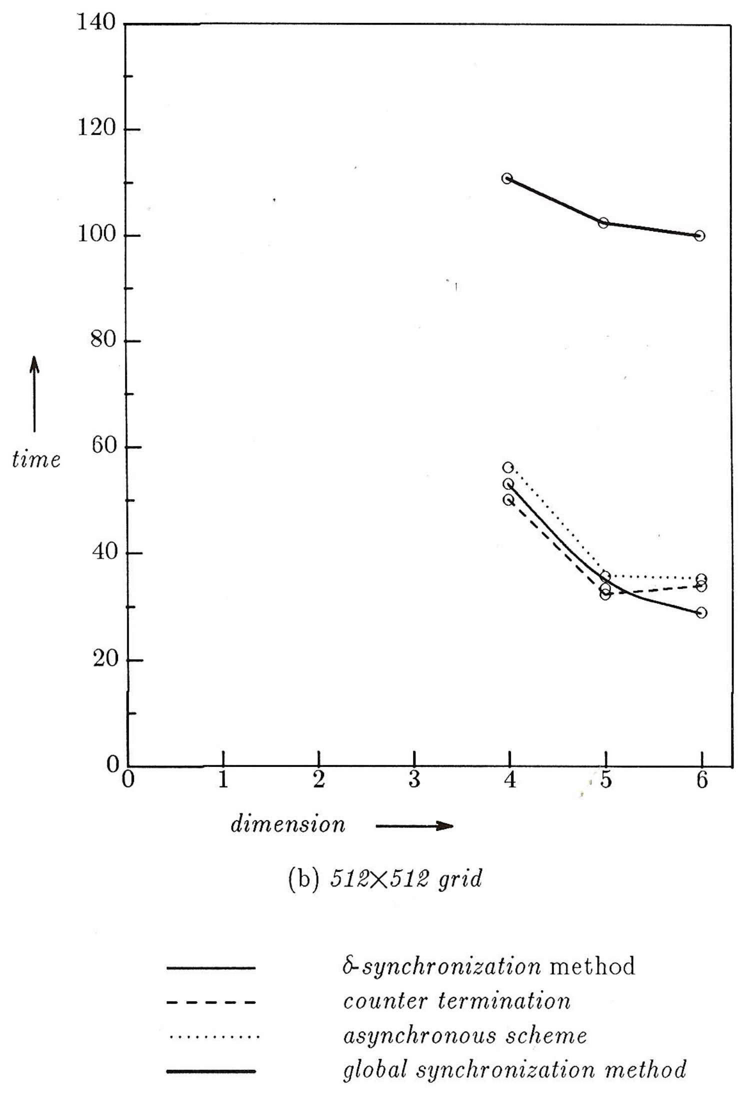


*Figure 20b.* Run time of each method with overlapping.

2.17. This work points out the significance of synchronization overhead and overlapping of computation and communication in MIMD parallel processing.

While our work considered only front wave expansion from the source to target cell, it is clear that our methods easily extend to the case of simultaneous expansion from both these cells.

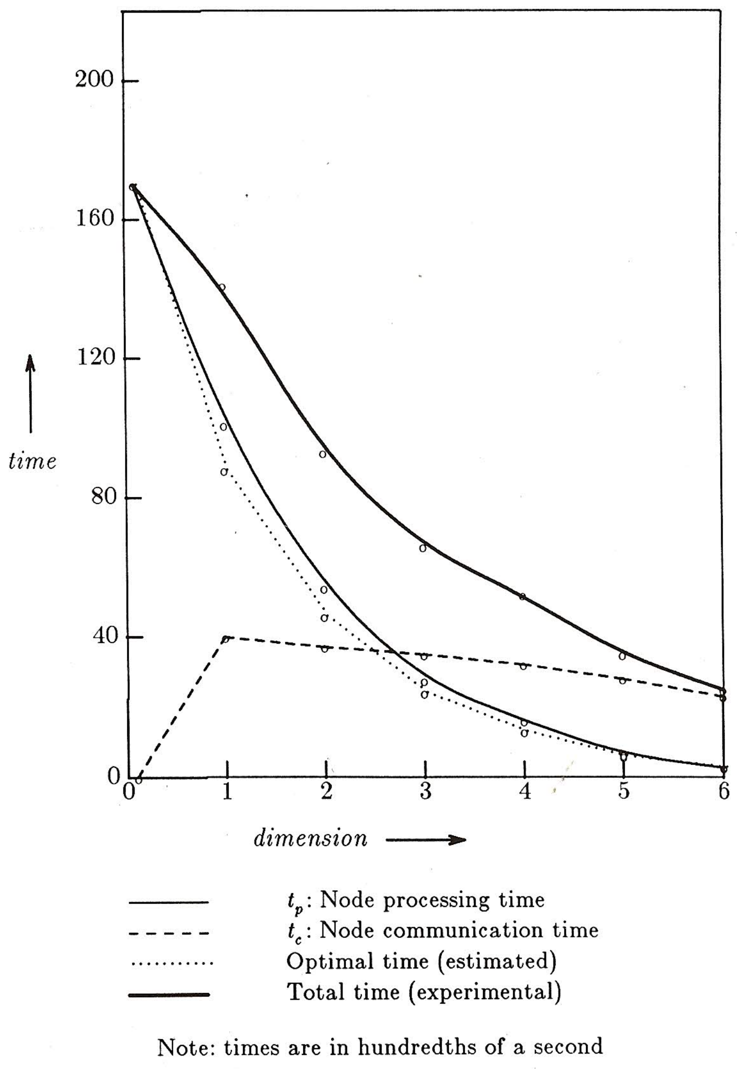


*Figure 21.* $t_p$ versus $t_c$ using *δ–synchronization* method with a random 512 × 512 grid for path length 266. (Performances for *dimension* 0, 1, 2, and 3 are estimated because of the memory limit.)

## Acknowledgments

This research was supported, in part, by the National Science Foundation under Grants MCS-83-05567 and DRC-84-20935.

## References

Blank, T., Stefik, M., and vanCleemput, W. 1981. A parallel bit map processor architecture for DA algorithms. In *Conference Proceedings—The 18th Design Automation Conference,* ACE/IEEE, pp. 837–845.

Dunigan, T.H. 1987. Hypercube performance. In *Conference Proceedings—Hypercube Multiprocessors 1987* (Knoxville, Sept. 29–Oct. 1, 1986), SIAM,pp. 178–192.

Iosupovici, A. 1986. A class of array architectures for hardware grid routers. *IEEE Trans. CAD, CAD-5, 2* (Apr.), 245–255.

Mudge, T.N., Ratenbar, R.A., Lougheed, R.M., and Atkins, D.E. 1982. Cellular image processing techniques for VLSI circuit layout validation and routing. In *Conference Proceedings—The 19th Design Automation Conference,* ACE/IEEE, pp. 537–543.

Nair, R., Hong, S.J., Liles, S., and Villani, R. 1982. Global wiring on a wire routing machine. In *Conference Proceedings—The 19th Design Automation Conference,* ACE/IEEE, pp. 224–231.

Palmer, J., Colley, S., Hayes, J.P., Mudge, T.N., and Stout, Q.F. 1986. Architecture of a hypercube supercomputer. In *Conference Proceedings—International Conference on Parallel Processing,* Pennsylvania State University, pp. 653–660.

Saad, Y., and Schultz, M.H. 1985. Topological properties of hypercubes. *Technical report YALE/DCS/RR-389,* Dept. of Computer Science, Yale University.

Suzuki, K., Matsunaga, Y., Tachibana, M., and Ohtsuki, T. 1986. A hardware maze router with application to interactive rip-up and reroute. *IEEE Trans. CAD, CAD-5, 4* (Oct.), 466–476.

Won, Y.J., Sahni, S., and El-Ziq, Y. 1987. A hardware accelerator for maze routing. In *Conference Proceedings—The 24th Design Automation Conference,* ACE/IEEE, pp. 800–807.