

# 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  $\rightarrow$ ,  $\leftarrow$ ,  $\downarrow$ , and  $\uparrow$  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

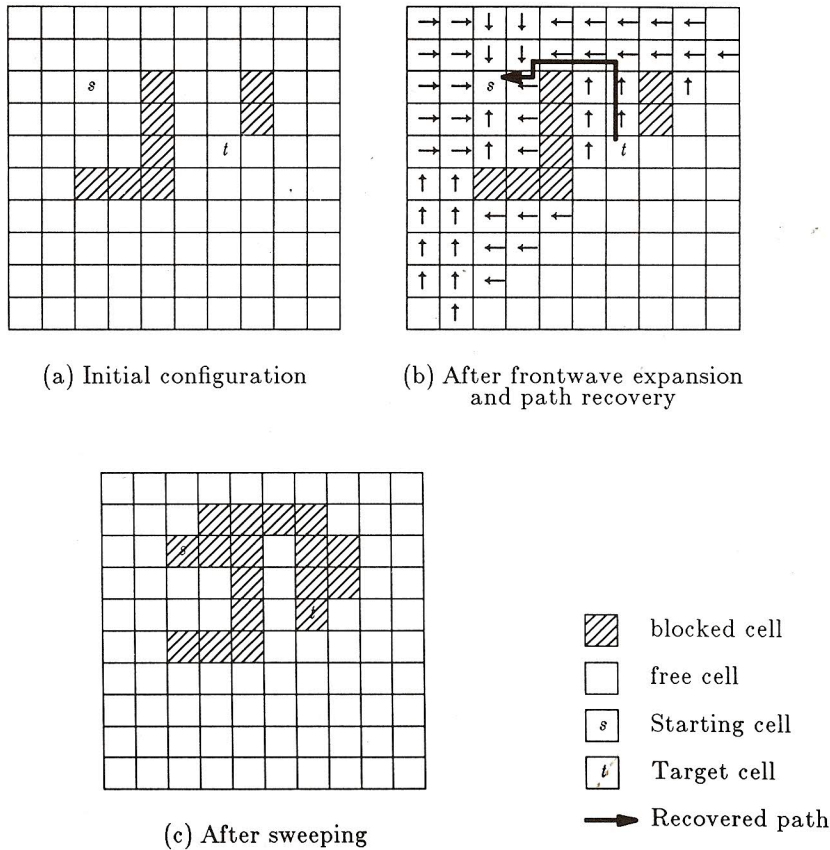


Figure 1. Maze routing phases.

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 array 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

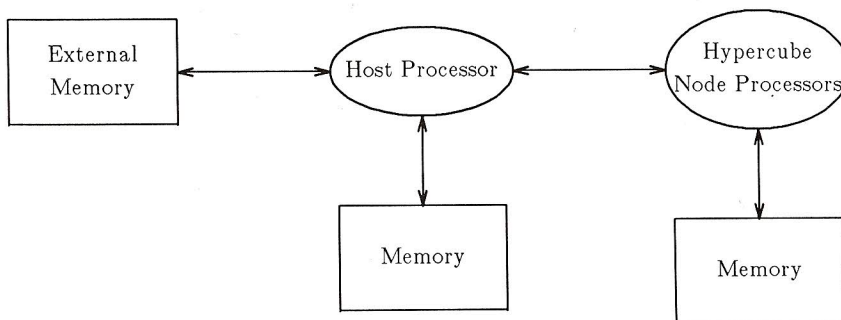


Figure 2. Hypercube multiprocessor.

