

PARALLEL MAZE ROUTING ALGORITHMS ON A
HYPERCUBE MULTICOMPUTER

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

TK
7874
.K87
1991

By
Tahsin Mertefe KURÇ
November 1991

PARALLEL MAZE ROUTING ALGORITHMS ON A
HYPERCUBE MULTICOMPUTER

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By

Tahsin Mertefe Kurç

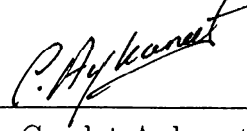
November 1991

Tahsin Mertefe Kurç
Bilkent University

B. 9562

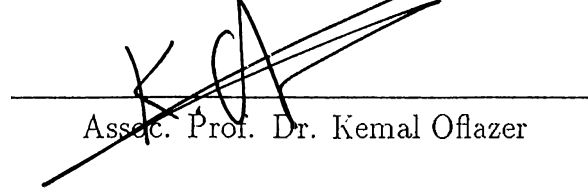
TK
7874
K87
1391

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



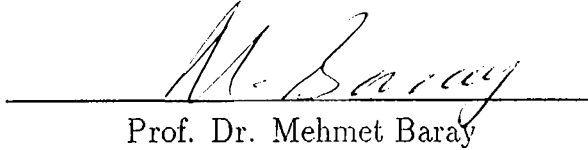
Assoc. Prof. Dr. Cevdet Aykanat (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



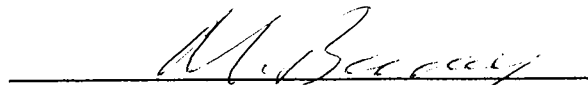
Assoc. Prof. Dr. Kemal Oflazer

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Prof. Dr. Mehmet Baray

Approved by the Institute of Engineering and Science:



Prof. Mehmet Baray, Director of the Institute of Engineering and Science

ABSTRACT

PARALLEL MAZE ROUTING ALGORITHMS ON A HYPERCUBE MULTICOMPUTER

Tahsin Mertefe Kurç

M. S. in Computer Engineering and Information Science

Supervisor: Assoc. Prof. Dr. Cevdet Aykanat

November, 1991

Global routing phase is a time consuming task in VLSI layout. In global routing phase of the layout problem, the overall objective is to realize all the net interconnections using shortest paths. Efficient heuristics are used for the global routing phase. However, due to the assumptions and constraints they impose, heuristics may fail to find a path for a net even if one exists. Re-routing is required for such nets. This re-routing phase requires the exhaustive search of the wiring area. Lee's maze routing algorithm and Lee type maze routing algorithms are exhaustive search algorithms used in re-routing phase.

These algorithms are computationally expensive algorithms and consume large amounts of computer time for large grid sizes. Hence, these algorithms are good candidates for parallelization. Also, these algorithms require large memory space to hold the wiring grid. Therefore, the effective parallelization of these algorithms require the partitioning of the computations and the grid among the processors. Hence, these algorithms can be parallelized on distributed-memory message passing multiprocessors (multicomputers).

In this work, efficient parallel Lee type maze routing algorithms are developed for hypercube-connected multicomputers. These algorithms are implemented on an Intel's iPSC/2 hypercube multicomputer.

Keywords: VLSI layout, maze routing, Lee's maze routing algorithm, Lee type maze routing algorithms, multicomputer, hypercube topology.

ÖZET

HİPERKÜP ÇOK İŞLEMCİLİ BİLGİSAYARINDA PARALEL LABİRENT YOL BELİRLEME ALGORİTMALARI

Tahsin Mertefe Kurç

Bilgisayar Mühendisliği ve Enformatik Bilimleri Bölümü

Yüksek Lisans

Tez Yöneticisi: Doçent Dr. Cevdet Aykanat

Kasım, 1991

Tümdevre tasarımında, devre bağlantılarının yapılması zaman alan bir iştir. Burada amaç bütün devre bağlantılarını en kısa yolları kullanarak yapmaktır. Eğer, her seferinde bir devre grubunun bağlantısı yapılırsa bunun adına *labirent yol belirleme yöntemi* denir.

Bu yöntem için, hüristik algoritmalar vardır. Ancak, bu tip algoritmalar devre bağlantılarına getirdikleri kısıtlamalardan dolayı, bazen var olan bağlantıları bulamazlar. Bu yüzden, devrelerin bulunduğu alanın, tümünden taranması gerekebilir. Lee'nin algoritması ve Lee benzeri algoritmalar bu tip algoritmalarlardır.

Lee'nin algoritması ve Lee benzeri algoritmalar hesaplama bakımından pahalı ve devre yüzeyi için çok bilgisayar hafızası gerektiren algoritmalarlardır. Bu nedenle bu tip algoritmalar çok işlemcili bilgisayarlarda, paralel olarak çözmek için uygundur.

Bu çalışmada, Lee benzeri labirent yol bulma algoritmalarının, hiperküp çok işlemcili bilgisayarında paralelleştirilmesi anlatılmaktadır.

Anahtar kelimeler : Tmdevre tasarımı, labirent yol bulma yntemi, Lee'nin labirent yol bulma algoritması, Lee benzeri labirent algoritmaları, ok ilemcili bilgisayar, hiperkp topolojisi.

ACKNOWLEDGEMENT

I wish to thank very much my supervisor Assoc. Prof. Dr. Cevdet Aykanat, who has guided and encouraged me during the development of this thesis.

I am grateful to Professor Mehmet Baray and Assoc. Prof. Dr. Kemal Oflazer for their remarks and comments on the thesis.

It is pleasure to express my thanks to all my friends for their valuable discussions and to my family for providing morale support during this study.

Contents

1	INTRODUCTION	1
2	SEQUENTIAL MAZE ROUTING ALGORITHMS	10
2.1	Lee's Maze Routing Algorithm	10
2.2	Lee Type Algorithms For Routing of Multipin Nets	13
2.2.1	Using Prim's Algorithm	17
2.2.2	Using Kruskal's Algorithm	18
3	PARALLELIZATION OF LEE'S ALGORITHM	23
3.1	Grid Partitioning and Mapping	24
3.2	Parallel Front Wave Expansion	29
3.2.1	Expansion Starting From Source Only	29
3.2.2	Expansion Starting From Source and Target	35
3.3	Termination Detection	42
3.3.1	Global Synchronization	42
3.3.2	Counter Termination Scheme1	43
3.3.3	Counter Termination Scheme2	43

3.4	Overlapping Communication with Computation	46
3.5	Asynchronous Scheme	49
3.5.1	Expansion Starting From Source Only	49
3.5.2	Expansion Starting From Source and Target	51
3.6	Parallel Path Recovery and Sweeping	55
3.6.1	Non-pipelined scheme	57
3.6.2	Pipelined Scheme	57
3.7	Experimental Results	63
4	PARALLEL ALGORITHMS FOR MULTIPIN NETS	72
4.1	Parallel Akers' Algorithm for Multipin Nets	72
4.2	Parallel Kruskal's Steiner Tree Algorithm	74
4.3	Experimental Results	77
5	CONCLUSIONS	84

List of Figures

1.1	Grid representation of the wiring surface in gate array layout.	3
1.2	(a) Single bend path (heuristic can find such a path) (b) Two-bend path (heuristic fails to find such a path).	4
1.3	Routing of nets using heuristic (a) Routing of net (S_1, T_1) (b) Routing of net (S_2, T_2)	5
1.4	Routing of nets using heuristic (a) Routing of net (S_3, T_3) (b) Routing of net (S_4, T_4)	6
1.5	Heuristic fails to find the path for (S_5, T_5) , because the path (dotted lines) violates cell capacities.	7
1.6	8 node hypercube structure	8
2.1	A sample global grid for Lee's maze routing algorithm	11
2.2	Front wave expansion phase of the Lee's maze routing algorithm	11
2.3	Path recovery phase of the Lee's maze routing algorithm	12
2.4	Sweeping phase of the Lee's maze routing algorithm	13
2.5	Front wave expansion phase of the Lee's algorithm (a) Initial cycles of front wave expansion phase (b) Successful termination of front wave expansion	14

2.6	Path recovery and sweep phases of Lee's algorithm after the front wave expansion phase (a) Path Recovery phase (b) Final configuration after sweep phase	15
2.7	The use of the sweep queue, cells marked as 1,2,3 are added into the sweep queue at expansion cycles 3,3, and 2, respectively. . .	16
2.8	Sequential version of the algorithm for routing multipin nets using Prim's algorithm	18
2.9	Front wave expansion phase of the Akers' algorithm beginning from the terminal cell 'a' (a) initial two cycles (b) Initial cycle after the first path (a to b) is connected.	19
2.10	The final configuration after all three pins are connected	20
2.11	Sequential version of the algorithm for routing multipin nets using Kruskal's algorithm	21
2.12	Routing of a three pin net using Kruskal's algorithm (a) Initial cycles of the algorithm (b) After connecting all pins	22
3.1	Mesh embedding (a) hypercube of dimension 2 (b) hypercube of dimension 3 (c) hypercube of dimension 4	25
3.2	Tiled decomposition of a 16x16 grid onto 2x4 mesh embedded 3-dimensional hypercube	26
3.3	Scattered decomposition of a 16x16 grid	28
3.4	Decomposition of 16x16 grid into subblocks of (a) $h = w = 2$ (b) $h = 2w = 4$	30
3.5	Local data structures for a node processor.	31
3.6	Node program for the Sonly scheme	32
3.7	Encoding the status information	33

3.8	Mapping of local coordinates onto the repeating mesh template	36
3.9	Expansion starting from source and target scheme (a) Initial cycles (b) Collision of two front waves	38
3.10	Node program for the S+T scheme.	39
3.11	Host and node programs for the counter termination scheme 1.	44
3.12	Host and node programs for the counter termination scheme 2.	45
3.13	Overlapping communication and computation	47
3.14	Host and node programs for the asynchronous Sonly scheme.	50
3.15	Failure to find the shortest path. If processors P_k , P_m and P_l are faster than processor P_i , target T can be reached by a longer path.	52
3.16	Labeling of an already labeled cell by a shorter path, the shaded cells have been reached by a shorter path hence the cells expanding from cell c will relabel the shaded cells	53
3.17	Host and node programs for the asynchronous S+T scheme.	54
3.18	(a) Cell 1 is added into the sweep queue without any extra communication (b) Cell 1 is added into the sweep queue in parallel algorithm while it is not in sequential algorithm	58
3.19	Host program for the non-pipelined path recovery and sweep phase	59
3.20	Node program for non-pipelined path recovery and sweeping scheme.	59
3.21	The calculation of r value.	60
3.22	Node program for the pipelined path recovery and sweep phase.	62
3.23	Effect of w values on the performance of the parallel algorithm for $N = 1024$, $P = 4$. (a) $h = w$ (b) $h = 2w$	65

3.24	Speed-up for various parallel algorithms for front wave expansion phase	66
3.25	Speed-up vs grid size	66
3.26	Efficiency vs grid size	67
3.27	Speed-up figures for asynchronous algorithms	67
3.28	Effect of w values on the performance of path recovery (a) $h = w$ (b) $h = 2w$	69
3.29	Effect of w values on the performance of path recovery + sweep (a) $h = w$ (b) $h = 2w$	70
3.30	Speed-up for parallel algorithms for sweep + path recovery phase	71
4.1	Node program for Akers' algorithm	73
4.2	Host program for parallel algorithm for multipin nets using Kruskal's algorithm	75
4.3	Node program for the parallel algorithm for multipin nets using Kruskal's algorithm	76
4.4	Effect of h,w values on the execution of parallel Akers' algorithm (a) $h = w$ (b) $h = 2w$	79
4.5	Effect of h,w values on the execution of parallel Kruskal's Steiner tree algorithm (a) $h = w$ (b) $h = 2w$	80
4.6	Speed-up figure for parallel Akers' algorithm on a 512x512 grid for 4, 7, 10 pin nets	81
4.7	Speed-up figure for parallel Akers' algorithm on a 1024x1024 grid for 4, 7, 10 pin nets	81
4.8	Speed-up figure for parallel Kruskal's Steiner tree algorithm on a 512x512 grid for 4, 7, 10 pin nets	82

4.9	Speed-up figure for parallel Kruskal's Steiner tree algorithm on a 1024x1024 grid for 4, 7, 10 pin nets	83
-----	--	----

1. INTRODUCTION

With recent advances in VLSI technology, it is now feasible to manufacture integrated circuits with several hundred thousand, even millions of transistors. This manufacturing capability together with the economic and performance benefits of large scale VLSI systems necessitates the automation of the circuit design process. The circuit design process, the layout of integrated circuits on chips, is a complex task. The major research issue in the design automation is the development of efficient and easy-to-use systems for circuit layout.

In the combinatorial sense, the layout problem is a constrained optimization problem. A layout problem instance is given by a description of the circuit by a netlist. A netlist describes the switching elements and their connecting wires. The question is to find an assignment of the geometric coordinates of the circuit components in the planar layer(s) that minimizes certain cost criteria while maintaining the fabrication technology constraints. Most of the optimization problems encountered during the integrated circuit layout are intractable, that is they are NP-hard [1]. Hence, heuristic methods are used to find solutions in reasonable time. Usually, the layout problem is decomposed into subproblems which are then solved one after another. These subproblems are usually NP-hard as well, but they are more suitable for heuristic solutions than the whole layout problem. A typical layout problem decomposition is *component placement* followed by the *global routing*. In the global routing phase, the approximate course of wires are determined. The global routing phase is followed by *detailed routing* phase to determine the exact course of wires.

There are two major kinds of layout methodologies, full-custom layout and

semi-custom layout [1]. In full-custom layout, the design starts on an empty piece of silicon. The designer has a wide range of freedom in component placement and routing. In semi-custom layout, this freedom is severely restricted. The design starts on a prefabricated silicon that already contains all switching elements (e.g. gate arrays) [1, 2] or involves the use of basic circuit components from geometrically restricted libraries (e.g. standard cells) (Chap. 1, pp. 18-26 in [1]). Semi-custom layout is more suitable for design automation.

In *gate array* layout, initially the design area is not empty. There are prefabricated switching elements (cells), such as boolean gates or flip-flops, on the wafer. In the gate array layout, the placement problem is actually an assignment problem. Each gate in the netlist of the given circuit is assigned a cell on the wafer that will implement this gate. These cells, implementing the gates, are then interconnected using only top metal layer(s) so that the given netlist description is realized. The fabrication in gate arrays is simpler since the last few steps of the fabrication process have to be custom-tailored. Furthermore gate arrays are less expensive since the number of masks to describe the given circuit is reduced considerably. The placement and detailed routing phases of layout problem are out of the scope of this work. More information on placement and detailed routing can be found in [1].

In global routing phase of the layout problem, the routing area can be represented as a 2-dimensional grid as shown in Fig. 1.1, when one metal layer is used for wiring. The grid (also called global grid) is divided into squares called *cells*. There are specially designated cells called *pins*, such as cells S_1 , T_1 , S_2 , T_2 in Fig. 1.1. A *net* is defined to be the set of pins to be interconnected. For example, the net N_1 is denoted by two pins S_1 and T_1 . In Fig. 1.1, all nets have two pins, hence they are called *two-pin nets*. However, in practice some of the nets may have more than two pins, such nets are called *multipin nets*. The overall aim in global routing is to realize all the net interconnections using the shortest paths. Here, a *path* is defined to be the interconnecting wire between the pins of a net. The paths are realized by passing wires through the channels in the cells. The vertical and horizontal lines between neighbor cells represent the channels. Wires can go from one cell to another adjacent cell by either crossing a vertical or a horizontal channel. Hence, moves from

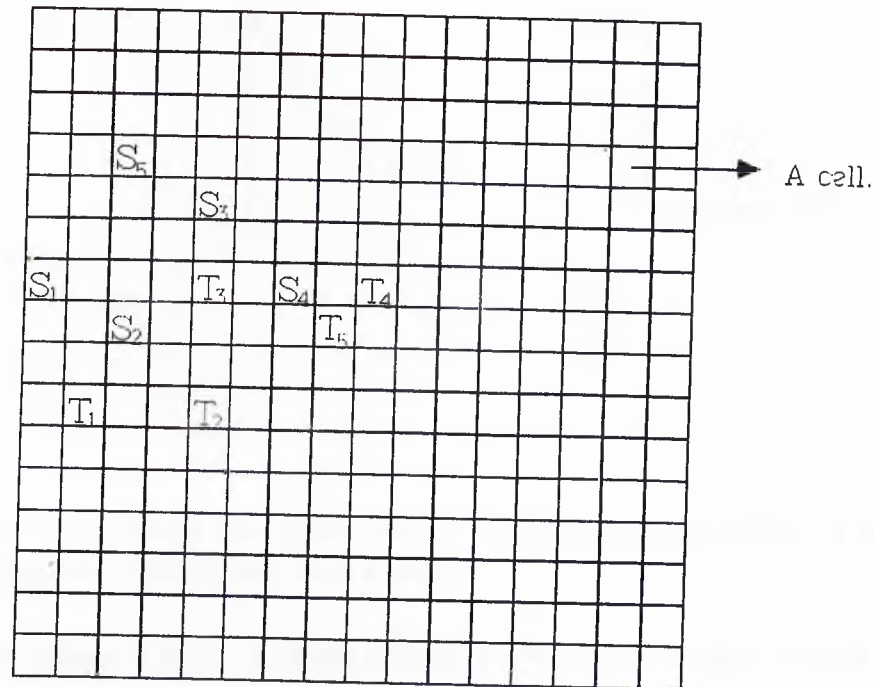


Figure 1.1. Grid representation of the wiring surface in gate array layout.

a cell are restricted to four directions (south,north,west,east). However, due to the technological constraints, the channels are assigned with a channel capacity representing the number of wires that can cross that channel. As the interconnections between the pins (net terminals) are constructed some of the cells are declared to be blocked, that is no more wires can pass through those cells. In this work, for simplicity, each cell is assumed to have a wiring capacity of a single wire. If nets are interconnected (routed) one net at a time basis, the global routing phase reduces to *maze routing*.

Since there may be thousands of nets to be routed, global routing is a time consuming task. Hence, heuristics are used for global routing and maze routing [3, 4, 5, 6, 10]. However, due to the assumptions and constraints they impose, heuristic algorithms may fail to find a path even if one exists. This can be illustrated by Figs. 1.3 and 1.4. In these figures, the routing of nets are done by using a heuristic which allows routing of nets having at most a single

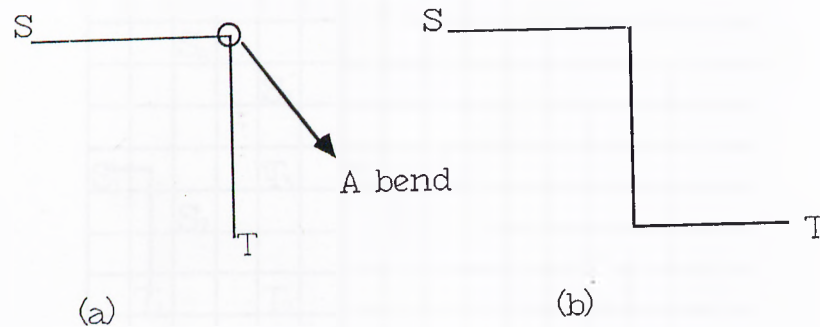
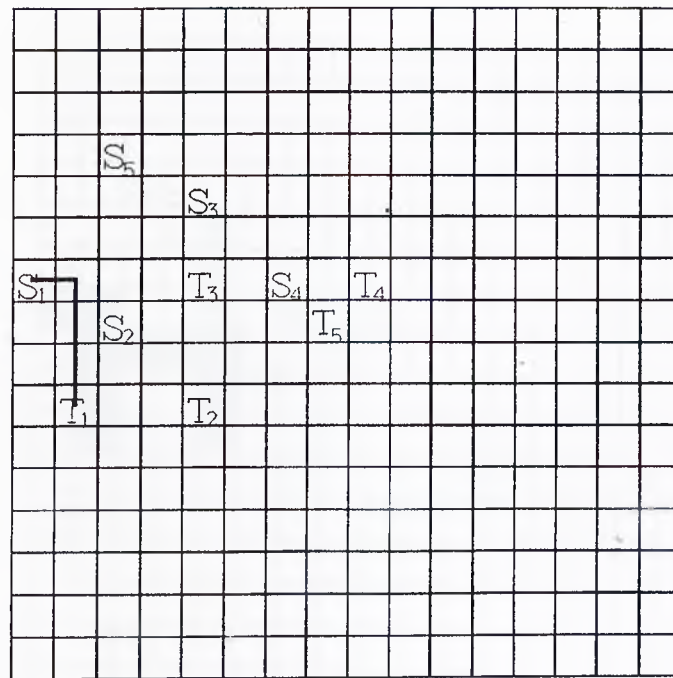


Figure 1.2. (a) Single bend path (heuristic can find such a path) (b) Two-bend path (heuristic fails to find such a path).

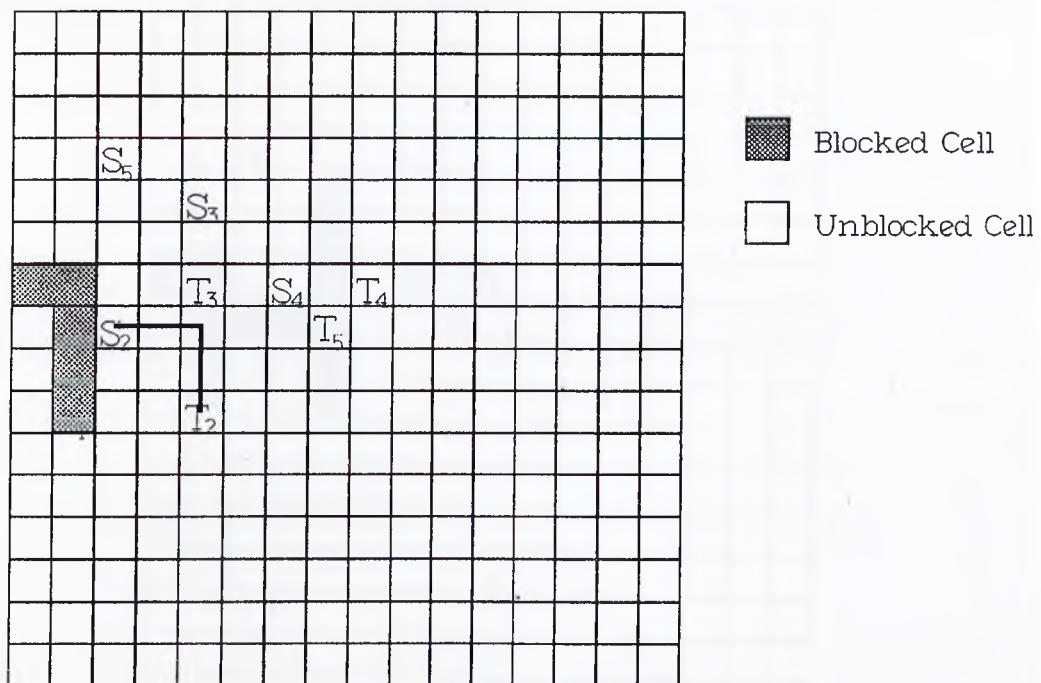
bend as shown in Fig. 1.2. After routing a net, the cell capacities are updated and cells on the path are declared as blocked. The routing of net (S_5, T_5) , however, can not be done by this heuristic, because heuristic can only find a single bend path (dotted lines in Fig. 1.5), which violates the cell capacities of some cells. The re-routing of this net is required. The re-routing of this net can be achieved by exhaustive search of the wiring area. Lee's maze routing algorithm and Lee type algorithms for multipin nets are such type of exhaustive search algorithms.

These algorithms are computationally expensive algorithms and consume large amounts of computer time for large grid sizes. Hence, these algorithms are good candidates for parallelization. Also, these algorithms require large memory space to hold the wiring grid. Therefore, the effective parallelization of these algorithms require the partitioning of the computations and the grid among the processors. Hence, these algorithms can be parallelized on distributed-memory message passing multiprocessors (multicomputers).

A multicomputer is an ensemble of processors interconnected in a certain topology. In a multicomputer, each processor has its own local memory and there is no globally shared memory in the system. Each processor runs independently (asynchronously). The cooperation, synchronization and data exchange

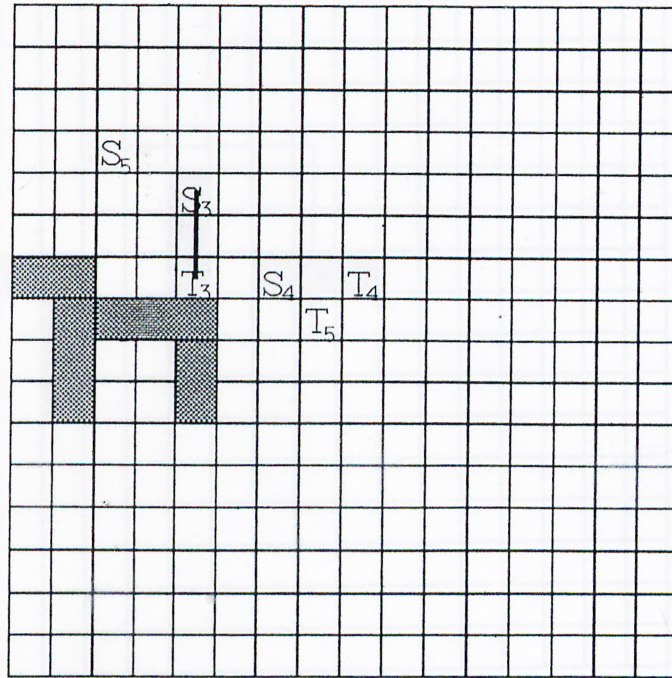


(a)

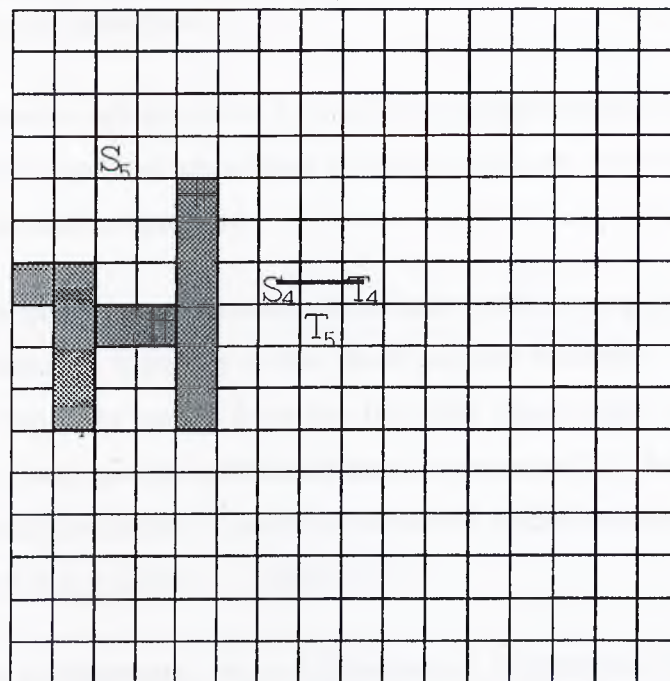


(b)

Figure 1.3. Routing of nets using heuristic (a) Routing of net (S_1, T_1) (b) Routing of net (S_2, T_2) .



(a)



(b)

Figure 1.4. Routing of nets using heuristic (a) Routing of net (S_3, T_3) (b) Routing of net (S_4, T_4) .

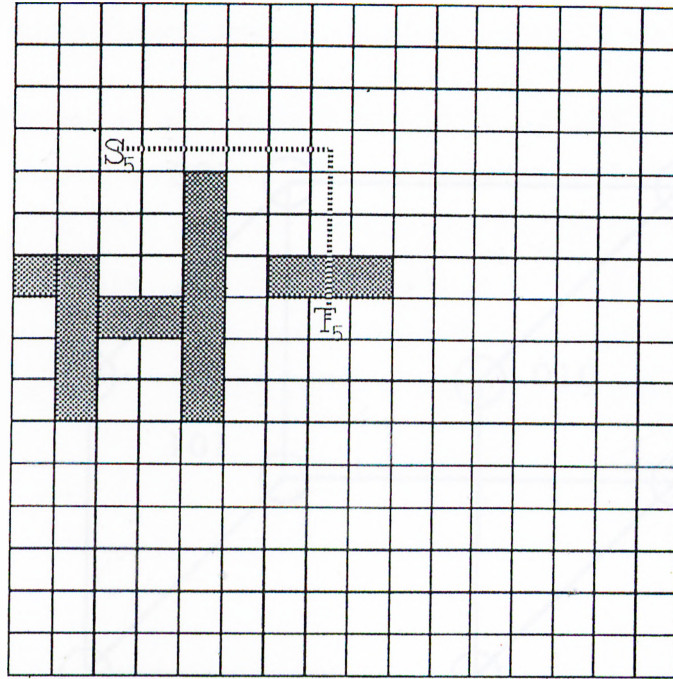


Figure 1.5. Heuristic fails to find the path for (S_5, T_5) , because the path (dotted lines) violates cell capacities.

between processors are achieved by explicit message-passing between processors. Therefore, the interconnection topology plays an important role on the performance of such computers.

Among the many interconnection topologies such as ring, mesh etc., hypercube interconnection topology is the most popular topology. The popularity of hypercube topology comes from the fact that many other topologies (such as ring, mesh, tree) can be embedded onto hypercube [11]. In addition, there are commercially available hypercube connected multicomputers such as FPS T-series ¹, NCUBE ², iPSC/1 ³, iPSC/2 ⁴.

In such an architecture, for a d dimensional hypercube, there are 2^d processors (nodes). Each node is connected directly to d other nodes. Figure 1.6 represents a 3-dimensional hypercube structure. The binary encoding of a processor differs in only one bit from the neighbor's encoding. The processors

¹FPS T-series is a registered trade mark of FPS Inc.

²NCUBE is a registered trade mark of NCUBE Inc.

³iPSC/1 is a registered trade mark of Intel Inc.

⁴iPSC/2 is a registered trade mark of Intel Inc.

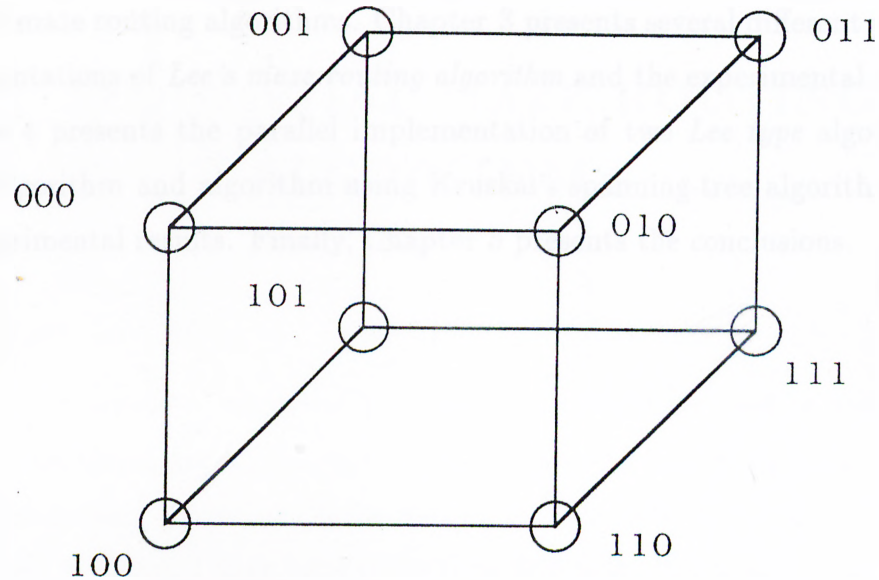


Figure 1.6. 8 node hypercube structure

can directly communicate to d neighbors only. The communication between processors that are not connected directly is done through other processors by either software or hardware. Maximum distance between two processors in a hypercube is d .

Achieving speed-up through parallelism in such architectures is not straightforward. The algorithm must be designed so that data and computations are distributed evenly among processors to achieve the maximum load balance. In a parallel machine with high communication latency, the algorithms must be designed so that the large amounts of computations are done between communication steps. Another factor effecting the parallel algorithms is the ability of parallel systems to overlap the computation with communication. A good parallel algorithm should exploit these factors and the topology of the architecture to achieve maximum speed-up.

In this work, the parallel implementation of *Lee's maze routing algorithm*

and *Lee type multipin net algorithms* on a commercially available multicomputer implementing the hypercube connection topology is addressed.

The organization of this thesis is as follows, Chapter 2 presents the sequential maze routing algorithms. Chapter 3 presents several different parallel implementations of *Lee's maze routing algorithm* and the experimental results. Chapter 4 presents the parallel implementation of two *Lee type* algorithms, Akers' algorithm and algorithm using Kruskal's spanning-tree algorithm, and the experimental results. Finally, Chapter 5 presents the conclusions.

2. SEQUENTIAL MAZE ROUTING ALGORITHMS

This chapter presents sequential maze routing algorithms used in exhaustive search of the wiring area. First section describes a well known algorithm, called Lee's maze routing algorithm [7], for routing two-pin nets. Some nets, however, as is stated in Chap. 1 may have more than two pins. Routing of such nets is the direct translation of *Steiner Tree problem*[15, 16] into the context of routing in rectangular grids. There are two algorithms that are the variations of Lee's maze routing algorithm. These algorithms are presented in section 2.

2.1 Lee's Maze Routing Algorithm

Lee's maze routing algorithm is a well known algorithm for routing two-pin nets. In the two-pin net problem, the routing area is represented as a two-dimensional grid as shown in Fig. 2.1. Each cell has a status which may be blocked or unblocked initially. This status information is kept in a two-dimensional array called *status array*. There are two special cells, called source (S) and target (T) (see Fig. 2.1). The aim is to find the shortest path between source and target cells.

Lee's maze routing algorithm consists of three phases, namely, *front wave expansion*, *path recovery*, and *sweeping* [8]. *Front wave expansion* phase is a *breadth-first* search strategy starting from the source cell. The description of the algorithm for the *front wave expansion* phase is given in Fig. 2.2.

The labeling operation (at Step 2) of a *free* and *unlabeled* adjacent cell

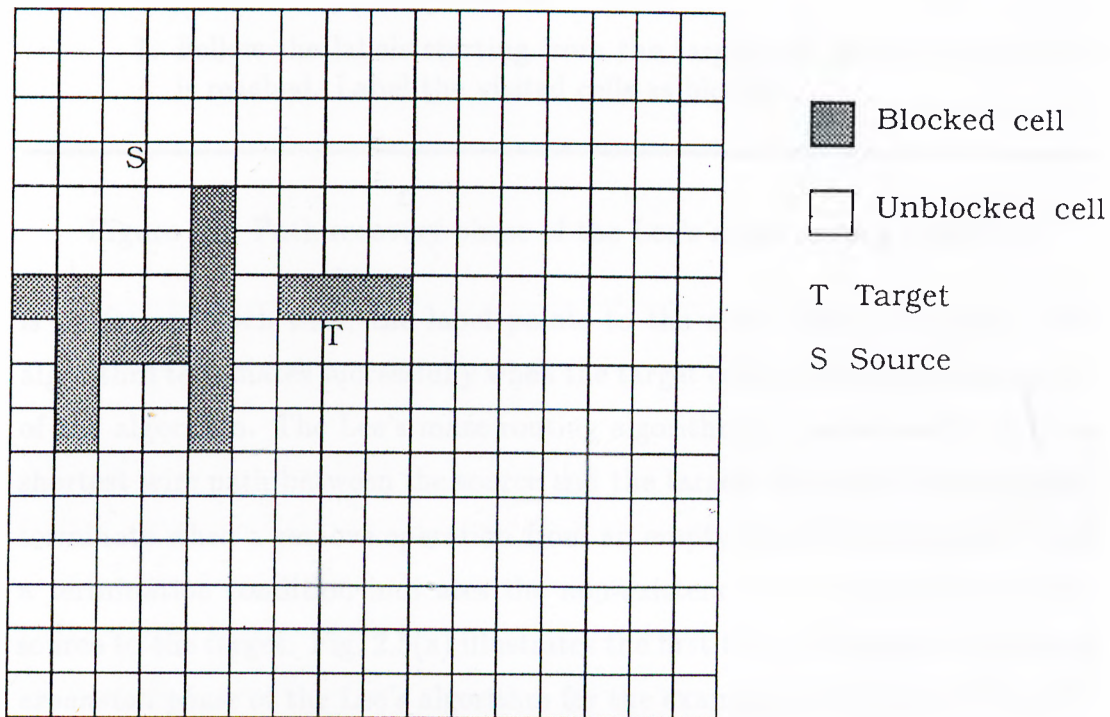


Figure 2.1. A sample global grid for Lee's maze routing algorithm

A queue, called expansion queue, initially contains only the source cell. A queue, called sweep queue, is initially empty. A two dimensional $N \times N$ *Status* array holds the status for the cells of an $N \times N$ grid. All the *free* cells are initially *unlabeled*.

1. Remove a cell c from the expansion queue.
 2. Examine the four adjacent cells of the cell c using the current information in the *Status* array. Discard the *blocked* and already *labeled* adjacent cells. Update the status of the *unlabeled free* adjacent cells as *labeled* in the *Status* array and add those cells to the expansion queue. If all adjacent cells of the cell c are either blocked or already labeled, then add the cell c into the sweep queue.
 3. Go to step 1 until either target cell is labeled or expansion queue becomes empty.
-

Figure 2.2. Front wave expansion phase of the Lee's maze routing algorithm

-
1. Follow the labels starting from the target cell until the source cell is reached. Label the visited cells as blocked.
-

Figure 2.3. Path recovery phase of the Lee's maze routing algorithm

is performed such that, the label points to the cell c being expanded. The algorithm terminates successfully when the target cell t is labeled during Step 2 of the algorithm. The Lee's maze routing algorithm is guaranteed to find the shortest wire path between the source and the target. The algorithm may also terminate when a remove operation from an empty queue is attempted. Such a termination condition indicates the non-existence of a wire-path from the source to the target. Fig. 2.5(a) illustrates the first two cycles of the *front wave expansion phase* of the Lee's algorithm for the example grid shown in Fig. 2.1. Labeling process at Step 2 of the algorithm is illustrated by the following four labels, \downarrow , \leftarrow , \uparrow , and \rightarrow in the figure. The *front wave expansion* phase is followed by the *path recovery* and *sweeping* phases. Fig. 2.5(b) illustrates the successful termination of the *front wave expansion* phase.

In the path recovery phase, the labels are followed starting from the target cell to construct the path between source and target (see Fig. 2.6(a)). The algorithm is given in Fig. 2.3.

After the path recovery phase is completed, the labeled cells in the front wave expansion phase have to be unlabeled so that next net can be routed. This unlabeled operation is carried out in sweeping phase. The sweeping phase is given in Fig. 2.4.

At the end of front wave expansion phase the expansion queue contains the cells expanded in the last expansion cycle of the front wave expansion phase. These terminal cells are already labeled and connected to their parents. Thus, the cells labeled in the expansion paths starting from the source cell and terminating at these terminal cells can be unlabeled by following their labels (step 2 of the sweep algorithm). However, during the front wave expansion phase, some of the expansion paths initiated from the source cell are blocked

-
1. Remove a cell c from *sweep queue* or *expansion queue*
 2. Follow the labels starting from c until a blocked or unlabeled cell is reached. Unlabel the visited cells in the status array.
 3. Repeat steps 1 and 2 until both *sweep queue* and *expansion queues* are empty.
-

Figure 2.4. Sweeping phase of the Lee's maze routing algorithm

either due to blocked cells or due to the already labeled cells. The terminal cells of these blocked expansion paths are added into the sweep queue at step 2 of the front wave expansion algorithm (see Fig. 2.7). Hence, the cells labeled in these blocked expansion paths should also be unlabeled during the sweeping phase. Fig. 2.6(b) illustrates the final configuration after the *path recovery* and *sweeping* phases.

2.2 Lee Type Algorithms For Routing of Multipin Nets

The routing of multipin nets is the direct translation of *Minimum Steiner tree* problem [15] into the context of routing. The definition of the *Minimum Steiner tree* problem (or Steiner tree problem) for general graphs [1] is as follows :

Instance : A connected undirected graph $G = (V^1, E^2)$ with edge cost function $\lambda : E \rightarrow R_+$ and a subset $R \subset V$ of *required* vertices.

configurations : All edge-weighted trees.

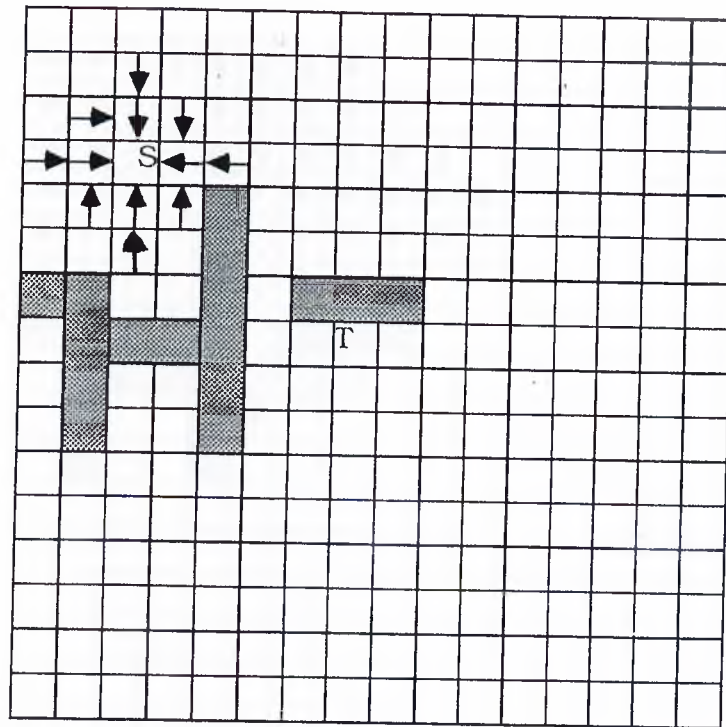
Solutions : All *Steiner trees* for R in G ; that is, all subtrees of G that connect all vertices in R and all of whose leaves are vertices in R .

minimize : $\lambda(T) = \sum_{e \in E_T} \lambda(e)$

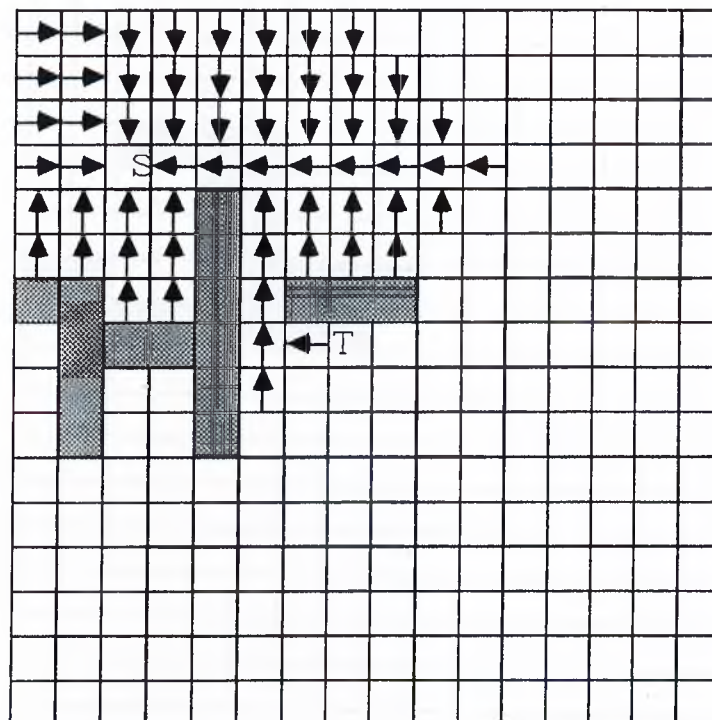
The Steiner tree problem is an NP-hard problem. The existing approximate algorithms try to find an suboptimal solution in reasonable time. These

¹ V represents the vertices of the graph G

² E represents the edges of graph G

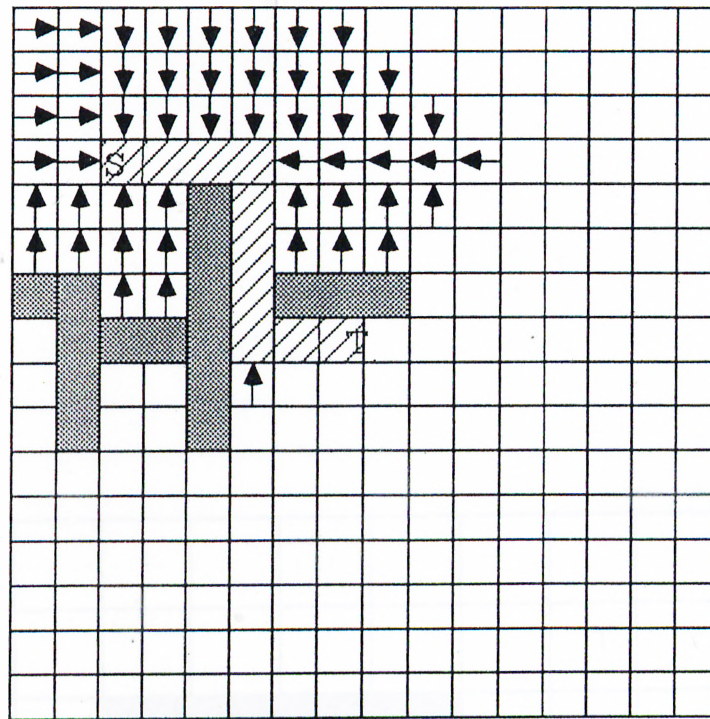


(a)

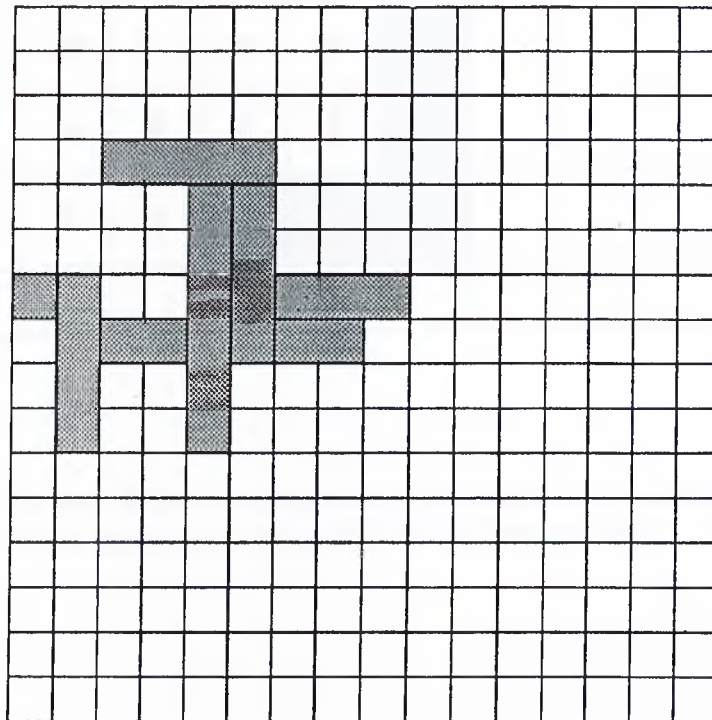


(b)

Figure 2.5. Front wave expansion phase of the Lee's algorithm (a) Initial cycles of front wave expansion phase (b) Successful termination of front wave expansion



(a)



(b)

Figure 2.6. Path recovery and sweep phases of Lee's algorithm after the front wave expansion phase (a) Path Recovery phase (b) Final configuration after sweep phase

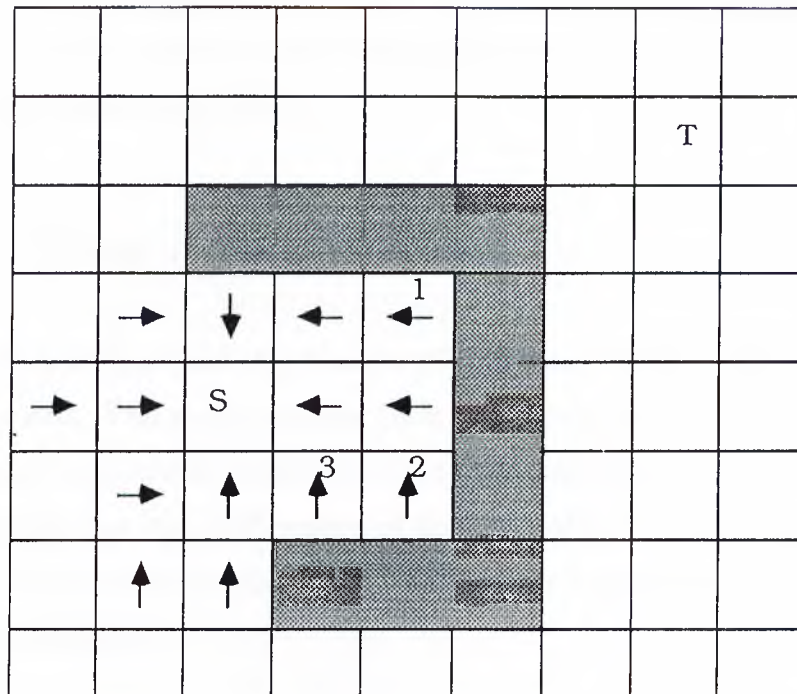


Figure 2.7. The use of the sweep queue, cells marked as 1,2,3 are added into the sweep queue at expansion cycles 3,3, and 2, respectively.

approximate algorithms try to find a good Steiner tree by combining minimum-spanning-tree and shortest path calculations. The cost of the Steiner tree T_{sub} , on grid graphs, found by these algorithms is bounded by

$$Cost(T_{Sub}) \leq \frac{3}{2}Cost(T_{Opt}) \quad (2.1)$$

as shown in [16], where $Cost(T_{Sub})$ is the cost of the suboptimal tree and $Cost(T_{Opt})$ is the cost of optimal Steiner tree. In this work, parallelization of two algorithms that use Prim's and Kruskal's algorithms [1] for minimum-spanning-tree calculations and Lee's maze routing algorithm for the shortest path calculations is addressed. Following two sections present the sequential versions of these algorithms.

2.2.1 Using Prim's Algorithm

Using Prim's algorithm [1], Akers[1, 17] has developed an algorithm to route multipin nets. The algorithm uses Lee's routing algorithm for the connection of pins. Prim's algorithm is used for solving the *minimum spanning tree* problem. Akers' algorithm is a modification of this algorithm into the *Steiner tree* problem. The pins in the multipin net are called the terminal cells. The algorithm is given in Fig. 2.8.

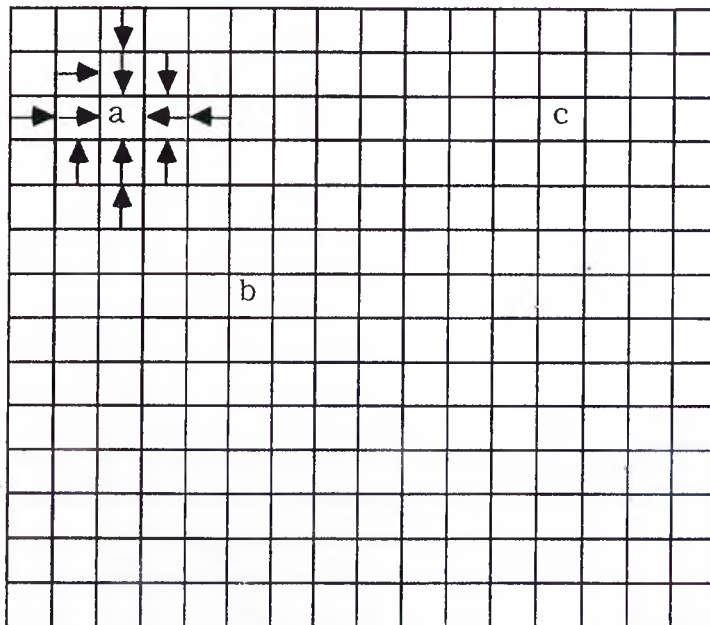
At step 2 of the algorithm, the set of sources consists of all the visited cells during the previously and currently constructed respective shortest paths. The propagation of the new front waves starts from all of these cells taking them as new sources. Fig. 2.9 and 2.10 illustrates the steps of the Akers' algorithm for connection of a multipin net.

-
1. Choose an arbitrary pin of the net and perform Lee's front wave expansion phase to propagate a unidirectional search wave starting from this pin cell until it hits another terminal cell.
 2. Perform path recovery to construct the respective shortest path. Add all the cells visited during the path recovery phase into the set of sources.
 3. Perform the sweeping phase to unlabel all labeled cells during step 2 for the next search wave.
 4. Propagate unidirectional multi-search waves starting from the set of multi-sources in the expansion queue until an unlabeled terminal cell is reached.
 5. Goto step 2 until all terminal cells of the net are labeled.
-

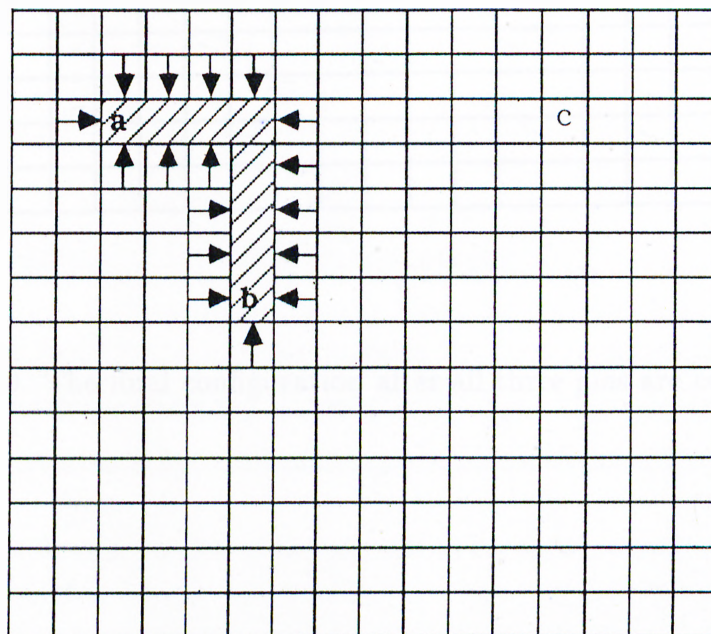
Figure 2.8. Sequential version of the algorithm for routing multipin nets using Prim's algorithm

2.2.2 Using Kruskal's Algorithm

If we base the *Steiner tree* computations onto Kruskal's algorithm [1] a faster algorithm [1] can be derived for the connection of multipin nets. This algorithm basically propagates search waves starting from all required pins (terminal cells). The algorithm using Kruskal's spanning-tree algorithm (Kruskal's Steiner tree algorithm) is given in Fig. 2.11. At the beginning all pins form a distinct tree. During the search phase of the algorithm when two search waves starting from different trees collide these two trees are merged and a new tree is formed. There are two procedures to perform the above mentioned task. $UNION(c_i, c_j)$ merges two different trees to which c_i and c_j belong. $TREE(c_i)$ returns the tree that c_i belongs to. Since each cell may belong to different trees, the status information of a cell in status array indicates both the label status (labeled, unlabeled, blocked) and a tree information to be used in procedures $TREE(c_i)$ and $UNION(c_i, c_j)$. Fig. 2.12 shows the routing of a single three pin net using this algorithm.



(a)



Blocked Cell

(b)

Figure 2.9. Front wave expansion phase of the Akers' algorithm beginning from the terminal cell 'a' (a) initial two cycles (b) Initial cycle after the first path (a to b) is connected.

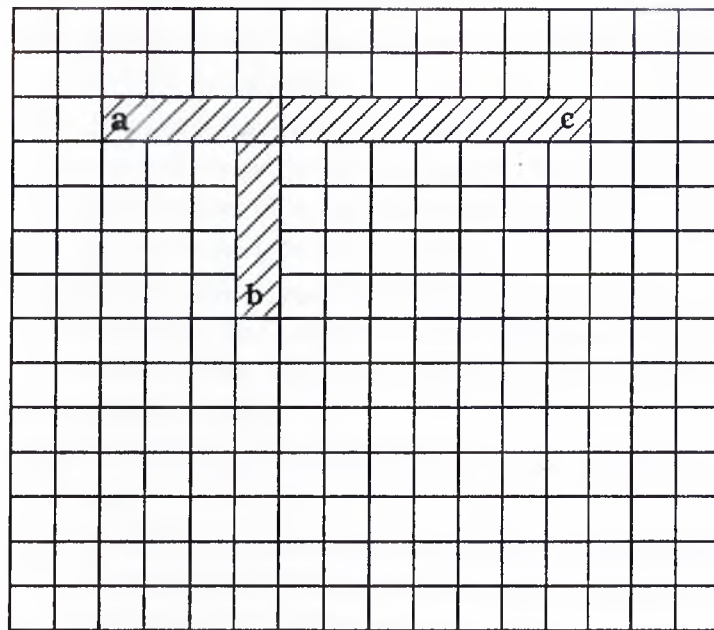
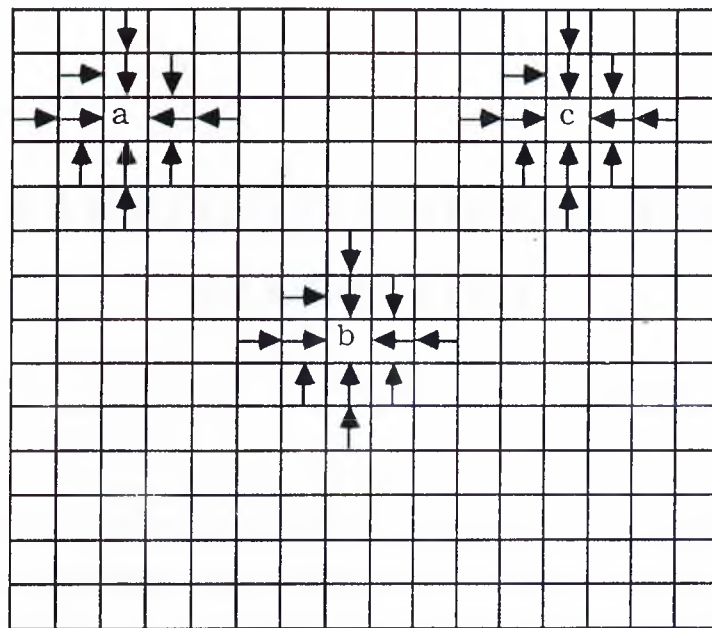


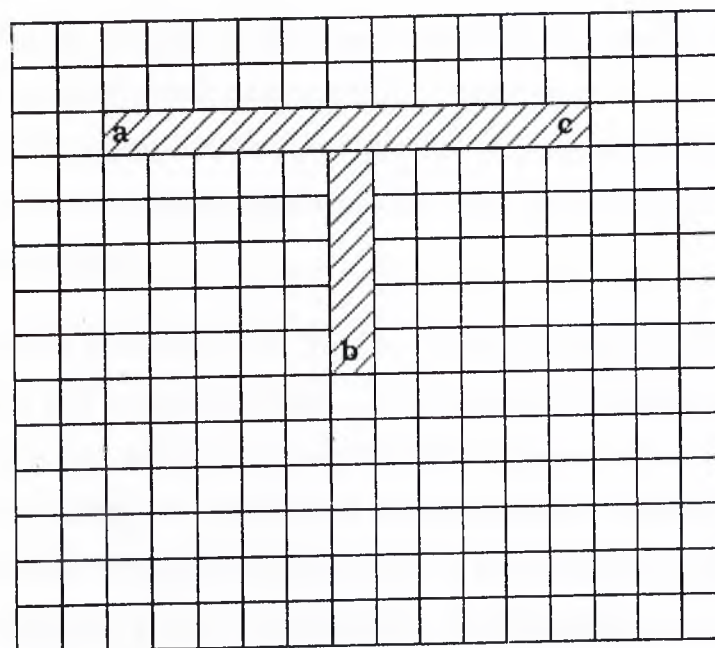
Figure 2.10. The final configuration after all three pins are connected


-
1. Add all the terminal cells into the *expansion queue*.
 2. Perform Lee's *front wave expansion* phase to propagate multi search waves starting from all terminal cells.
 - (a) choose a cell c from queue and examine its adjacent cells a_c 's for expansion.
 - (b) If a_c is a free cell then add a_c to expansion queue, label the cell to point to the parent cell c and update the tree information of a_c in status array so that the cell a_c belongs to the same tree as its parent.
 - (c) If a_c is labeled and $TREE(a_c)$ is not equal to $TREE(c)$ this indicates the collision of two different front waves (search waves), then call $UNION(c, a_c)$. Save the pair of colliding adjacent cells.
 - (d) If $TREE(a_c) = TREE(c)$ or a_c is blocked, ignore the cell a_c .
 - (e) If all four adjacent cells a_c 's of cell c are blocked or labeled such that they belong to the same tree with the cell c then add the cell c to the sweep queue.
 3. Repeat step 2 until all trees are merged.
 4. Perform path recovery starting from the collision points of different trees to form the interconnections between required pins.
 5. Perform the sweeping phase.
-

Figure 2.11. Sequential version of the algorithm for routing multipin nets using Kruskal's algorithm



(a)



 Blocked Cell

(b)

Figure 2.12. Routing of a three pin net using Kruskal's algorithm (a) Initial cycles of the algorithm (b) After connecting all pins

3. PARALLELIZATION OF LEE'S ALGORITHM

This chapter presents the principles and ideas used for the parallel implementation of Lee's algorithm. Lee's algorithm is a special case of multipin net algorithms. The ideas and principles for parallel implementation of Lee's algorithm will then be adopted for the parallel implementation of multipin net algorithms.

As indicated in Chapter 2, the Lee's Maze Routing algorithm consists of three phases, namely *front wave expansion*, *path recovery*, *sweeping* phases. Each phase of the algorithm has been parallelized, and the following sections present the proposed parallel algorithms for three phases. Each phase is considered independently.

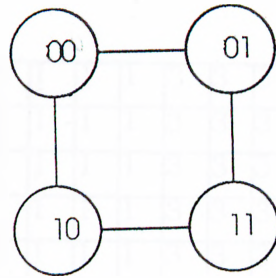
The sequential complexity of the Lee's maze routing is due to the front wave expansion and sweeping phases. As is discussed in Section 3.2, the parallel front expansion scheme proposed in this chapter avoids interprocessor communication during the distributed sweeping phase computations. However, interprocessor communication can not be avoided during the distributed front wave expansion phase computations. Furthermore, as is discussed in Section 3.1, the processor utilization during the distributed front wave expansion computations is very sensitive to the grid partitioning scheme employed. Hence, the grid partitioning and mapping scheme is chosen by mainly considering the computational requirements of the front wave expansion phase.

3.1 Grid Partitioning and Mapping

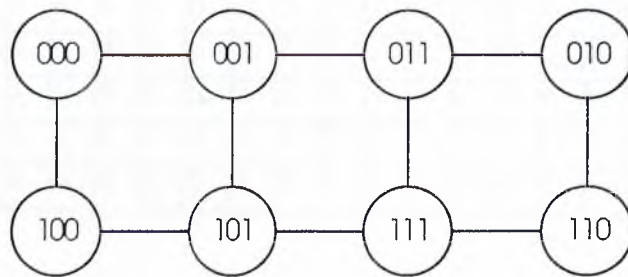
The effective parallel implementation of the *front wave expansion* algorithm on a hypercube multicomputer requires the partitioning and mapping of the *expansion* computations and the data structures associated with the grid (i.e. *status* array). This partitioning and mapping should be performed in a manner that results in low interprocessor communication overhead and low processor idle time. Even partitioning of the status array onto the node processors is an easy task since a fixed size two dimensional grid is to be partitioned and mapped to the processors of the hypercube. Even partitioning of the expansion computations, on the other hand, is not easy because expansion computations are not predictable and depends on the data (blocked cells etc.). However, as will be explained later, the partitioning of the status array affects the partitioning of expansion computations, and as a result, affects the processor utilization and interprocessor communication overhead.

In the front wave expansion phase, the *atomic* operation can be considered as the *expansion* of a single cell in the current *front wave*. In this *atomic* process, the *north*, *east*, *south*, and *west* adjacent cells of the cell being expanded are examined. Hence, the nature of communication required in *front wave expansion* phase corresponds to a two dimensional mesh. That is, each processor needs to communicate only to its *north*, *east*, *south*, and *west* neighbors. Hence, only mesh embedded hypercube structure will be considered and partitioning and mapping of the grid is done considering only a mesh embedded hypercube structure. It is well known that, a $2^{\lfloor d/2 \rfloor} \times 2^{\lfloor d/2 \rfloor}$ processor mesh can be embedded into a d -dimensional hypercube [11]. Fig. 3.1(a), (b), and (c) represent the mesh embedding into hypercubes of dimensions 2, 3, and 4, respectively.

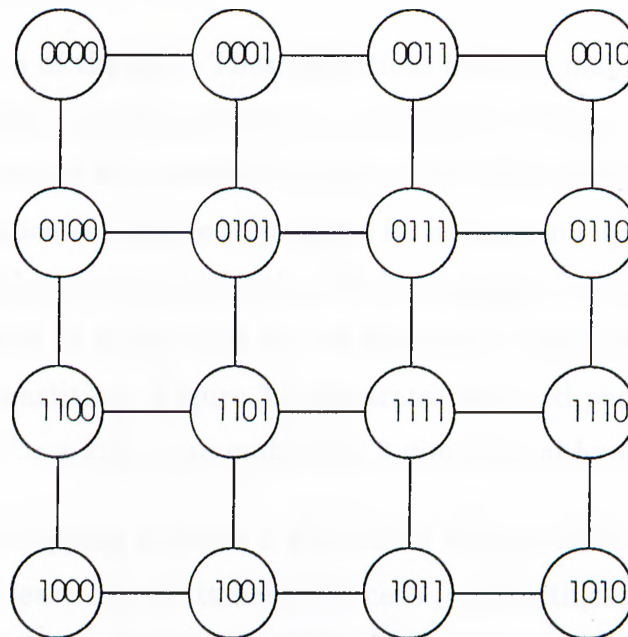
The even partitioning and mapping of the global status array is trivial since a two dimensional $N \times N$ mesh grid is to be partitioned and mapped onto a two dimensional mesh embedded hypercube. This mapping can be achieved by applying *tiled* decomposition in partitioning the grid. Assume that N is a power of two (i.e. $N = 2^n$) and $n > \lfloor d/2 \rfloor$. In the *tiled* decomposition, the grid is covered with rectangles of size $2^{n-\lfloor d/2 \rfloor} \times 2^{n-\lfloor d/2 \rfloor}$ starting from the top left corner and then proceeding left to right and top to bottom. Each rectangle cover



(a)



(b)



(c)

Figure 3.1. Mesh embedding (a) hypercube of dimension 2 (b) hypercube of dimension 3 (c) hypercube of dimension 4

0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
0	0	0	0	1	1	1	1	3	3	3	3	2	2	2	2
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6
4	4	4	4	5	5	5	5	7	7	7	7	6	6	6	6

Figure 3.2. Tiled decomposition of a 16x16 grid onto 2x4 mesh embedded 3-dimensional hypercube

defines a partition of the grid. These partitions are then mapped to processors in such a way that, partitions that are adjacent in the grid are mapped to adjacent processors of the processor mesh embedded in the hypercube. In this mapping, each processor will be responsible for holding and updating the status information (local status array) for the cells belonging to its local grid partition. Each processor will be responsible for the expansion computations for the cells in its local grid partition. Figure 3.2 illustrates the tiled partitioning scheme for a 16x16 grid for a 2x4 mesh embedded 3-dimensional hypercube.

In the given mapping scheme, a grid cell is defined to be a boundary cell if and only if at least one of its neighbor cells (i.e. north, east, south, west) is in a different partition. It is obvious that only boundary cells have a potential to cause interprocessor communication. The volume of possible interprocessor communication can be reduced by decreasing the number of boundary cells. It is well known that the number of boundary cells in a rectangular partition

with fixed number of cells can be minimized by choosing a square partition. The proposed mapping scheme achieves square partitions for even dimensional hypercubes (i.e. even d). For odd dimensional hypercubes the partitions are rectangles with long sides only twice the short sides. Such rectangle partitions minimize the number of boundary cells while maintaining the perfect balanced partitioning of the status array.

The *tiled* decomposition scheme ensures the mesh communication topology and even distribution of the data structures (status array) among the processors of the hypercube. This mapping scheme also minimizes the volume of interprocessor communication during the front wave expansion phase. However, in spite of these nice properties, it does not ensure the even distribution of the front wave expansion computations. Assume that, in Fig. 3.2, the source cell and the target cell are located at the top left and bottom right corner of the global grid, respectively. Also, assume that there are no blocked cells in the grid. During several initial cycles, only processor P_0 will perform front wave expansion computations while remaining processors stay idle. Similarly, during several final cycles, only P_6 will be busy with front wave expansion computations while the remaining processors stay idle. Hence, the tiled decomposition scheme yields very low processor utilization.

Processor utilization can be maximized by applying *scattered* decomposition scheme. Scattered decomposition scheme is achieved by imposing a periodic processor mesh template over the grid cells starting from the top left corner and proceeding left to right and top to bottom. Figure 3.3 illustrates the scattered decomposition of a 16x16 grid for a 2x4 mesh embedded 3-dimensional hypercube. In this scheme, adjacent grid cells in the grid are assigned to adjacent processors of the mesh embedded hypercube, thus ensuring the mesh communication topology. This scheme also ensures the even distribution of the status array among the processors of the hypercube. However, in the scattered decomposition, all local cells assigned to individual processors are boundary cells. In fact, for $d \geq 2$, all four neighbors of an individual local cell belong to adjacent processors. Hence, the expansion of any local cell in all four directions require interprocessor communications. Thus, scattered decomposition scheme causes large volume of interprocessor communication.

0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6
0	1	3	2	0	1	3	2	0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6	4	5	7	6	4	5	7	6

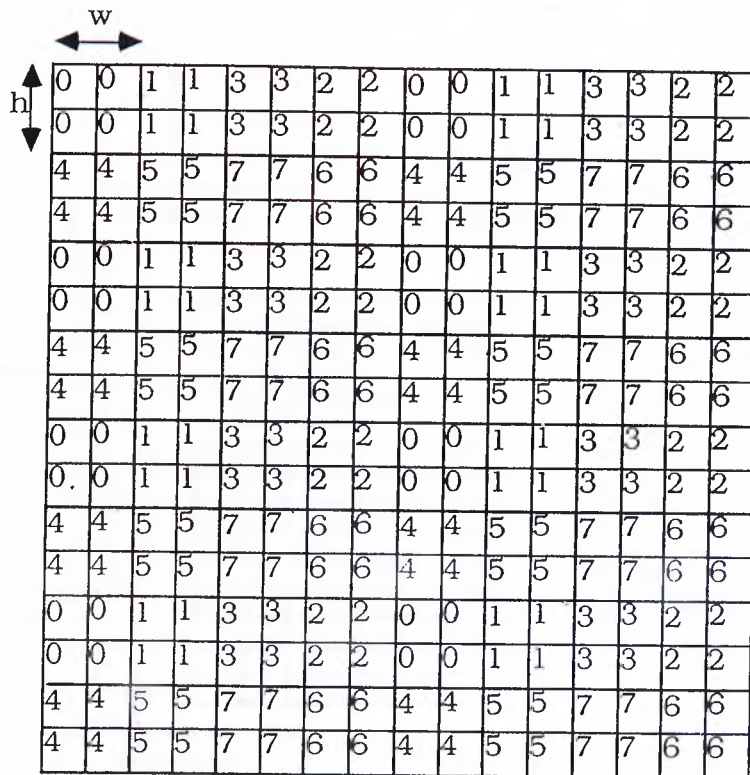
Figure 3.3. Scattered decomposition of 16x16 grid

The analysis of these two decomposition schemes shows that there is a trade-off between processor utilization and volume of interprocessor communication. This trade-off is resolved by combining tiled and scattered decomposition schemes as is also proposed in [8]. In this scheme, $N \times N$ grid is transformed into a coarse grid by applying tiled decomposition assuming much larger $P_1 \times P_2$ mesh processor array, where P_1 and P_2 are powers of two, $P_1 = P_2$ for even d , $P_2 = 2P_1$ for odd d , and $P_1 \times P_2 \gg 2^d$ but P_1 and $P_2 \leq N$. Hence, effectively the $N \times N$ grid is covered with $h \times w$ rectangle (or square) subblocks where $h = \frac{N}{P_1}$, $w = \frac{N}{P_2}$ and h, w values are power of two where $h \ll N$. Then, scattered decomposition is applied to the generated coarse grid. That is, a periodic processor mesh template (of size $2^{\lfloor d/2 \rfloor} \times 2^{\lfloor d/2 \rfloor}$) is imposed over the $h \times w$ grid subblocks starting from the top left corner and proceeding left to right and top to bottom. Figure 3.4 shows the mapping of 2×2 and 4×2 grid subblocks over a 16×16 routing grid to the processors of a 2×4 processor mesh embedded in a 3-dimensional hypercube. In this scheme, $h = w$ for even hypercube dimensions and $h = 2w$ for odd hypercube dimensions. The width w of the rectangles constitutes the characteristic of the decomposition. This mapping scheme reduces to scattered mapping scheme when $h = w = 1$ ($P_1 = P_2 = N$) and it reduces to tiled decomposition scheme when $h = \frac{N}{2^{\lfloor d/2 \rfloor}}$ and $w = \frac{N}{2^{\lfloor d/2 \rfloor}}$ ($P_1 = 2^{\lfloor d/2 \rfloor}$ and $P_2 = 2^{\lfloor d/2 \rfloor}$). The processor idle time will decrease with decreasing w . However, the volume of interprocessor communication will decrease with increasing w . Hence, the trade-off between processor utilization and volume of interprocessor communication can be resolved by selecting an appropriate value for w .

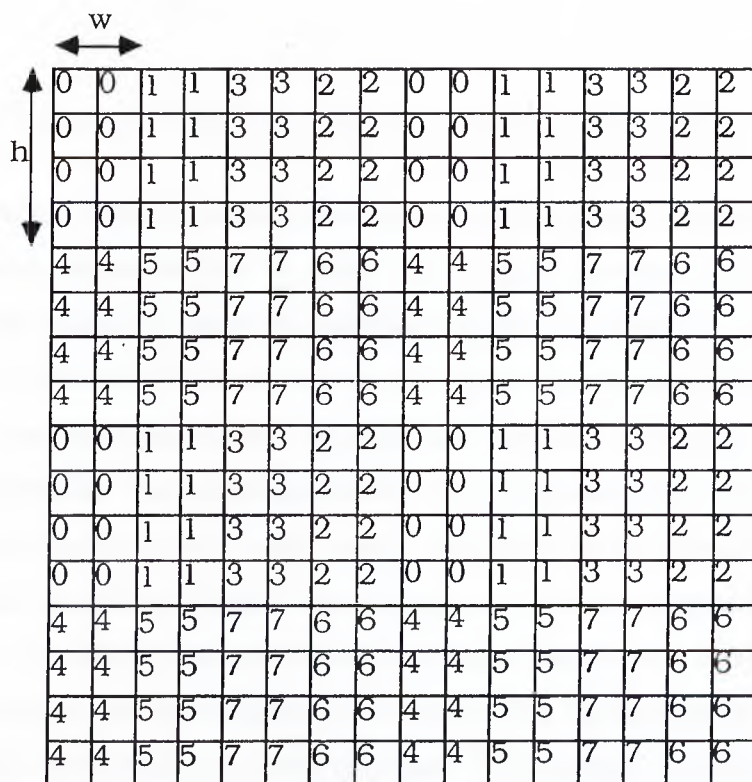
3.2 Parallel Front Wave Expansion

3.2.1 Expansion Starting From Source Only

The expansion starting from source only (Sonly) scheme initiates a *breadth first* search starting from the *source* cell as in the original Lee's algorithm[7]. The *status array* is partitioned and mapped onto the node processors according to the mapping scheme presented in section 3.1. Hence, each processor stores and maintains a local *status array* to keep dynamic and static status information



(a)



(b)

Figure 3.4. Decomposition of 16x16 grid into subblocks of (a) $h = w = 2$
 (b) $h = 2w = 4$

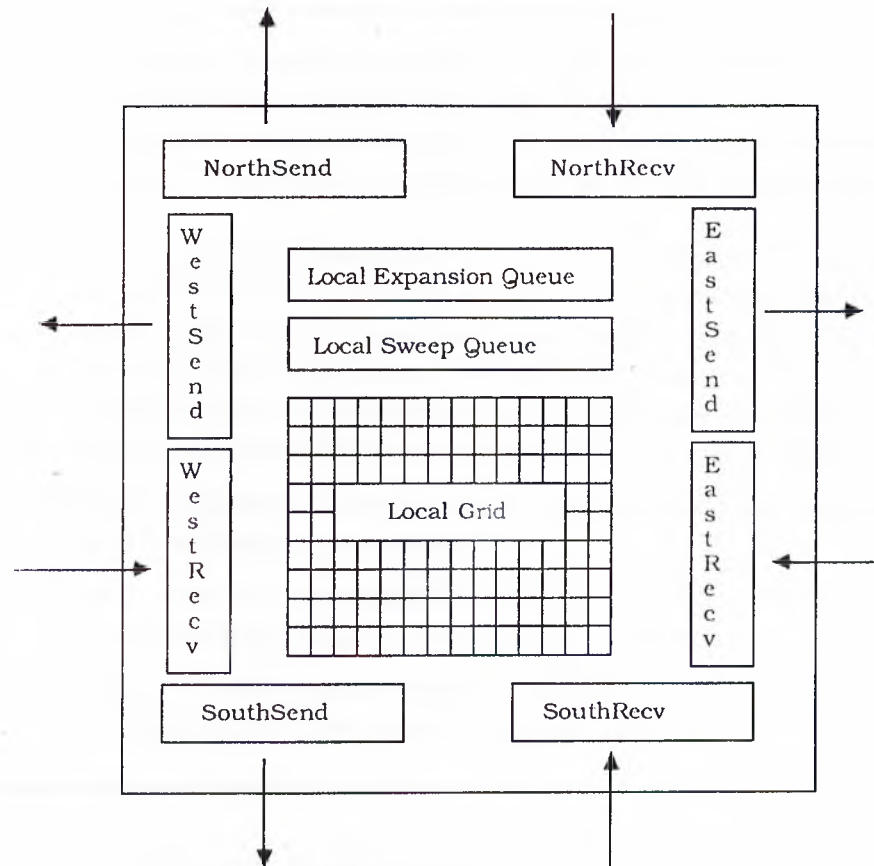


Figure 3.5. Local data structures for a node processor.

for its local cells. Each processor maintains a local expansion queue to process the *front wave expansion* for its local cells. Each processor also maintains a local *sweep* queue to store the blocked expansion paths for the *sweeping* phase. As is indicated in Section 3.1, the expansion of each local boundary cell require communication with at least one neighbor processor. Hence, in order to accomplish that communication, each processor also maintains four different *send* queues (*north*, *east*, *south*, and *west send* queues) for storing and transmitting cell coordinate information to its four neighbor processors in the mesh. Similarly, each processor maintains four *receive* queues to store information sent by its four neighbor processors. Fig. 3.5 illustrates the view of the local data structures for a node processor. The parallel algorithm is given in Fig. 3.6

The local x and y coordinates of the local cells in the current *wave* are stored in the local circular queues. One byte word $Status(x, y)$ is allocated in

Initially, all the local queues are empty. The host processor broadcast the coordinates of the source cell and target cell to all processors. The processor which owns the source cell location adds the local coordinates of the *source* cell to its local queue. Then each processor executes the following algorithm.

1. Each processor examines the cells in its local expansion queue for expansion in four directions. The local adjacent cells of the cells being expanded are examined for adding them to the local expansion queue for later expansion. The adjacent cells which are detected to belong to grid partitions assigned to neighbor processors are added to the corresponding *send* queues for later communication.
 2. Each processor transmits the information in its four *send* queues to their destination processors.
 3. Each processor examines the cells in its four *receive* queues for adding them to its local queue for later expansion.
 4. Each processor repeats steps 1, 2, and 3 until host signals the termination of the *front wave expansion* phase.
-

Figure 3.6. Node program for the Sonly scheme

the two dimensional local *status* array for the status information of each local cell. The encoding of the status of a cell is shown in the Figure 3.7. The least significant three bits (bits 0,1,2) of $Status(x,y)$ hold the current *routing* status of the local cell located at the local x and y coordinates. Six different *routing* status information are; *blocked*, *unlabeled*, and *labeled from north*, *east*, *south* and *west*. The status information is obtained by examining the value of these three bits. The values and the meanings are

000 : Unlabeled
 001 : Blocked
 010 : Labeled from North (Connected to North)
 011 : Labeled from South
 100 : Labeled from West
 101 : Labeled from East.

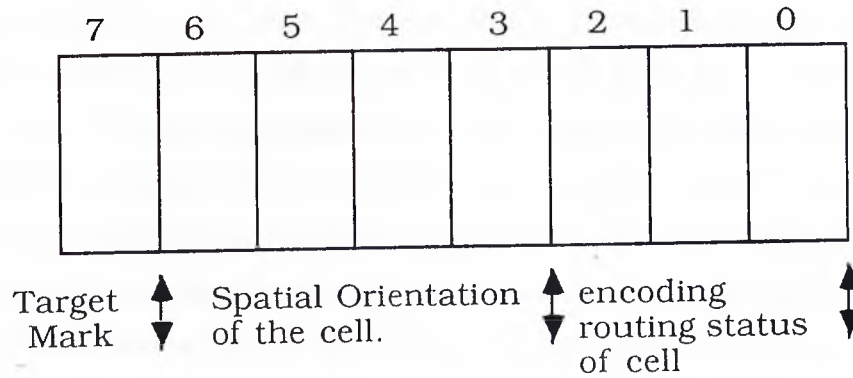


Figure 3.7. Encoding the status information

As is discussed earlier, the expansion of a boundary cell necessitates inter-processor communication. Hence, each processor should store spatial orientation information in its status array for its local cells. Four bits (bits 3,4,5,6) in the one byte word are reserved for *spatial orientation* information in the local grid of the processor. The assertion of a particular bit indicates that the cell is in the local partition boundary in the corresponding expansion direction. That is, the adjacent cell in that expansion direction is not a local cell, and it is assigned to the neighbor processor in that *expansion* direction. Hence, that adjacent cell should be added into the particular send queue in that expansion direction. Note that, a cell may be a boundary cell in more than one *expansion* directions. For example, if bit b_5 and bit b_6 is asserted then the cell is at the *north-west* boundary. Also note that, bits $b_6b_5b_4b_3 = 0000$ indicates that the cell is a local interior cell whose four adjacent cells belong to the local grid partition. This bitwise horizontal encoding scheme for spatial orientation is chosen in order to decrease the complexity of the local expansion computations.

Whenever, the processor which owns the *target* cell labels the *target* cell, (either at Step 1 or Step 3), it signals the host about the successful termination. The host processor, upon receiving such a message, broadcasts a message to all processors to terminate the *front wave expansion* phase and enter into the *path recovery* phase. Then, the processor which owns the target cell initiates a *path recovery* beginning from the *target* cell.

If a cell is found to belong to the partition assigned to a neighbor processor, the x,y coordinates of the cell are put into the corresponding send queue and sent to the neighbor processor. However, due to the partitioning of the global grid onto processors, each cell in a processor's local grid has local coordinates. Hence, when the non-local adjacent cell a_c of a boundary cell c is transferred to a neighbor processor, the x,y coordinates of the cell a_c have to be converted to the local coordinates in the receiving processor. This conversion operation is an overhead associated with the parallelization. The expansion computation associated with an individual cell has fine granularity. Hence, an efficient scheme should be devised for this conversion in order to keep this overhead low. This conversion can be performed using two schemes. In the first scheme, the local coordinates of the cell is converted to global coordinates and then to the local coordinates of the receiving processor. Such an operation is computationally expensive operation. In the second scheme, the local-to-local conversion is achieved directly. This efficient scheme is briefly discussed in the following paragraph.

Note that, the left-top corner of the global grid (and local grids) is chosen as the origin of the $x-y$ coordinate system. Hence, x -coordinate increases in east direction and y -coordinate increases in the south direction. In the proposed mapping, grid cells in a particular row (column) of the global grid are partitioned into successive contiguous blocks of size w (h). Then, successive cell blocks in a row (column) of the global grid are mapped to the successive processors in a periodically repeating row (column) of the processor mesh template. Hence, local x -coordinates (y -coordinates) of a boundary cell between two successive processors in an individual mesh template differ by w (h). However, if a cell is a boundary cell between two boundary processors in a row (column) of the repeating processor mesh template, then its local coordinates are equal in these two adjacent processors. Note that, local y -coordinates (x -coordinates) of all cells in the same row (column) of the global grid are equal in all processors. Fig. 3.8 illustrates the local indexing of the local status arrays for the mapping of 16×16 grid to a 4×4 mesh embedded 4-dimensional hypercube. As is seen in this figure, w should be added/subtracted for the local-to-local conversions during the east-west communications between the following pairs of adjacent

processors (12-13, 13-15, 15-14), since 12-13-15-14 constitute a row of the processor mesh template. There is no need for conversion during the east-west communications between the pair of adjacent processors (14-12), since this pair of adjacent processors is the boundary processors of the repeating processor mesh template. Similarly, h should be added/subtracted for the local-to-local conversion during the north-south communications between the following pairs of adjacent processors (1-5, 5-13, 13-9), since 1-5-13-9 constitute a column of the processor mesh template. There is no need for conversion during the north-south communications between the pair of processors (9-1), since this pair of adjacent processors is the boundary processors of the repeating processor mesh template.

In this scheme, each processor statically determines the number to be added for the local-to-local conversion for each boundary expansion direction by examining its particular location in the processor mesh template. Hence, the overhead associated for the local-to-local conversion required during the expansion of a boundary cell in the boundary direction is only a single addition operation.

In spite of the given partitioning scheme, the above parallel algorithm may result in low processor utilization for large h and w values. Some processors may still stay idle particularly during the initial and final *front wave expansion* cycles. This is due to the *expansion* of a single *front wave* beginning from the source cell. Note that, finding a routing path from source to target is equivalent to finding a path from target to source. Hence, two *front waves*, one beginning from the source (*source front wave*) and the other one beginning from the target (*target front wave*), can be *expanded* concurrently. If the source and the target cells are assigned to different processors, this scheme has a potential to increase the processor utilization.

3.2.2 Expansion Starting From Source and Target

This scheme initiates *breadth first* search starting from the *target* cell as well as starting from the *source* cell[14]. Figure 3.9(a) illustrates the initial two cycles of this scheme for the example grid shown in Figure 2.1. The Figure 3.9(b)

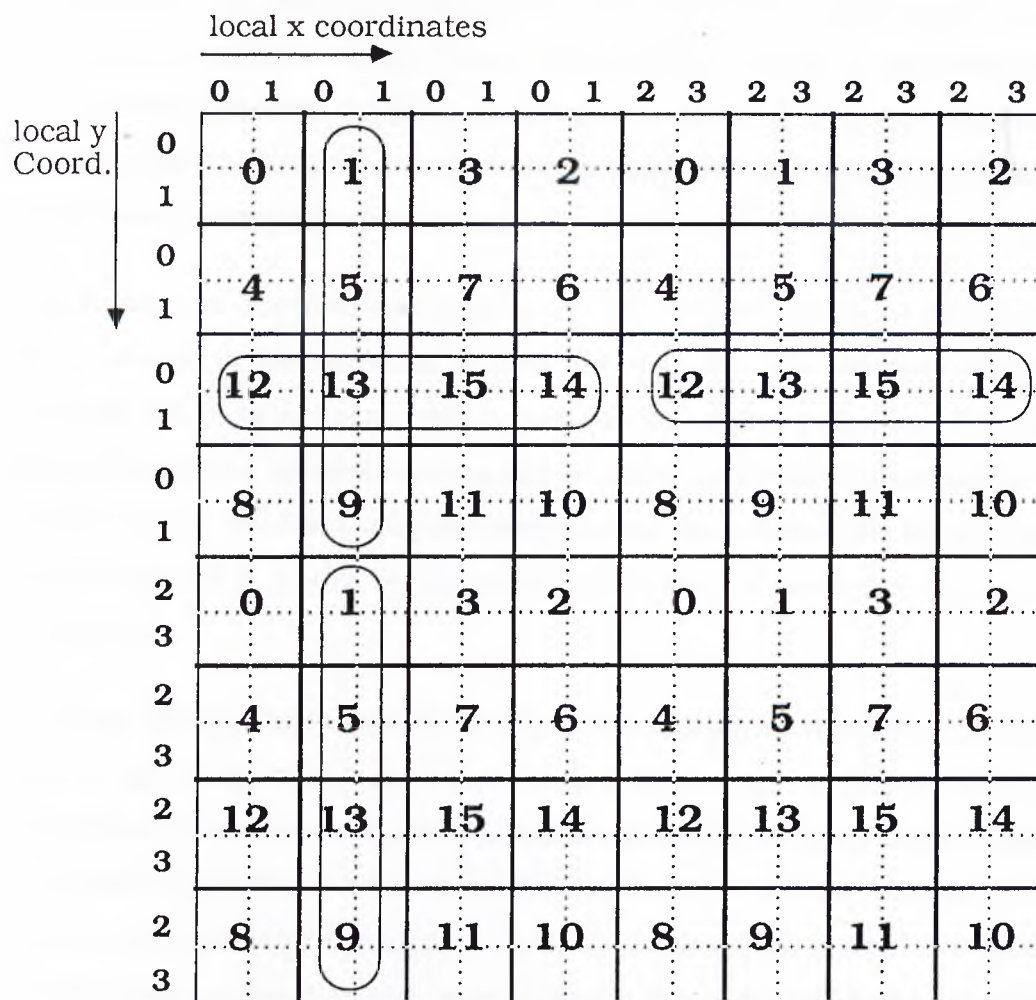


Figure 3.8. Mapping of local coordinates onto the repeating mesh template

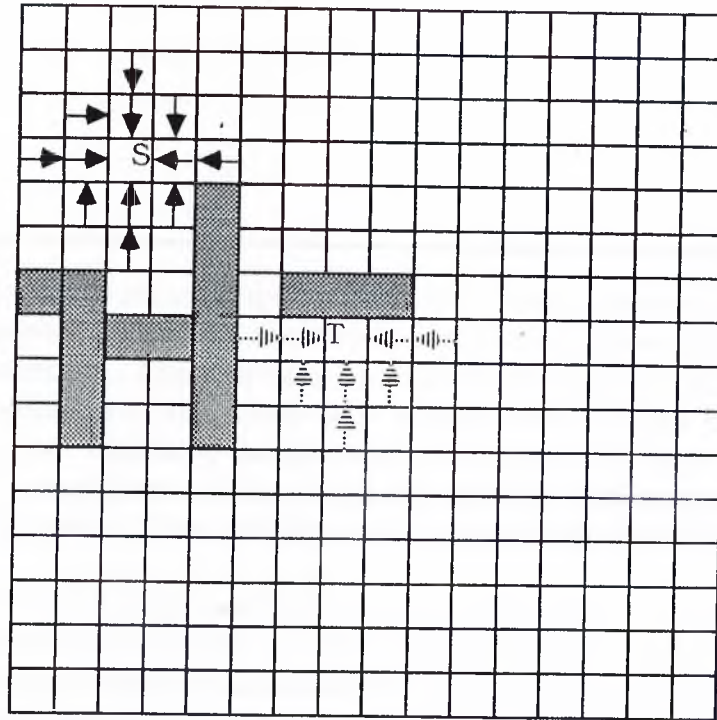
illustrates the *collision* of two concurrent *front waves* initiated from the source cell s and the target cell t . The parallel algorithm for expansion starting from source and target scheme (S+T scheme) is given in Fig. 3.10.

In this scheme, three identifying parameters are needed for each cell in the queues; its local x,y coordinates, and a *tag* to indicate the type of the *front wave* it belongs to. The cells that belong to the *source front wave* are identified with *positive* x,y coordinate values, whereas, the cells that belong to the *target front wave* are identified with *negative* x,y coordinate values. This *tagging* scheme is chosen in order to keep the local memory requirement due to local queues and the volume of communication low.

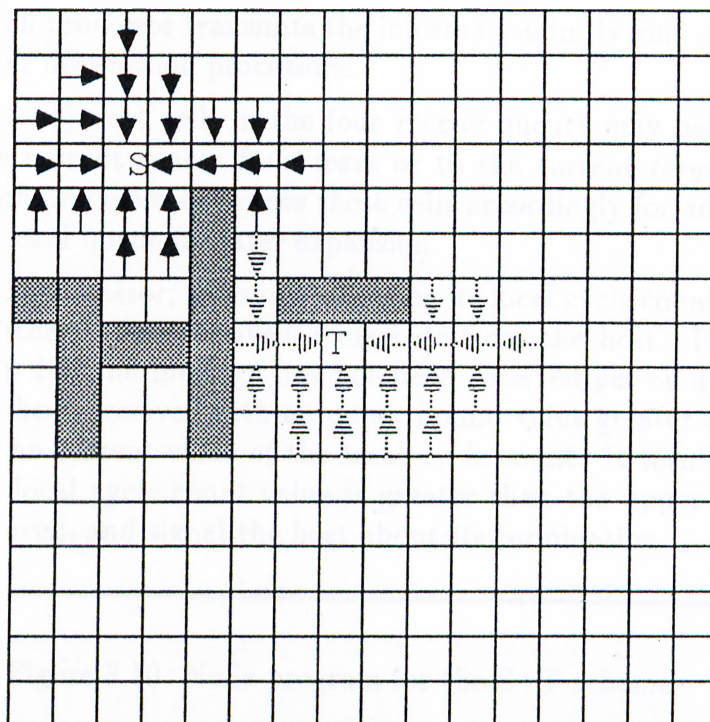
Note that, in the local status array the most significant bit of each status byte is unused in the encoding scheme shown in Fig. 3.7. Hence, the most significant bit of each status byte is reserved for *tagging* purposes. If a local cell has a *blocked* or *unlabeled* routing status, this bit conveys no information. If, however, a local cell has a *labeled* routing status, the value of this bit indicates whether the cell is *labeled* on the *target front wave*, or *labeled* on the *source front wave*.

During the expansion process at Step 1, the routing status of four adjacent cells of a cell being expanded are examined. If the current routing status of an adjacent cell is *unlabeled*, the local x,y coordinates of the adjacent cell are *tagged* accordingly (depending on the *tag* of the cell being expanded) and added to the local queue for later expansion. Then, the adjacent cell is labeled with the reverse expansion direction and *tagged* with the *tag* of the cell being expanded in the local *status* array. However, if the current routing status of an adjacent cell is *blocked* or *labeled* with the same *tag* of the cell being expanded, then the adjacent cell is discarded and added into the local sweep queue. Otherwise, if the adjacent cell is already *labeled* with a different *tag* compared to the *tag* of the cell being expanded, it shows the *collision* of two different *front waves*. The processor which detects the *collision* at Step 1, signals the host about the collision. It also includes the current value of the local cycle count and the local coordinates of the pair of adjacent cells in the message.

At Step 3, the local cells stored in the *receive buffers* are examined in a



(a)



(b)

Figure 3.9. Expansion starting from source and target scheme (a) Initial cycles
(b) Collision of two front waves

Initially, all local queues are empty and all local cycle counts are initialized to 1. The host processor broadcasts the coordinates of the *source* cell and the *target* cell to all processors. The processor which owns the source cell location adds the local coordinates of the *source* cell together with a source front wave *tag* to its local queue. Similarly, the processor which owns the *target* cell location adds the local coordinates of the *target* cell together with a *target* front wave *tag* to its local queue. Then each processor executes the following algorithm.

1. The cells in the local queue may belong either to the current *source front wave* or to the current *target front wave*. Each processor examines these cells accordingly for expansion in four directions. The local adjacent cells of the cells being expanded are examined for adding to the local queue for later expansion. The adjacent cells that are detected to belong to grid partitions assigned to neighbor processors are added to the corresponding *send* queues for later communication together with the *tag* of the cells being expanded.
2. Each processor transmits the information in its four *send* queues to their destination processors.
3. The adjacent cells in the four *receive* queues may belong either to the current *source front wave* or to the current *target front wave*. Each processor examines these cells accordingly for adding them to its local queue for later expansion.
4. Each processor, after incrementing its local cycle count by 1, checks whether it has received a message from the host. It proceeds to Step 1 if the message has not been received yet or if the message has been received with an upper bound value greater than or equal to the current value of the local cycle count. It terminates only if the local cycle count value is greater than the upper bound value received, and signal the host about its termination.

Figure 3.10. Node program for the S+T scheme.

similar way. However, each individual *collision* at this step will be detected concurrently by two neighbor processors. This situation corresponds to the mutual collision of two different type of *front waves* at two adjacent boundary cells in two adjacent grid partitions assigned to two neighbor processors. For example, if a processor detects a *collision* during the examination of a cell in its *east receive* queue, its *east* neighbor processor will concurrently detect the same *collision* during the examination of a cell in its *west receive* queue. In this case, those two neighbor processors will inform the host about the same *collision*.

The given parallel algorithm does not guarantee that all processors will be executing the same front wave expansion cycle at any instant of time. If a snapshot of the parallel system is taken, some of the processors may be found to be *leading* some others by a number of expansion cycles. A *leading* processor may be the first processor which detects a *collision*. Hence, if the host processor terminates the *front wave expansion* as soon as it receives a *collision* message, the path to be recovered may not be the shortest path from source to target. All *lagging* processors should be allowed to perform *expansion* until the cycle count of the *leading* processor which has detected the *collision* the first time. Those *lagging* processors have potential to detect collisions on earlier cycles. This is achieved by the scheme given at Step 4 of the algorithm. The host processor, after the receiving the first *collision* message, broadcasts the cycle count q in this message as an upper bound on the local cycle counts. The *leading* processors which have already performed the q -th *front wave expansion* cycle terminate and inform the processor about their termination. The *lagging* processors which have not yet performed the q -th *front wave expansion* cycle continue to execute the given algorithm until the q -th cycle. These processors also inform the host about their termination after executing the q -th cycle. Meanwhile, the host processor stores all the subsequent *collision* messages until it receives P termination messages. The host processor chooses the *collision* message with a minimum cycle count. Then it broadcasts the coordinates of the chosen pair of adjacent cells to signal the initiation of the *path recovery* phase.

As is explained earlier, at step 3, two neighbor processors will inform the

host about the same mutual collision. More than one pair of processors may try to inform the host about many such mutual collisions during the same or successive few cycles. In order to decrease message traffic for such situations, the pair of processors detecting the same collision may apply a local precedence relation for informing the host. For example, in a east-west mutual collision, only the processor which detects the collision during examining its west receive queue will inform the host about the collision. The neighbor processor which detects the same collision while examining its east receive queue will take no action for such collisions. The north-south mutual collisions are resolved similarly. This scheme also relieves the host processor from trying to detect such mutual collisions.

In the proposed parallel algorithm, the number of *expansion* cycles to be performed in the *front wave expansion* phase is reduced by a factor of two compared to the original parallel algorithm. Hence, the total number of local communications is reduced by a factor of two, since the number of local communications per *expansion* cycle is fixed to four. The proposed algorithm is also expected to reduce the total number of expanded cells, on the average, almost by a factor of two. Assume a large grid with no blockages. If the original algorithm requires q *expansion* cycles to reach the *target* cell, the proposed algorithm will require only $q/2$ *expansion* cycles to reach the *collision* of two front waves. The total number of cells expanded in the original algorithm will be $1 + 2q(q + 1)$ compared to $2 + q(q + 2)$ in the proposed algorithm[14]. The total number of expanded cells in both algorithms will of course vary when blockages exist in the grid. Unfortunately, the proposed algorithm increases the amount of computation required for the expansion of an individual cell. However, the computational overhead per cell expansion will decrease with increasing blockage percentage in the grid. As is discussed above, an unused bit in a local cell *status* word and unused bits in the cell coordinate information words are used for *tagging* purposes. In this way, the total memory requirement and communication volume requirement for an individual cell is not increased compared to the original algorithm. Hence, the proposed algorithm will also reduce the total volume of communication since the total number of expanded cells is reduced. Furthermore, the proposed parallel algorithm will increase the processor utilization, compared to the original algorithm, when source and

target cells are assigned to different processors. The relative increase in the processor utilization will grow with increasing h,w parameters, and with increasing mesh distance between the two processors which own the source and the target cells.

3.3 Termination Detection

At Step 3 of both parallel algorithms, each processor issues four *synchronous (blocking) receive* messages in order to receive information from its four neighbor processors. Hence, processors do not proceed to the next *front wave expansion* cycle before receiving messages from all four neighbors. In order to prevent deadlock, each processor always send messages to its four neighbors even if its *send queues* are empty. Hence, the *synchronous receive* messages at Step 3, constitute a local synchronization between neighbor processors. Due to this local synchronization, both of the above parallel algorithms are guaranteed to find the shortest path between the source and the target whenever a path from source to target exists. However, these parallel algorithms will not terminate if no path exists between the source and the target. In the parallel front wave expansion algorithms, the unsuccessful termination condition occurs only when the local expansion queues and the local receive queues of all processors become empty at the same expansion cycle. The schemes to provide global termination detection for such cases are discussed in this section.

3.3.1 Global Synchronization

In this scheme, host processor is used to perform global synchronization at the end of each *expansion cycle*[8]. At the end of each *expansion cycle* (after Step 4), each processor sends a message to the host to indicate whether its local queue is empty or not. Then each processor issues a *synchronous receive* to receive an enable message from the host to proceed with the next *expansion cycle*. Host waits for receiving local queue status information from all processors by issuing P successive *synchronous receives*. If host detects that all local queues are empty, then it broadcasts a signal to all processor of the

hypercube to indicate the unsuccessful termination of the *front wave expansion* phase. Otherwise, host broadcasts an enable signal to all processors. Hence, no processor can start the $(q+1)$ -th *expansion* cycle until all other processors complete the q -th *expansion* cycle. In this scheme, for each *expansion* cycle, host receives P messages serially and broadcasts the enable message in $\log_2 P$ time. Thus, this scheme for global termination detection introduces a large amount communication overhead. Furthermore, it decreases the processor utilization since each processor has to be globally synchronized with all other processors (through host) before beginning the next *front wave expansion* cycle.

3.3.2 Counter Termination Scheme1

This scheme is very similar to the scheme proposed in [8]. In this scheme, host processor holds a counter to count the number of non-empty processors. This counter is initialized to 1 in *Sonly* scheme. In *S+T* scheme, this counter is initialized to 1 if source and target cells belong to the same processor or 2 if source and target cells belong to different processors. The host and node programs for this version of the counter termination scheme for a P processor hypercube are given in Fig. 3.11.

In this scheme host program has to wait for Δ seconds after counter becomes 0, since there may be transient messages. This Δ seconds is a machine specific parameter. After Δ seconds, if there is no message then it means that each processor is empty and there is no transient message. Hence, the host program can terminate the program.

3.3.3 Counter Termination Scheme2

In order to avoid the use of such a machine specific parameter, another counter termination scheme is proposed. In this scheme, host maintains a one dimensional array to keep a counter for each *front wave expansion* cycle. The host and node programs for this version of the counter termination scheme for a P processor hypercube are given in Fig. 3.12.

Host Program

1. Initialize the counter.
2. Enable *front wave expansion*
3. **If** a status signal (+1/ - 1) is received **then**
 - (a) counter = counter + received value.
 - (b) **If** *counter* = 0 **then**
wait for Δ second
if no message arrives **then** terminate the program.
4. Goto step 3.

Node Program

1. Wait for enable signal from host.
 2. Perform the next *front wave expansion* steps of the corresponding algorithm.
 3. **If** the status of processor changes from *non-empty* to *empty* **then** send -1 to host as status message.
 4. **If** the status of processor changes from *empty* to *non-empty* **then** send +1 to host processor as status message.
 5. Goto step 2.
-

Figure 3.11. Host and node programs for the counter termination scheme 1.

Host Program

1. $counter(q) = 0$ for all possible q
2. Enable *front wave expansion*
3. **If** a signal with label q received **then**
 - (a) $counter(q) = counter(q) + 1$
 - (b) **If** $counter(q) = P$ **then** terminate
4. Go to Step 3

Node Program

1. $q = 0$
 2. Wait for an enable signal from the host
 3. $q = q + 1$
 4. Perform steps 1, 2, and 3 of the front wave expansion algorithm
 5. **If** all local queues are empty **then**
 - (a) signal the host with the cycle count q
 6. Perform step 4 of the front wave expansion algorithm
 7. Go to Step 3
-

Figure 3.12. Host and node programs for the counter termination scheme 2.

The condition $counter(q) = P$ checked by the host indicates that all local queues are empty at the q -th expansion cycle. Hence, the given algorithm ensures the global termination detection when there is no path from source to target.

3.4 Overlapping Communication with Computation

In the parallel algorithms given in sections 3.2.1 and 3.2.2, after sending the data in four send queues, each processor may wait idle for the arrival of data from its four neighbor processors. These parallel algorithms can be re arranged as shown in Fig. 3.13 to reduce the idle time. The overlapped algorithm given in Fig. 3.13 is similar to the one proposed in [8, 12]. Note that, in the non-overlapped schemes (Figures 3.6 and 3.10) the non-local adjacent cells (of depth $q + 1$) are transmitted at step 2 after being added to the send queues (at step 1) of the same expansion cycle. In the overlapped scheme, the transmission of the non-local adjacent cells (of depth $q + 1$) encountered during step 3 of the q^{th} expansion cycle are delayed until step 1 of the next expansion cycle. Hence, the transmission of data in the send queues constructed in the previous expansion cycle is initiated before the local expansions in the current cycle. Thus, the local expansion computations (step 3) performed by each processor are overlapped with the communication time required for the initiation and arrival of the data from its four neighbor processors.

However, due to the load imbalance, all four messages may not be received upon completion of the local expansion computations at step 3. In order to reduce idle time in such cases, each processor performs *in place* expansion computations for the cells in already received queues. Each processor checks the status of the receive queues by polling whenever it completes the *in place* expansion of the cells in the already received queues. In this way, the *in place* expansion computations performed by a processor on an already received queue may be overlapped with the transmission time(s) of the message(s) which are initiated from its other neighbors and which have not arrived yet.

Note that, in the non-overlapped scheme, local boundary cells in the receive

-
1. Issue four non-blocking receives.
 2. Issue four non-blocking sends for front wave cells at depth q to initiate the transmission of send queues to corresponding processors.
 3. Expand the cells (of depth q) in local expansion queue and add the adjacent cells belonging to the local grid of the neighbor processors into corresponding send queues.
 4. Poll the status of the issued non-blocking receives and expand the cells (of depth q) in already received queues and add their adjacent cells into the local queue or send queues. Repeat step 4 until all four messages are received and processed.
 5. Repeat the steps 1,2,3,4 until the target cells is reached (Sonly) or a collision occurs (S+T scheme).
-

Figure 3.13. Overlapping communication and computation

queues are examined only for the sake of adding them into the local expansion queue. That is, the status of these boundary cells are checked and these cells are labeled and added into the local expansion queue if they are found to be currently free *cells*. Hence, the expansion computations for such cells are performed on the next cycle. However, in the overlapped scheme, the free boundary cells are labeled and examined for expansion at step 4 of the same expansion cycle. Hence, a special *in place* expansion code can be written for each receive queue. Free boundary cells in a particular receive queue are not examined for expansion in the direction of that receive queue. For example, free boundary cells in the *West Receive Queue* are not examined for expansion in the west direction. In the non-overlapped scheme, the label information of the cell in the local expansion queue should be accessed from the status array and checked to avoid the expansion in the labeling direction. Hence, the *in place* expansion computations reduces the complexity of an individual expansion computation.

The overlap mentioned so far is in fact the overlap of the local computation in an individual processor with the initiation and transit times of the messages sent from its four neighbor to that processor. That is, algorithm in

Fig. 3.13 achieves overlap of communication and computation between neighbor processors. The algorithm given in Fig. 3.13 also achieves the overlap of communication and computation within each individual processor. These local overlaps can be achieved in both local receive and send operations as discussed in the following paragraphs.

The node executive ($NX/2$) of the iPSC/2 handles short messages (≤ 100 bytes) and long messages (> 100 bytes) differently. Short incoming messages are always stored first in a buffer inside the $NX/2$ area regardless of a pending *receive* for that message and then copied from the $NX/2$ buffer to the user buffer. However, long incoming messages are directly copied into the user buffer if a *receive* is pending for that message. If not, the message is kept in the $NX/2$ buffer until a *receive* is issued for that message. The local messages in the given algorithms are predicted and observed to be, in general, of long type messages. Hence, although the *receive queues* are to be processed at step 4, non-blocking *receive* messages are issued as early as possible at step 1 of each cycle. This scheme is chosen to ensure that *receives* are already pending for the incoming long messages so that they can be directly copied into the *receive queues* instead of being copied into the $NX/2$ area and then transferred into the indicated *receive queues* due to the late issued *receives*.

At step 2, the *send* operations for the *send queues* constructed at steps 3 and 4 of the previous cycle are initiated. Then, each processor continues execution by expanding the front wave cells in its local queue as indicated at step 3. Hence, the set-up time and the transit time for the four *send* operations at step 2 are overlapped with the computations at step 3 and even at step 4 of the given algorithm. The set-up and the local transit times for the *send* operations are overlapped on the cycle-stealing basis and the interprocessor network transit time of the messages are overlapped completely.

The non-blocking *send* messages issued at step 2 returns control back to the node program just after informing $NX/2$ about the *send* requests. The expansion computations at steps 3 and 4 may contaminate the buffers allocated for the *send queues* by inserting new cells (to be transmitted on the following cycle). A *switching buffer* is used for each *send queue* in order to ensure the

transmission of the correct data. A buffer of size $2M$ ($\text{Buffer}[2M]$) is allocated as a send queue, where M is the maximum number of front wave cells that can be transmitted between any two neighbor processors at any depth. The first half of the buffer ($\text{Buffer}[0\dots M-1]$) is transmitted, while expansion computations at steps 3 and 4 use the second half of the buffer ($\text{Buffer}[M\dots 2M-1]$) in *even* expansion cycles and vice versa in *odd* expansion cycles. In this scheme, a buffer area is used for transmitting data on alternate expansion cycles. Hence, synchronization on an asynchronous send message issued at step 2 of an expansion cycle can be delayed until step 3 of the next expansion cycle, thus providing the maximum overlap between communication and computation.

3.5 Asynchronous Scheme

The local synchronization steps in the given algorithms require that at the end of each front wave expansion cycle, each processor communicates to its four neighbor processors even if there is no data in send queues. In that case, each processor may send and receive null data. This can be avoided by sending messages to neighbors only when there is data in the send queues. The neighbor processors do not wait for the messages to arrive from neighbor processors and continue to the expansion of cells in the expansion queue. As a result of this modification, it is possible that the first time target cell is reached it is not the shortest path (see Fig. 3.15). In order to ensure termination only when a shortest path is found, it is necessary to associate cells with path depths from the source. The asynchronous parallel algorithms for both *Only* and *S+T* schemes are given in the following subsections. In both algorithms, since the cycle notion of the previous algorithms are lost, the *the counter termination scheme2* can not be used. Therefore, the *counter termination scheme1* has been used.

3.5.1 Expansion Starting From Source Only

This scheme initiates a breadth first search starting from source cell. The

Host Program

1. last-path = empty, spath = ∞
2. Enable node processes
3. Wait message from nodes.
4. **If** new-depth < spath **then**
 - i. last-path = new-path.
 - ii. spath = new-depth.
 - iii. send spath to nodes.
5. **If** all processors empty **then**
terminate the program.

Node Program

1. path-depth = ∞
 2. Remove a cell from local queue
If (cell's depth < (path-depth - 1)) **then** Expand the cell.
 3. Send non-empty send queues to neighbor processors.
 4. **If** a queue is received **then**
put the cells into the local queue.
 5. **If** a message is received from host **then**
path-depth = message value.
 6. Inform the host about status (path found etc.)
 7. Goto step 2.
-

Figure 3.14. Host and node programs for the asynchronous Sonly scheme.

algorithms for host and node programs are given in Fig. 3.14. The communications between the host and the node processors are performed at step 3 and 4 of the host algorithm and steps 5 and 6 of the node algorithm. In this scheme, additional information is used by a two-byte word for each cell in the send and receive queues to indicate its distance from source cell. Similarly, an additional two byte word is associated with each status word of the local status array. If a local cell has a labeled routing status on the *source front wave* then this word indicates the depth of the cell from the *source* cell.

The expansion of a cell at step 2 of the node algorithm needs more explanation. First, if the depth d_c of the cell c being expanded is larger than or equal to *path-depth* (i.e. $d_c \geq \text{path} - \text{depth}$) then that cell c is not expanded. This is done because a new path will not have a depth less than the previous path. If, however, the depth of the cell being expanded is less than *path-depth* then its adjacent cells are examined for expansion. If the current status of an *adjacent* cell a_c is already labeled then its depth d_{a_c} is compared with the depth d_c of the cell being expanded. If its depth is less or equal to the depth of the cell being expanded, (i.e. $d_{a_c} < (d_c + 1)$) then the adjacent cell is discarded. Otherwise, the adjacent cell a_c is added to the front of the local queue (instead of adding it to the rear of the local queue) with its depth being one more than the depth of the cell being expanded (i.e. $d_{a_c} = d_c + 1$) (see Fig. 3.16). Hence, the local queue behaves as a LIFO instead of FIFO for such collisions. The LIFO scheme implemented for such collisions implicitly prevents the further expansion of the cells in the local queue which are expanded originating from the indicated adjacent cell. This scheme is expected to increase the overall performance since the expansion of such cells in local queue with their current depth information have no chance to find a shorter path between the *source* and the *target*.

3.5.2 Expansion Starting From Source and Target

The host and node algorithms for the asynchronous $S+T$ scheme are shown in Fig. 3.17. The communications between the host and the node processors are performed at step 3 of the host algorithm and step 5 and 6 of the node

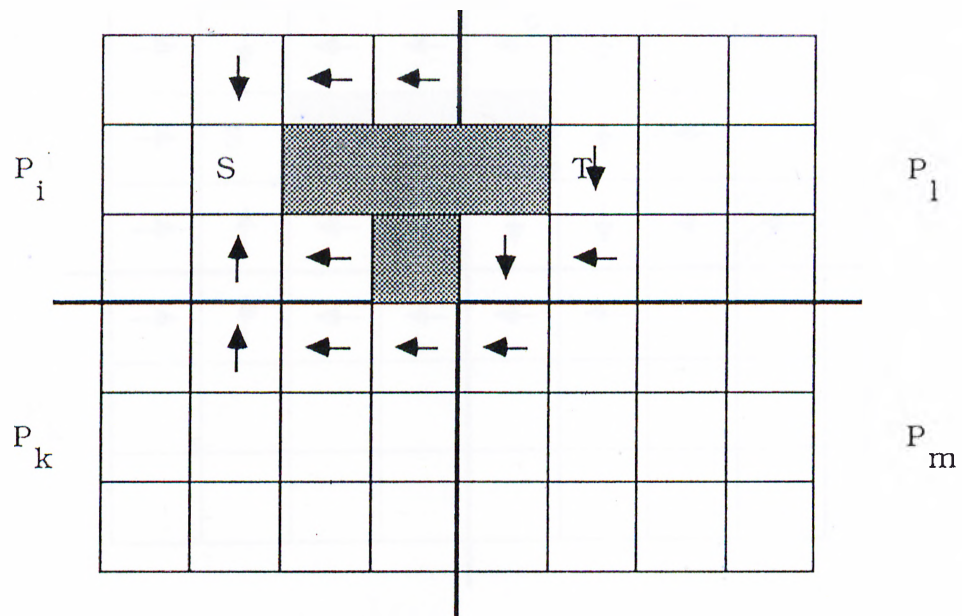


Figure 3.15. Failure to find the shortest path. If processors P_k , P_m and P_l are faster than processor P_i , target T can be reached by a longer path.

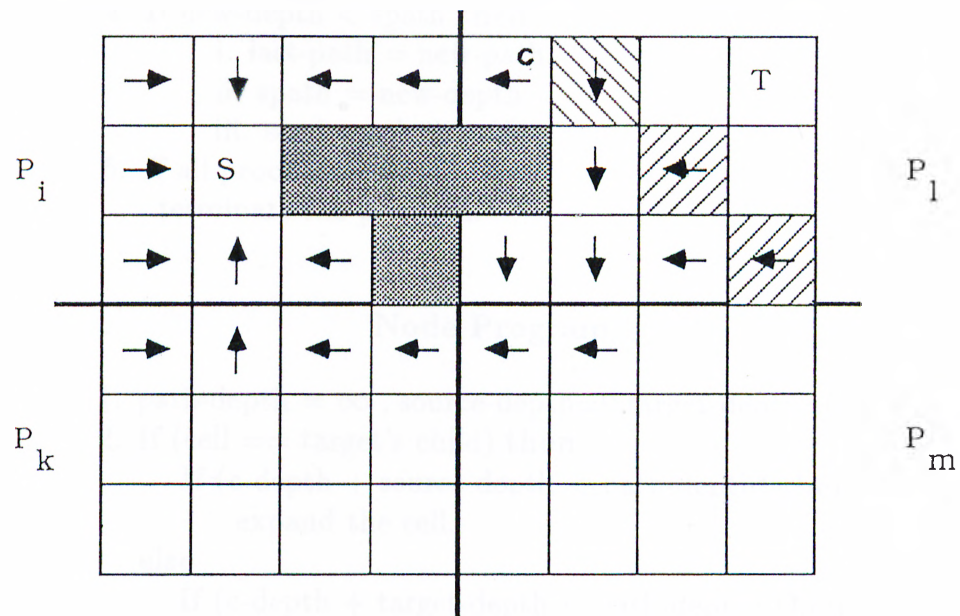


Figure 3.16. Labeling of an already labeled cell by a shorter path, the shaded cells have been reached by a shorter path hence the cells expanding from cell c will relabel the shaded cells

Host Program

1. last-path = empty,spath = ∞
2. Enable node processes
3. Wait message from nodes.
4. If new-depth < spath then
 - i. last-path = new-path.
 - ii. spath = new-depth.
 - iii. send spath to nodes.
5. If all processors empty then
terminate the program.

Node Program

1. path-depth = ∞ , source-depth=0,target-depth=0.
 2. If (cell == target's child) then
 - If (c-depth + source-depth < path-depth) then
expand the cell.
 - else
 - If (c-depth + target-depth < path-depth) then
expand the cell.
 3. send non-empty send queues to neighbors
 4. If (a queue is received) then
put the cells into local queue.
 5. If (message from host) then
 - If (first time receive)then
Exchange Minimum
path-depth = message value.
 6. Inform the host about status.
 7. Goto 2.
-

Figure 3.17. Host and node programs for the asynchronous S+T scheme.

algorithm. In this scheme, additional information is used by a two-byte word for each cell in the send and receive queues to indicate its depth either from source cell or from target cell. This information is represented by *c-depth* for a cell in the above algorithm. Similarly, an additional two byte word is associated with each status word of the local status array. If a local cell has a labeled routing status on the *source front wave* or *target front wave* then this word indicates the depth of the cell from the *source* cell and the *target* cell respectively.

The expansion of a cell c at step 2 of the node algorithm needs more explanation. If the current status of an *adjacent* cell a_c is labeled with the same tag of the cell c being expanded, then its depth d_{a_c} is compared with the depth d_c of the cell being expanded. If its depth is less than or equal to the depth of the cell being expanded (i.e. $d_{a_c} \leq d_c$), then the adjacent cell is discarded. Otherwise, the adjacent cell is added to the front of the local queue (instead of adding it to the rear of the local queue) with its depth being one more than the depth of the cell being expanded. Hence, the local queue behaves as a LIFO instead of FIFO for such collisions as is discussed in Section 3.5.1.

In node programs, two counters for counting the source cell depths (source-depth) and target cell depths (target-depth) are held. This is necessary because the path is found by the collision of two front waves. In addition, unlike *synchronous S+T* scheme, at step 5 of the node programs there is an exchange of the minimum source-depth and target-depth of the all processors. If the path found signal is received from the host processor, each node processor searches the its internal queue for minimum source-depth and target-depth, then these local minimums are exchanged in $\log(P)$ time for global minimums.

As is stated in [8], due to the need for keeping depth information, asynchronous scheme may be impractical for large grid dimensions.

3.6 Parallel Path Recovery and Sweeping

As is presented in section 3.7, it has been experimentally observed that *S+T* scheme outperforms the *Only* scheme and overlapped scheme gives better

performance results compared to the non-overlapped scheme. Hence, parallel path recovery and sweeping phases are derived assuming that overlapped S+T scheme is used for front wave expansion phase.

At the end of the front wave expansion phase, the host program broadcasts the global locations of the colliding adjacent cells. As is discussed earlier these two adjacent cells may belong to the same processor or to two different adjacent processors. Then, the processor(s) which own(s) these two adjacent cells c_1 and c_2 starts the distributed path recovery phase. The distributed path recovery phase is terminated after both the source and the target cells are labeled as blocked.

In the front wave expansion phase, local cells are examined for expansion at step 3 and step 4 of the overlapped S+T scheme (Fig. 3.13). The local cells examined at step 3 may be either boundary cells or *interior* cells. The local cells examined at step 4 are boundary cells. A cell c being examined for expansion should be added into the local sweep queue if all of its four adjacent cells are either blocked or already labeled with the same tag of the cell c . At this step, the decision for adding a local *interior* cell c_i into the local sweep queue can easily be taken locally by an individual processor since the dynamic status information for all four adjacent cells of a interior cell c_i is maintained by the same individual processor. However, the decision for adding a local boundary cell c_b into the local sweep queue may not be taken by an individual processor. The situation may be such that the cell c_b can only be expanded into non-local adjacent cell(s) that may be blocked or labeled with the same tag of the cell c_b . However, the updated status information about this/these non-local cell(s) is/are maintained by the neighbor processor(s). The decision for adding such local boundary cells into the local sweep queue may require extra communications to exchange updated status information about the relevant boundary cells. In fact, only these cells will introduce interprocessor communication during the sweeping phase. If c_b can be expanded into an at least one local adjacent cell then there is no need to add c_b into the sweep queue even if a non-local adjacent cell of cell c_b is found to be free by a neighbor processor. Because, the parent cells on the paths from c_b back to the target or source cell will be unlabeled starting from a grandchild cell (of c_b) in the local

expansion or sweep queue. Hence, in order to avoid interprocessor communication during the sweeping phase and the extra communication during the front wave expansion phase, local boundary cells which can only be expanded to non-local adjacent cells at step 3 of the overlapped S+T front wave expansion algorithm are added into the sweep queue (see Fig. 3.18). The proposed scheme increases the total number of cells in the sweep queues hence increasing the total amount of sweeping computations compared to the sequential algorithm. However, this increase will decrease with increasing h, w values and increasing blockage factors.

The first algorithm for the parallel implementation of the *path recovery* and *sweeping* phases is given in the following subsection. This algorithm starts *sweeping* phase after the completion of *path recovery* phase.

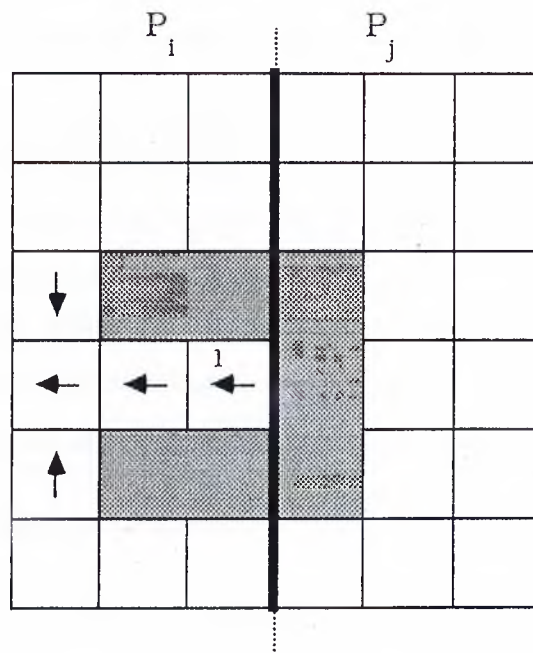
3.6.1 Non-pipelined scheme

In this scheme [13], the sweeping phase starts after the completion of path recovery phase. The host and node programs for the non-pipelined scheme are given in Figures 3.19 and 3.20.

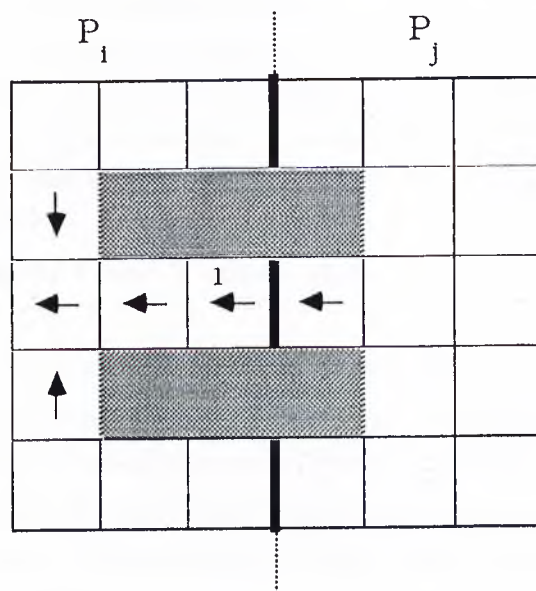
Since the path recovery phase is highly sequential by nature, most of the processors wait idle during the path recovery phase. In order to reduce this idle time, an efficient algorithm is proposed in the next subsection. In this scheme path recovery and sweep phases are pipelined in a way that processors which do not perform path recovery, can initiate the sweeping of some of the cells in its expansion and sweep queues [13].

3.6.2 Pipelined Scheme

Assume that the path from source to target is found in p cycles during the front wave expansion phase (assuming *Sonly* scheme is used for front wave expansion phase). Also assume that the cell Q is reached after q cycles of path recovery phase as is illustrated in Fig. 3.21.



(a)



(b)

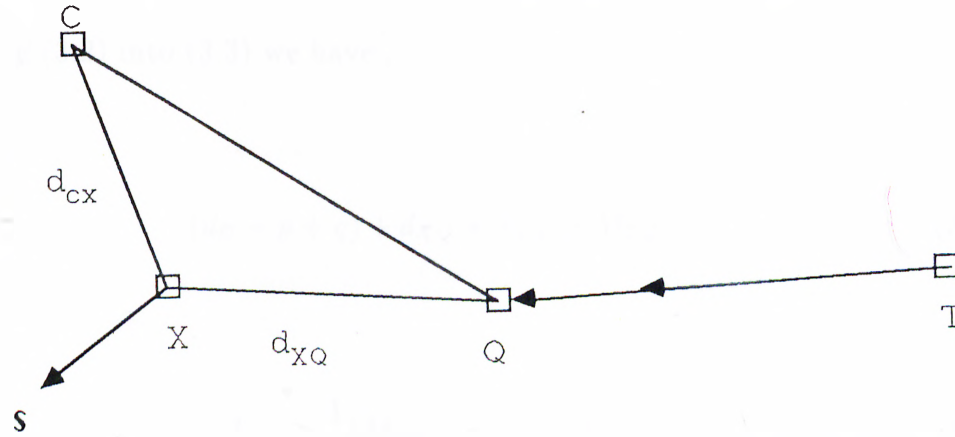
Figure 3.18. (a) Cell 1 is added into the sweep queue without any extra communication (b) Cell 1 is added into the sweep queue in parallel algorithm while it is not in sequential algorithm

-
1. Start path recovery phase by sending C_1 and C_2 to the processor(s) which own(s) these cells.
 2. Wait for source and target reached signal from nodes.
 3. Broadcast start sweep signal to nodes.
 4. Wait for P sweep terminated signals from nodes.
 5. Terminate the distributed sweep phase.
-

Figure 3.19. Host program for the non-pipelined path recovery and sweep phase

-
1. Wait for a path recovery cell C .
 2. Follow the labels starting from cell C until source, or target, or a non-local boundary cell is reached, and label visited cells as blocked. Inform the host if source or target cell is reached. If a non-local boundary cell is reached, send this cell to the neighbor processor which owns it.
 3. Repeat steps 1 and 2 until a start sweep phase signal is received from host.
 4. Remove a cell c from the local expansion or sweep queue.
 5. Follow the labels starting from c until a blocked, or an unlabeled, or a non-local boundary cell is reached. Unlabel visited cells.
 6. Repeat steps 4 and 5 until both sweep and expansion queues become empty. Then inform the host about the termination of local sweeping phase.
-

Figure 3.20. Node program for non-pipelined path recovery and sweeping scheme.

Figure 3.21. The calculation of r value.

Let cell C be a cell with depth d_C (from source) in the local expansion queue or in the sweep queue constructed during the front wave expansion phase, in the worst case the expansion path from C to source joins the *shortest path* from target to source at a cell X as is illustrated in Fig. 3.21. Then, the depth of the path from X to source, d_X , is

$$d_X = d_C - d_{CX} = d_Q - d_{XQ} \quad (3.1)$$

where d_{CX} and d_{XQ} are the depths of the paths from C and Q to X respectively. And $d_Q = p - q$ is the depth of Q to source. Hence,

$$d_{CX} = (d_C - p + q) + d_{XQ} \quad (3.2)$$

However,

$$d_{CX} + d_{XQ} \geq M_{CQ} \quad (3.3)$$

where M_{CQ} is the manhattan distance between C and Q .

Inserting (3.2) into (3.3) we have ,

$$(d_C - p + q) + d_{XQ} + d_{XQ} \geq M_{CQ} \quad (3.4)$$

$$d_{XQ} \geq \frac{1}{2}(M_{CQ} + p - q - d_C) \quad (3.5)$$

Inserting (3.5) into (3.2)

$$d_{CX} \geq d_C - p + q + \frac{1}{2}(M_{CQ} + p - q - d_C) \quad (3.6)$$

$$d_{CX} \geq \frac{1}{2}(M_{CQ} + d_C - p + q) \quad (3.7)$$

Hence,

$$r = \frac{1}{2}(M_{CQ} + d_C - p + q) \quad (3.8)$$

sweeping cycles can be performed for a cell C in the expansion or sweep queue at that instance of path recovery phase. After r cycles of sweeping is done, the new cell is added into the sweep queue with its new depth to source.

In the *pipelined scheme*, local sweep and expansion queues of each processor should have depth information. This depth information is used to show the

-
1. If a cell is received for path recovery, perform path recovery until source or target or boundary is reached.
 2. If boundary is reached during path recovery then send the boundary cell to corresponding neighbor processor. If boundary cell is tagged from target, set $Q_t =$ boundary cell, otherwise set $Q_s =$ boundary cell. Broadcast Q_s or Q_t .
 3. If source or target is reached during path recovery inform host.
 4. If not performing path recovery then remove a cell C from local expansion queue or local sweep queue if the local expansion queue is empty.
 - (a) Calculate the r value for the cell C .
 - (b) If $r \leq 0$ then put the cell back into the sweep queue.
 - (c) If $r > 0$ then perform sweeping until r becomes 0 or a blocked cell or an unlabeled cell or a boundary cell is reached.
 - (d) If $r = 0$ is reached, add the new cell into sweep queue with the new depth information.
 5. If new Q_s or Q_t received replace the old ones.
 6. If end of path recovery signal is received from host then goto step 7 else goto step 1.
 7. Perform the non-pipelined sweep algorithm.
-

Figure 3.22. Node program for the pipelined path recovery and sweep phase.

distance of a cell from source or target, according to its tag if $S+T$ scheme is used in front wave expansion phase. This depth information is required to calculate r , the number of sweeping cycles that can be performed for a cell. If $S+T$ scheme is used in front wave expansion phase, then there are two front waves. Hence, the path recovery is performed in two directions, one towards source and other towards target. Therefore, there are two Q cells, Q_s and Q_t , reached after q_s and q_t cycles of path recovery phase towards source and target, respectively. For each cell, r sweeping cycles is calculated using the corresponding Q cell in Eq. (3.8). The host program for the pipelined scheme is the same as the host program given for the non-pipelined scheme. The node program for the pipelined scheme is given in Fig. 3.22

3.7 Experimental Results

The proposed parallel algorithms has been coded in C and run on an iPSC/2 hypercube multicomputer with 8 nodes. Intel's iPSC/2 has one host processor, to perform the user interface tasks, down-loading and off-loading. Host processor in iPSC/2 contains 80386/80387, 64Kbyte cache, and 8 Mbytes of main memory. Each node processor contains 80386/80387, 64KByte cache, and 4 Mbytes of main memory.

The performance of the proposed algorithms have been experimented with randomly generated partial grids. These partial grids were generated by introducing blockages to represent existing interconnections. The blockages represented 40-45 % of the grid. All grids used for experimentation are $N \times N$ square grids of sizes 256x256, 512x512, and 1024x1024. The nets that were used for routing were also chosen randomly with the approximate length of N . The execution times represented in the figures represent the average execution time to route 4 different nets for each grid.

The abbreviations in the figures stand for the following :

S + T	Expansion starting from source and target
S	: Expansion starting from source only
M	Communication with host after manhattan distance
O	Overlapping the communication and computation
NO	Non-overlapped scheme
AS	Asynchronous scheme
NONPIP	Non-pipelined sweep
PIP	Pipelined sweep

First, the performance of the parallel front wave expansion algorithms are experimentally measured and compared.

Fig.3.23 was obtained by running synchronous algorithms for 1024x1024 grid on 4 processors. The effect of the h,w values on the performance of the algorithms was tested by varying h,w values. As is seen in Fig. 3.23, the computation time decreases at the beginning due to the decrease in the volume of

communication with increasing w values. After a minimum point, the computation time starts to increase. This is due to the increase in the processor idle time and deterioration in the load balance.

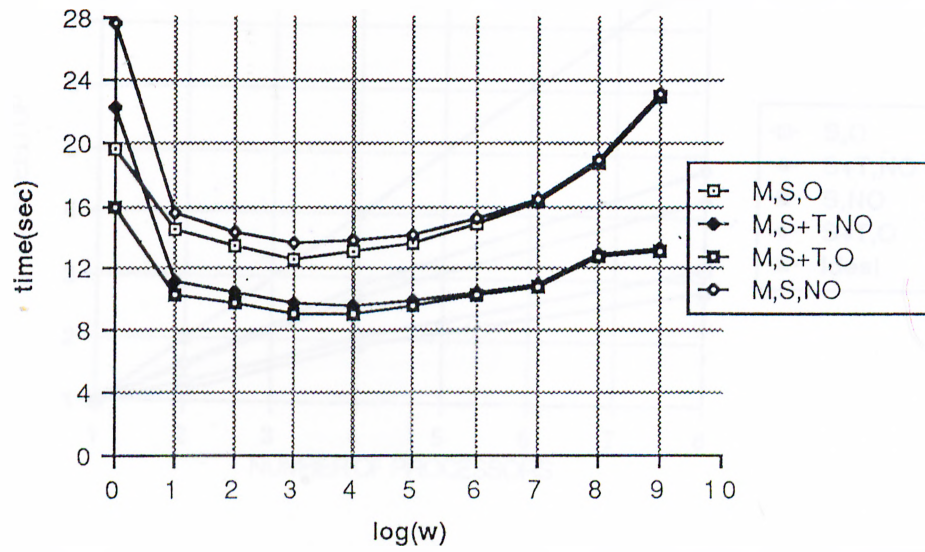
Fig. 3.24 represents the speed-up curves for different parallel schemes obtained on 1024×1024 grids by running the parallel algorithm with optimum h, w values. As is seen in the figure, S+T scheme outperforms the Sonly scheme, due to the increase in the processor utilization in the S+T scheme. The overlapped scheme performs better compared to non-overlapped scheme due to the reasons discussed in Section 3.4.

Figures 3.25 and 3.26 were obtained by running the overlapped S+T scheme with optimum w values on 256×256 , 512×512 and 1024×1024 grids. As is seen in Fig. 3.25, speed-up increases with increasing grid size and increasing number of processors. Note that, in Fig. 3.26 efficiency remains almost constant when both the grid dimension and number of processors are doubled. Hence, it can be concluded that the *S+T* scheme scales onto the hypercube.

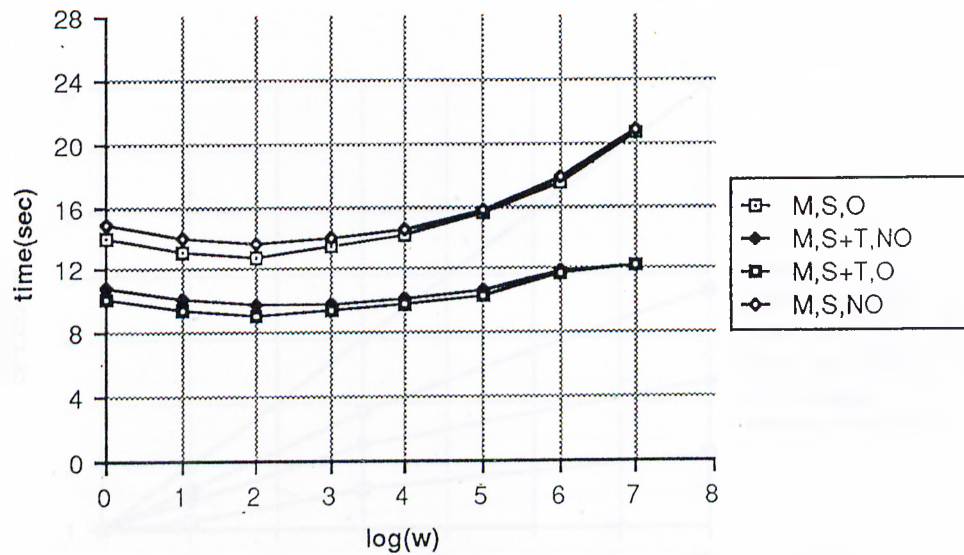
The performance of the asynchronous scheme is very low compared to the synchronous schemes as illustrated in Fig. 3.27. This is due to the computational overhead introduced in the expansion of an individual cell and the need to examine all the cells in the local expansion queue before terminating the front wave expansion phase.

After measuring the performance of the parallel algorithms for front wave expansion phase, the path recovery and sweep algorithms were applied to the grids used in front wave expansion phase. For this purpose, the *overlapped S+T* algorithm was run on the grids, then parallel path recovery and sweep algorithms were employed to construct the path between two terminal pins and sweep the grid surface. The timing results are the average of the time to perform path recovery and sweep for 4 two-pin nets.

Fig. 3.28 represents the effects of h, w values on the performance of the path recovery phase. The timings were obtained by running pipelined and non-pipelined algorithms for 256×256 grid on 8 processors. As is seen in Fig. 3.28, the computation time of the path recovery phase decreases with increasing w



(a)



(b)

Figure 3.23. Effect of w values on the performance of the parallel algorithm for $N = 1024$, $P = 4$. (a) $h = w$ (b) $h = 2w$

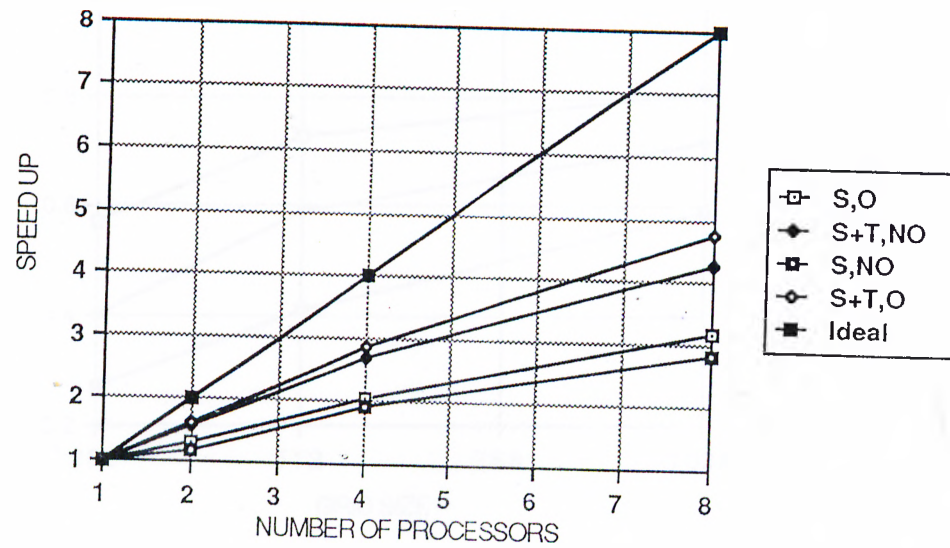


Figure 3.24. Speed-up for various parallel algorithms for front wave expansion phase

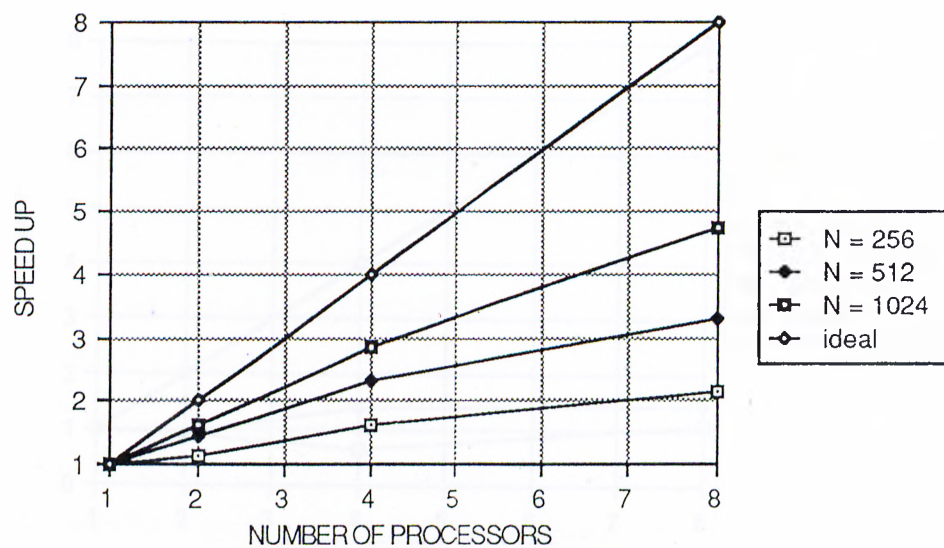


Figure 3.25. Speed-up vs grid size

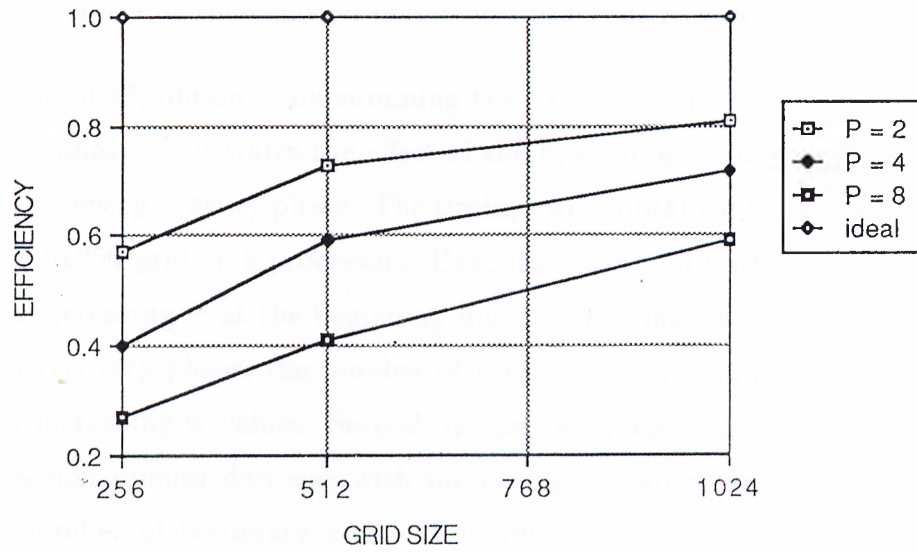


Figure 3.26. Efficiency vs grid size

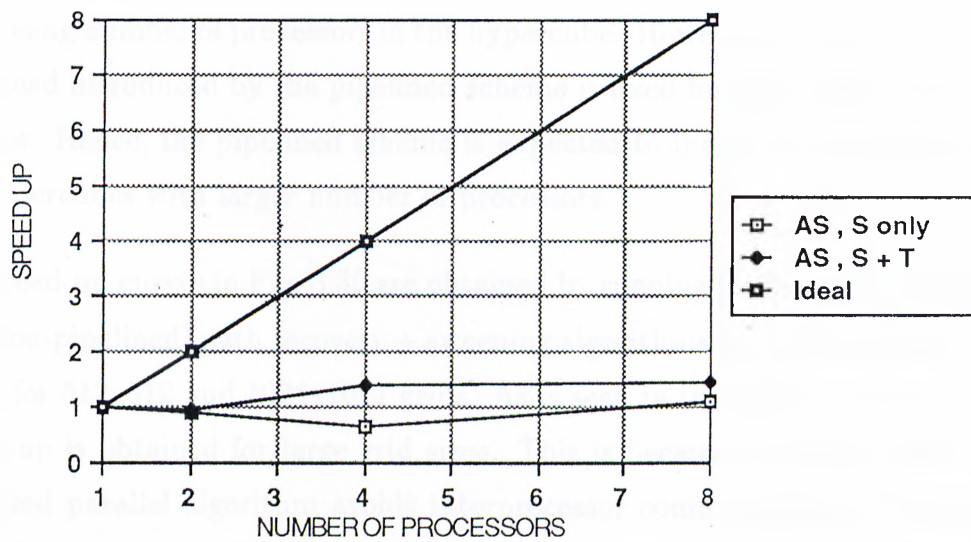
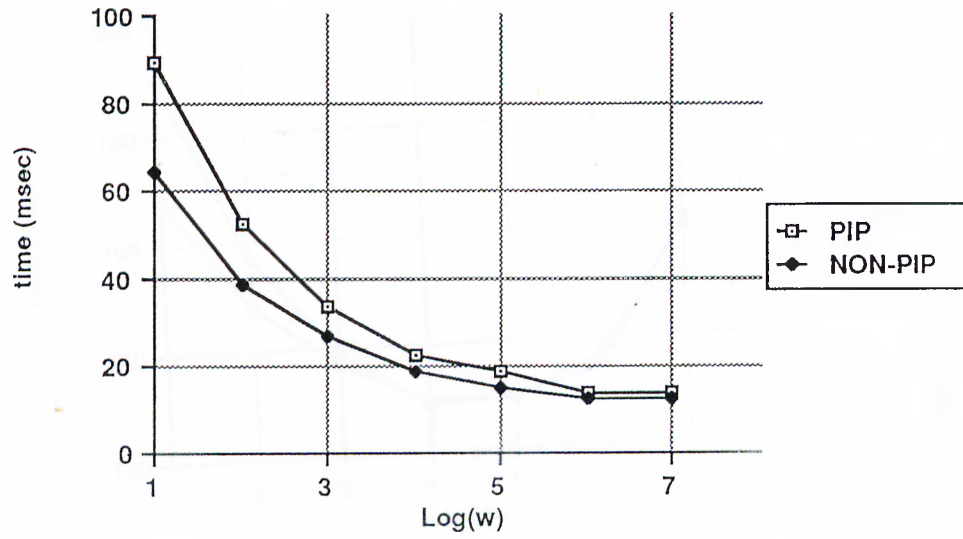


Figure 3.27. Speed-up figures for asynchronous algorithms

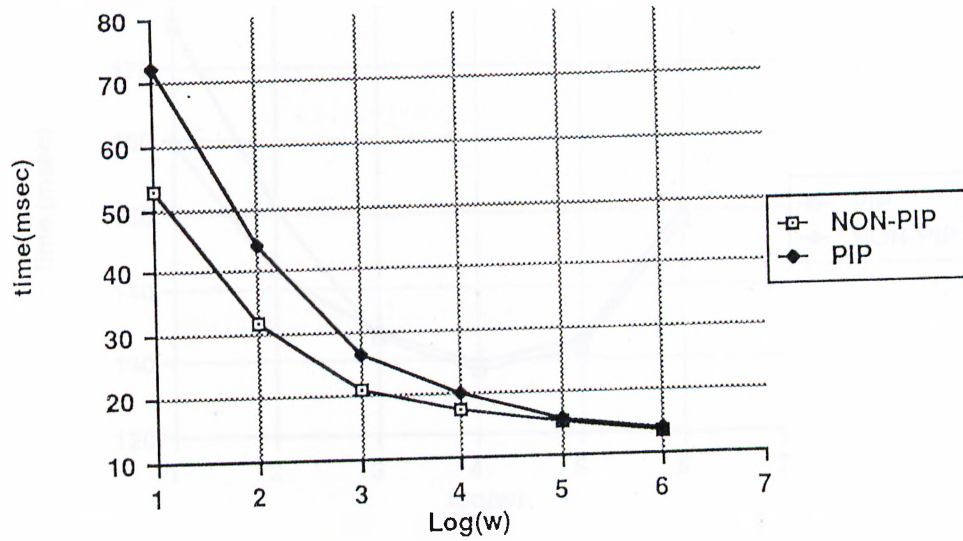
values due to the decrease in the number of interprocessor communications. Since the $S+T$ scheme was used in front wave expansion phase, the path recovery starts at the collision points and proceeds towards source and target.

Fig. 3.29, obtained by summing the execution times of path recovery and sweep phases, illustrates the effect of the h, w values on the performance of the *path recovery + sweep* phase. The timings were obtained by running algorithms for 256x256 grid on 8 processors. Execution times of both algorithms decrease with increasing w at the beginning due to following two reasons. First, in the path recovery phase, the number of interprocessor communications decreases with increasing w values. Second, in the sweeping phase, the size of the local expansion queues decrease with increasing w values due to the decrease in the number of boundary cells. However, efficiency begins to decrease after a turn over value for w in each scheme due to the deterioration in the load balance during the front wave expansion phase. Note that, deterioration in the load balance during the front wave expansion phase causes deterioration in the load balance during the sweep phase. That is, different processors may have very different number of grid cells to unlabeled during the sweep phase. As is also seen in Fig. 3.29, the pipelined scheme degrades the performance slightly due to the computational overhead involved in the calculation of r values. The percent processor idle time in the non-pipelined scheme will increase with increasing number of processors in the hypercube. However, the computational overhead introduced by the pipelined scheme is fixed for each cell in the local queues. Hence, the pipelined scheme is expected to increase the performance on hypercubes with larger number of processors.

Speed-up curves in Fig. 3.30 are obtained by running the parallel (pipelined and non-pipelined) path recovery + sweeping algorithms using the optimal h, w value for 512x512 and 1024x1024 grids. As is seen in this figure, almost linear speed-up is obtained for large grid sizes. This is because of the fact that the proposed parallel algorithm avoids interprocessor communication. Deviation from the ideal speed-up curve is due to the communication overhead in path recovery and load imbalance introduced in the front wave expansion phase.

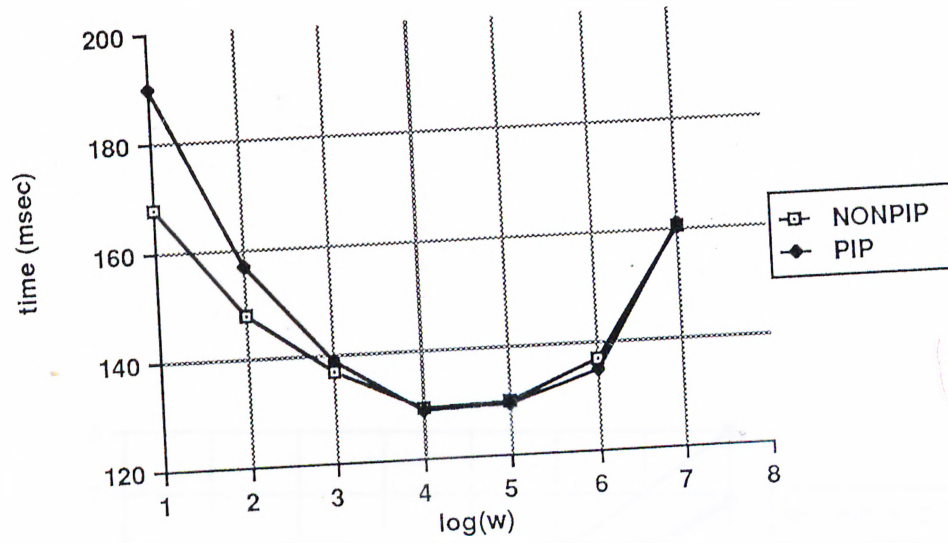


(a)

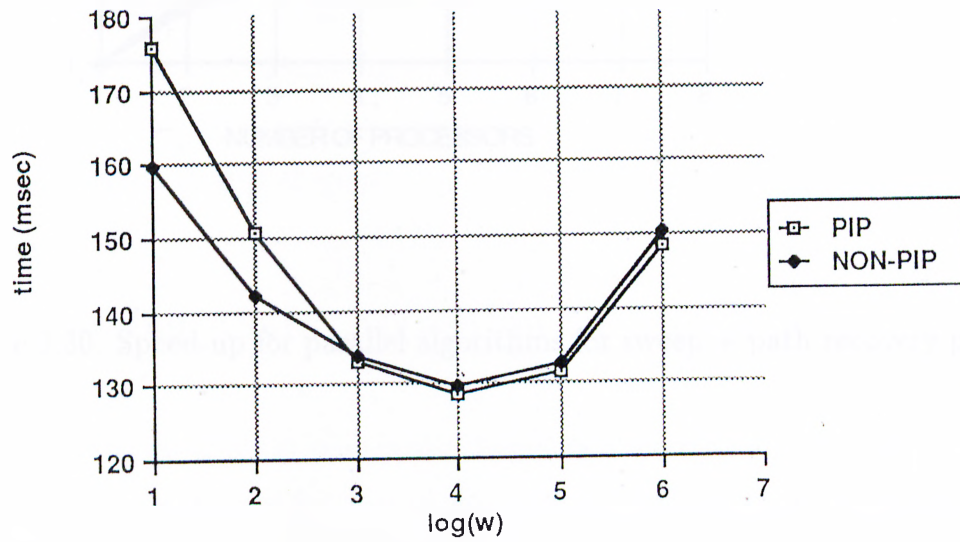


(b)

Figure 3.28. Effect of w values on the performance of path recovery (a) $h = w$
 (b) $h = 2w$



(a)



(b)

Figure 3.29. Effect of w values on the performance of path recovery + sweep
 (a) $h = w$ (b) $h = 2w$

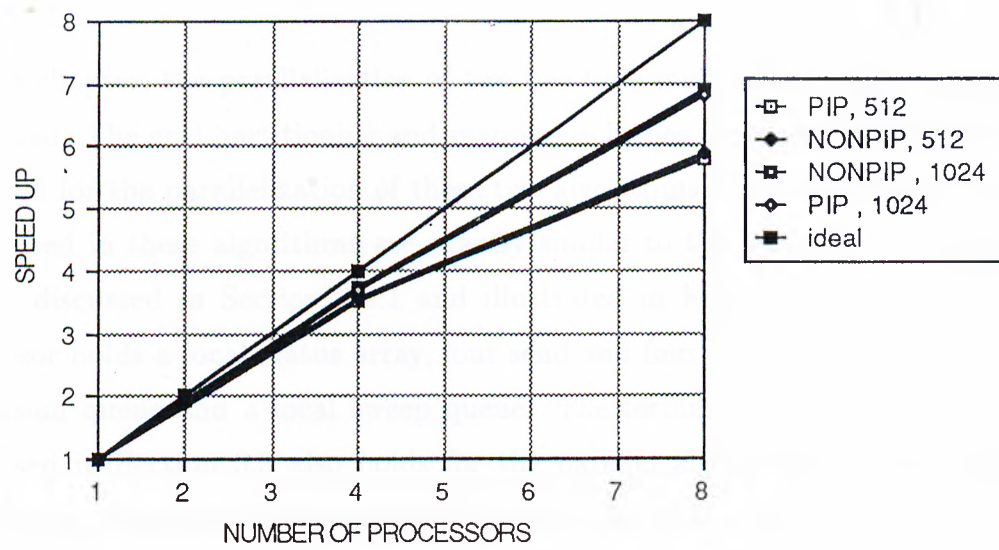


Figure 3.30. Speed-up for parallel algorithms for sweep + path recovery phase

4. PARALLEL ALGORITHMS FOR MULTIPIN NETS

In this chapter, the parallelization of two Lee type multipin net algorithms is discussed. The grid partitioning and mapping schemes proposed in Section 3.1 are used for the parallelization of these two algorithms. The local data structures used in these algorithms are exactly similar to the local data structure design discussed in Section 3.2.1 and illustrated in Fig. 3.5. That is, each processor holds a local status array, four send and four receive queues, a local expansion queue and a local sweep queue. The termination detection issues discussed in Section 3.3 also holds for the parallel algorithms in this chapter. Hence, these details are excluded in the presentation of the two parallel algorithms given in this chapter.

4.1 Parallel Akers' Algorithm for Multipin Nets

In this algorithm, a pin is arbitrarily chosen as a starting pin (starting source cell), for a unidirectional front wave expansion. The host program distributes the pins to the corresponding node processors. An arbitrary pin is chosen to be source, other pins are labeled as targets. The node that has the source starts to propagate a single ended front wave. The host program plays the role of managing the termination detection and synchronization between node processors during front wave expansion phase, path recovery phase and sweep phase as is discussed in Chap. 3. The parallel algorithm for node program is given in Fig. 4.1.

-
1. Each processor examines the cells in its local expansion queue for expansion in four directions. The local adjacent cells of the cells being expanded are examined for adding them to the local expansion queue for later expansion. The adjacent cells which are detected to belong to grid partitions assigned to neighbor processors are added to the corresponding *send* queues for later communication.
 2. Each processor transmits the information in its four *send* queues to their destination processors.
 3. Each processor examines the cells in its four *receive* queues for adding them to its local queue for later expansion.
 4. If one of the terminal cells is labeled then inform the host and other processors to start the parallel path recovery and sweep phases.
 5. Each processor adds the local visited cells during the path recovery phase into its local expansion queue as a new set of sources.
 6. Repeat steps 1,2,3,4,5 until all the terminal cells are labeled.
-

Figure 4.1. Node program for Akers' algorithm

At step 4 of the algorithm, if one unlabeled terminal cell is reached, the processor that detects the situation informs the host and other processors. Then, processors start path recovery. In path recovery phase, the visited cells are saved as the set of new local sources of each processor. At the end of sweeping phase, each processor adds these local sources into local expansion queue to start new search waves from these sources. Note that, all local expansion queues are flushed at the end of the sweep phase.

Note that, the first step (finding the first interconnection) of the Akers' algorithm is very similar to the *Only* version of the parallel Lee's algorithm discussed in Section 3.2. However, in the following steps Akers' algorithm initiates multi unidirectional search waves starting from the cells on the last connection path constructed. It is very likely that new set of sources will be distributed among a number of processors. Hence, a number of processors will be busy with the computations associated with the expansions of these multi-search waves even during the initial expansion cycles. These multi-search waves will reach the local grids of the idle processors very quickly thus increasing the overall processor utilization. Hence, parallel Akers' algorithm is expected to

yield better performance results compared to parallel two-pin net algorithms.

4.2 Parallel Kruskal's Steiner Tree Algorithm

In these scheme, each entry in the local status array is a two byte word to keep cell information. First byte is used to encode the labeling and spatial information of a cell, as is discussed in Chap. 3. The second byte is used to keep the tag of the terminal cell from which that grid cell is reached by front wave originating from that terminal cell. The tagging of the terminal cells is performed by numbering the cells starting from 0 to $T - 1$, if T terminal cells exist. For example, if the number in the byte is equal to 5, that means that grid cell is labeled by front wave originating from terminal cell tagged as 5. In a single byte, one can encode up to 256 cell tags, which is a reasonable number for gate array routing. This byte in the two-byte status word is also used for detecting the collision of two front waves belonging to different trees and for detecting to which tree the labeled cell belongs. This is done in the following way. Each node and host processor hold a one dimensional array of size T ($TreeArray[0...T-1]$). Each element of this array, which is accessed by indexing using the tag of the terminal cell, holds the tag of the tree (a number which is unique for each tree) to which the terminal cell belongs. This means that the front wave propagated from this terminal cell and grid cells labeled by this front wave also belongs to the same tree. For example, $TreeArray[5]$ holds the tag of the tree that the terminal cell 5 belongs to. Initially, since each terminal cell forms a distinct tree, each location in the tree array has a distinct number. When two trees are merged, this array is updated so that all terminal cells belonging to the same tree has the same number, which is unique for that tree, in the corresponding locations of the $TreeArray$. Assume that the tag byte of a local grid cell c is 5. It means that the cell c has been labeled originally by the front wave originated from terminal cell 5. However, it also means that the cell c belongs to tree tagged as $TreeArray[5]$.

At the beginning, host processor distributes the pins to corresponding processors. The processors, that receive these pin(s), put them into their local expansion queues. Then processors which have non-empty local expansion

-
1. Send the pins to corresponding nodes.
 2. Send enable signal.
 3. Wait for message from nodes.
 - If path found signal received then
 - Begin**
 - Receive c_1 and c_2
 - If c_1 and c_2 are from different trees then
 - Begin**
 - Record c_1 and c_2 to be used in path recovery.
 - Join two Trees and update Tree information.
 - Send updated tree information to nodes.
 - End.**
 - End.**
 5. If all pins are connected then
 - Begin**
 - Broadcast start path recovery signal and pairs of collision cells to nodes.
 - Wait end of path recovery signals from nodes.
 - Broadcast start sweep signal.
 - Wait end of sweep signals from nodes
 - Terminate the program.
 - End.**
 6. Goto step 3.
-

Figure 4.2. Host program for parallel algorithm for multipin nets using Kruskal's algorithm

queues, start a breadth first search expanding from these pins. The host and node algorithms are given in Figures 4.2 and 4.3.

The host and node processors hold the information about the currently constructed trees. The host program is assigned as a manager to keep track of the currently constructed tree, since more than one processor may detect intersection of different or the same trees at the same expansion cycle. The host processor is chosen as a master in order to prevent different processors constructing different local tree structures depending on their local decision. If one processor detects intersection of two trees, it informs the host processor and continues the execution. The host processor compares the new intersection with the tree information. If the intersection belongs to two different trees then it updates the tree array (TreeArray) and broadcasts the TreeArray to node processors. This avoids the further intersection of the same trees. However,

-
1. Each processor examines each cell in its local queue for expansion in four directions. The local adjacent cells of the cell being expanded are examined for adding into the local queue for later expansion. The unlabeled cells are labeled to point to the parent cell and the tree information of the cell in the status array is updated so that the adjacent cell a_c belongs to the same tree as its parent cell. If adjacent cell a_c is labeled and belongs to the same tree as its parent, then adjacent cell is ignored. If a_c is labeled and belongs to a different tree, that indicates the collision of two trees, the colliding cells are sent to the host processor along with the tree information of each cell. The adjacent cells belonging to the grid partitions assigned to the neighbor processors are put into the corresponding send queues along with the tree information of each cell.
 2. Each processor sends the four send queues to neighbor processors.
 3. Each processor examines the cells in its receive queues and add them into the local queue for later expansion.
 4. If new tree information is received then update the old tree information.
 5. Repeat steps 1,2,3,4 until start path recovery signal is received.
 6. Receive pairs of collision cells to start path recovery.
 7. Send end of path recovery signal to host.
 8. Perform sweep phase.
 9. Send end of sweep signal to host.
-

Figure 4.3. Node program for the parallel algorithm for multipin nets using Kruskal's algorithm

since node processors run in an asynchronous manner, the intersection of the same trees may be detected by more than one processor. The host program, in that case, only takes the first received intersection as valid and discards the others. The node processor detects the intersection of trees using the latest structure. Note that, the tree information has to be associated with the cells sent to the neighbor processors. Hence, each entry in the send and receive queues has an extra byte to be used in the same way as the second byte of the status word.

After all trees are merged, the host program sends the start of path recover signal and the pairs collision cells (collision points) to corresponding nodes. The nodes that receive those cells start path recovery phase. When all edges of the tree is constructed, the nodes start the distributed sweep phase.

The sequential Kruskal's Steiner tree algorithm requires that all search waves propagate at the same speed. Hence, the queues of different search waves are merged. However, as explained in Section 3.2.2. some processors may be lagging some others may be leading. This requires that the first time a processor detects an intersection, that intersection may not be the shortest path between two trees. Hence, a synchronization like the one proposed in Section 3.2.2 should be used. However, such a synchronization will obviously decrease the performance of the algorithm. On the other hand, the Kruskal's Steiner tree algorithm does not guarantee to find the optimal solution. That is, the cost of the solution found by this algorithm is bounded by some error (See Eq. (2.1)). In the parallel algorithms implemented in this work, the first tree intersection received by the host processor is assumed to be the shortest path between these two different trees. That is, the protocol in Section 3.2.2 is not implemented to wait for a couple of more expansion cycle to find a slightly better tree intersection.

4.3 Experimental Results

The algorithms were coded in C language and run on iPSC/2 hypercube multicomputer with 8 processors. The algorithms were tested on square grids of

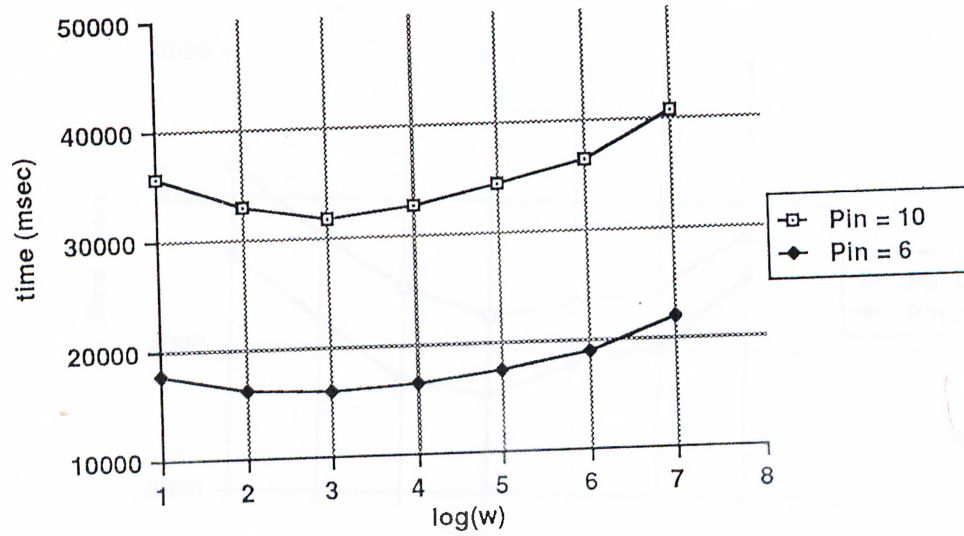
sizes 512x512 and 1024x1024. Randomly generated blockages were introduced to represent the previously routed nets. The blockages constituted the 40-45% of the grid area. The algorithms were tested to route 4, 6, 7, 10 pin nets. For this purpose, four 10 pin nets were generated randomly for each grid size. The length of the nets are chosen to be proportional to the grid size. The 4, 6, 7 pin nets were constructed by randomly choosing 4, 6, 7 pins out of the pins in the 10-pin net. The timing results obtained are the average execution times (frontwave expansion + path recovery + sweep) for routing four different nets for each grid size and for each number of pins.

Fig. 4.4 represents the variation of the execution time of the parallel Akers' algorithm with h,w values. This figure is obtained by running Akers' algorithm for a 512x512 grid on 4 processors for 10 and 6 pin nets. The time decreases at the beginning due to the decrease in the communication overhead with increasing h,w values. After a turnover value it starts to increase due to the increase in the processor idle time. Note that the curves obtained for 10 pin net is similar to the one for 6 pin net.

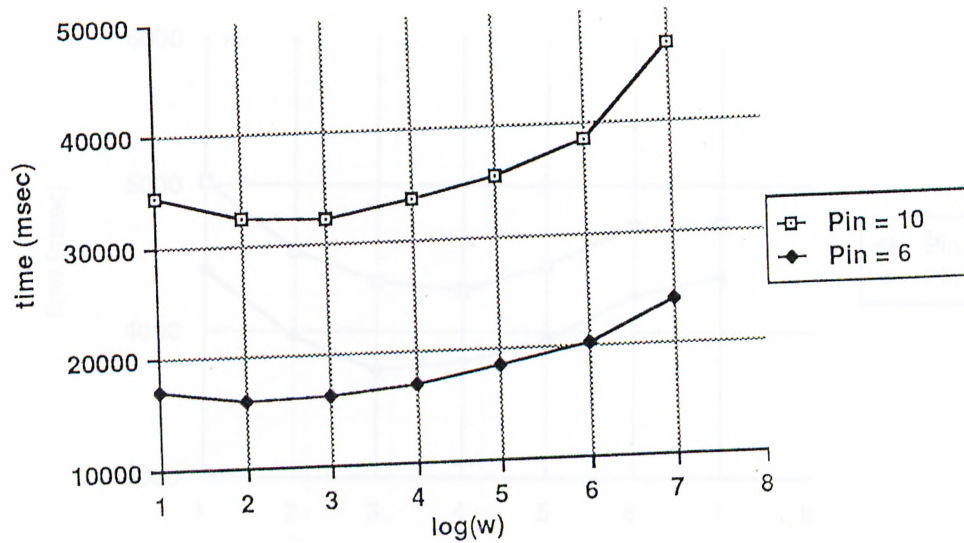
Fig. 4.5 represents the variation of the execution time of the parallel Kruskal's Steiner tree algorithm with h,w values. This figure is obtained by running Kruskal's Steiner tree algorithm for a 512x512 grid on 4 processors for 10 and 6 pin nets. The time decreases at the beginning due to the decrease in the communication overhead with increasing h,w values. After a turnover value it starts to increase due to the increase in the processor idle time. Note that the curves obtained for 10 pin net is similar to the one for 6 pin net.

Figures 4.6 and 4.7 represents the speed-up figures for Akers' algorithm. These figures were obtained by running Akers' algorithm for 512x512 and 1024x1024 grids at optimum h,w values for 4, 7, 10 pin nets. As is seen in the figures the speed-up increases with increasing number of pins and increasing number of processors. Also, when the grid size increases, the speed-up also increases.

Figures 4.8 and 4.9 represents the speed-up figures for Kruskal's Steiner tree algorithm. These figures were obtained by running Kruskal's Steiner tree algorithm for 512x512 and 1024x1024 grids at optimum h,w values for 4, 7, 10

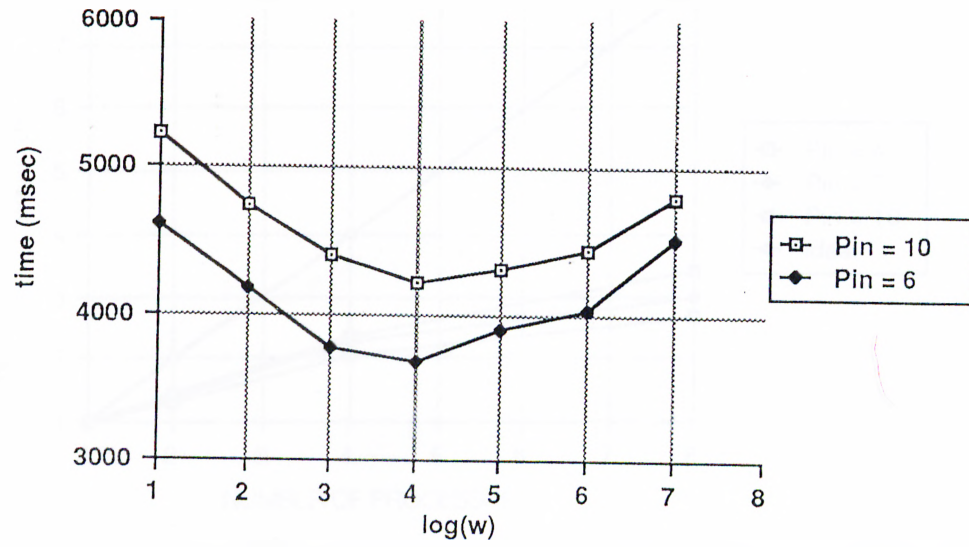


(a)

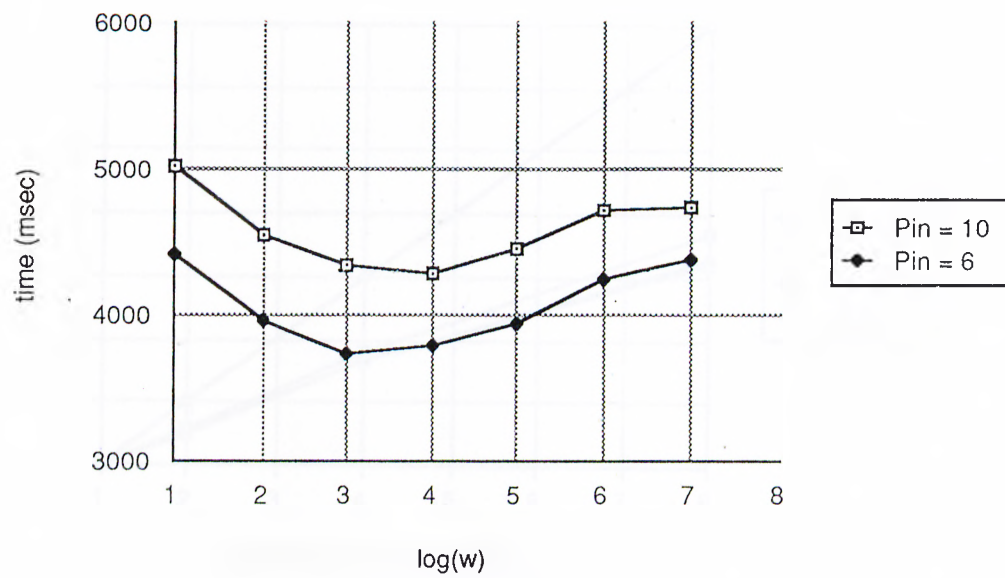


(b)

Figure 4.4. Effect of h, w values on the execution of parallel Akers' algorithm
(a) $h = w$ (b) $h = 2w$



(a)



(b)

Figure 4.5. Effect of h, w values on the execution of parallel Kruskal's Steiner tree algorithm (a) $h = w$ (b) $h = 2w$

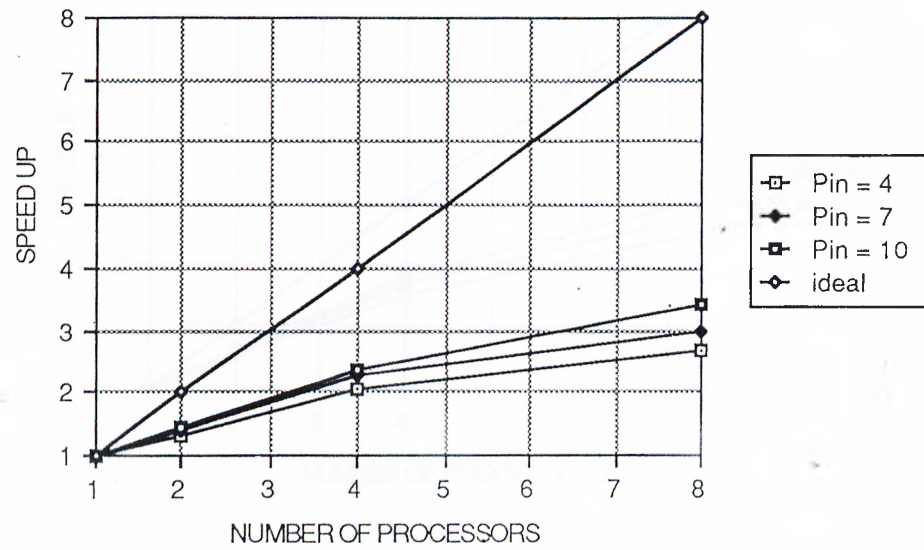


Figure 4.6. Speed-up figure for parallel Akers' algorithm on a 512x512 grid for 4, 7, 10 pin nets

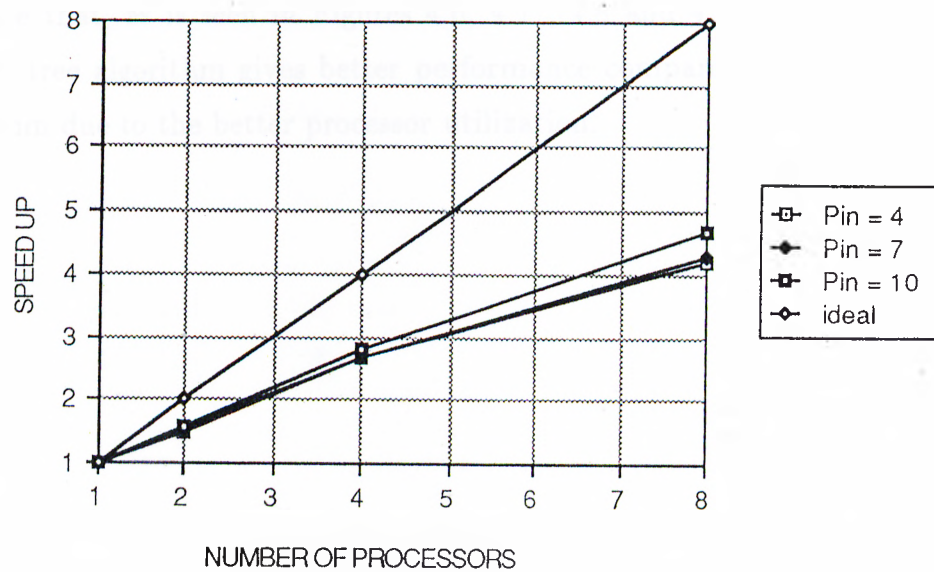


Figure 4.7. Speed-up figure for parallel Akers' algorithm on a 1024x1024 grid for 4, 7, 10 pin nets

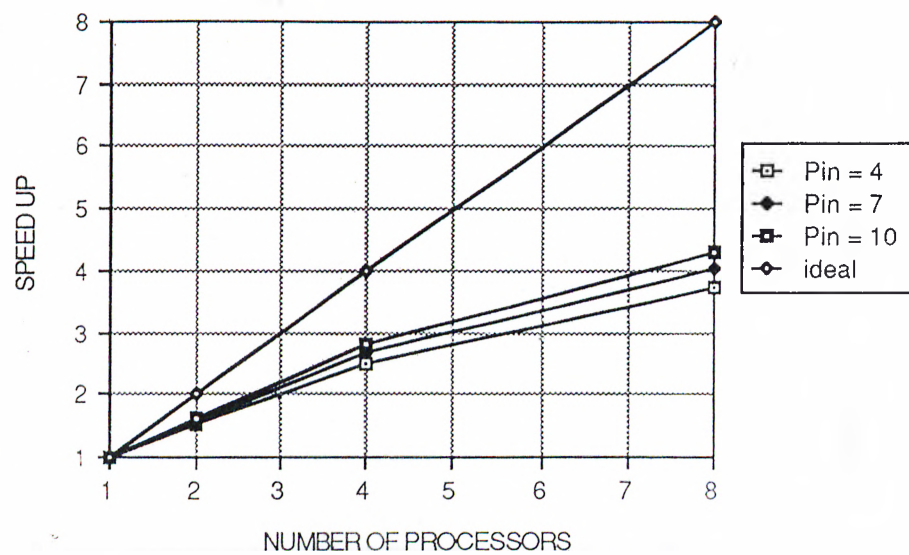


Figure 4.8. Speed-up figure for parallel Kruskal's Steiner tree algorithm on a 512x512 grid for 4, 7, 10 pin nets

pin nets. As is seen in the figures the speed-up increases with increasing number of pins and increasing number of processors. Also, when the grid size increases, the speed-up also increases.

Note that, as is seen in Figures 4.6, 4.7, 4.8 and 4.9, parallel Kruskal's Steiner tree algorithm gives better performance compared to parallel Akers' algorithm due to the better processor utilization.

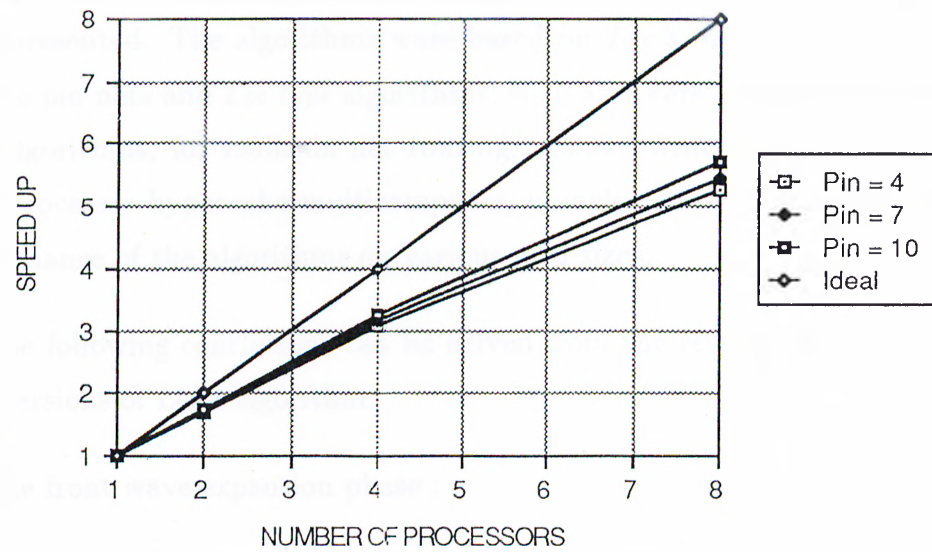


Figure 4.9. Speed-up figure for parallel Kruskal's Steiner tree algorithm on a 1024x1024 grid for 4, 7, 10 pin nets

5. CONCLUSIONS

In this thesis, parallel algorithms for maze routing process in VLSI routing were presented. The algorithms were based on *Lee's maze routing* algorithm for two pin nets and *Lee type* algorithms, such as Akers' and Kruskal's Steiner tree algorithms, for multipin net routing. Experiments were carried out on an 8 processor hypercube multicomputer, namely iPSC/2, for measuring the performance of the algorithms on various grid sizes.

The following conclusions can be driven from the results obtained for parallel versions of Lee's algorithm.

The front wave expansion phase :

- The number of cells expanded increases with increasing distance between source and target. This increases granularity and processor utilization. Hence, performance increases with increasing distance between source and target cells.
- Overlapping the communication and the computation scheme was also implemented.
- Expansion starting from source and target scheme gives better performance results compared to expansion starting from source only scheme. This is because the expansion starting from source and target scheme increases processor utilization and decreases the volume of interprocessor communication.

The path recovery and sweep phases :

- Performance increases with increasing distance between source and target cells, because the granularity and processor utilization increases with increasing distance.
- The non-pipelined sweep algorithm performs better than pipelined sweep algorithm due to the overhead involved in the calculation of r values. However, this overhead is constant and the pipelined sweep algorithm is expected to perform better for larger number of processors.
- Better speed-up figures are obtained in sweeping phase than in front wave expansion phase, since the proposed scheme avoids interprocessor communication in sweep phase.

In Lee's maze routing algorithms, expansion computations associated with an individual cell is a fine grain computation. Parallel versions of Lee's routing algorithm are communication bound for synchronization purposes. Furthermore, achieving perfect load balance is almost impossible due to the dynamic nature of the routing computations. Hence, it is very hard to achieve ideal speed-up.

For the multipin net algorithms, we can derive the following conclusions from the experimental results :

- Better performance figures are obtained with increasing length of the nets and increasing number of the pins due to the increase in processor utilization and granularity. The multi-source wave expansion increases the processor utilization compared to two-pin nets.
- Parallel Kruskal's Steiner tree algorithm performs better than parallel Akers' algorithm due to the increase in the processor utilization.

Bibliography

- [1] Thomas Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, B.G. Teubner, Stuttgart, John Wiley & Sons 1990.
- [2] Ernest E. Hollis, *Design of VLSI Gate Array ICs*, Prentice-Hall, Inc. Englewood Cliffs, NJ. 07632, 1986, Chap. 1, pp. 6 - 12.
- [3] Ravi Nair, "A Simple Yet Effective Technique for Global Wiring," *IEEE Trans. on CAD*, Vol. CAD-6, March 1987, pp. 165-172.
- [4] Carl Sechen, Alberto Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package." *IEEE Journal of Solid-State Circuits*, Vol. SC-20, April 1985, pp. 510-522.
- [5] Jeong-Tyng Li, Malgorzata Marek-Sadowska, "Global Routing for Gate Array," *IEEE Trans. on CAD*, Vol. CAD-3, Oct. 1984, pp. 298-307.
- [6] Benjamin S. Ting, Bou Nin Tien, "Routing Techniques for Gate Array," *IEEE Trans. on CAD*, Vol. CAD-2, Oct. 1983, pp. 301-312.
- [7] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electronic Computers*, Vol. EC-10, pp. 346-365, Sept. 1961.
- [8] Youngju Won and Sartaj Sahni, "Maze Routing On a Hypercube Multiprocessor Computer," *Proceedings of Intrnl. Conf. on Parallel Processing*, St.Charles August 1987, pp. 630-637.
- [9] Y. Won, S. Sahni, and Y. El-Ziq, "A hardware accelerator for maze routing," in *Proc. 24th Design Automation Conf.*, June 1987, pp.800-806.
- [10] J.S. Rose, "Parallel Global Routing for Standart Cells," *IEEE, CAD*, Vol.9, Oct. 1990, pp.1085-1095.

- [11] Y. Saad and M. Schultz, "Topological properties of hypercubes," *Research Report, YALEU/DCS/RR-389, Computer Science Dept., Yale University*, Jun. 1985.
- [12] T. M. Kurç, C. Aykanat, and F. Erçal, "Parallelization of Lee's Routing Algorithm on a Hypercube Multicomputer". in the Proceedings of *The Second European Distributed Memory Computing Conference*, Munich, Germany, April 22-24, 1991. pp. 244-253.
- [13] C. Aykanat, and Tahsin M. Kurç, "Efficient Parallel Maze Routing Algorithms on a Hypercube Multicomputer," Proc. of the 1991 Inter. Conf. on Parallel Processing, Vol.3, pp. 224-227, August 12-16,1991.
- [14] Frank Rubin, "The Lee path connection algorithm," *IEEE Trans. Comput.*, vol. c-23, pp.907-914, Sept.1974.
- [15] M. Hanan, "On Steiner's Problem with Rectilinear Distance," *J. SIAM Appl. Math.*, Vol. 14, No. 2, pp. 255-265, March 1966.
- [16] F.K. Hwang, "On Steiner Minimal Trees with Rectilinear Distance," *SIAM J. Appl. Math.*, Vol. 30, No. 1, pp. 104-114, Jan. 1976.
- [17] S.B. Akers, *Design Automation of Digital Systems; Theory and Techniques*, M.A. Breuer, Ed. Englewood Cliffs, NJ: Prentice-Hall,1972, chap. 6.