National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Election and Termination Detection in Specialized Communication Structures

By

G. H. Masapati

THESIS
SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES AND RESEARCH OF
THE UNIVERSITY OF OTTAWA
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
PH. D. DEGREE IN COMPUTER SCIENCE

The Ph. D. program in Computer Science
is jointly administrated with Carleton University by
Ottawa-Carleton Institute for Computer Science

ISBN 0-315-95957-6

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

This thesis addresses *election* and *termination detection* problems in specialized communication structures. Three communication structures; namely, composite complete network (i.e., a complete network whose size is composite), recursively scalable network and H-mesh, are examined.

The objective in considering composite complete networks is to investigate the effect on the message complexity of the election algorithms when N the size of the complete network is composite. Specifically, it is shown that when N is a multiple of 2, 3, 4, 5, or 6, the message complexity of the existing election algorithm for the complete network can be improved. To this end, an hybrid approach to designing election algorithms for complete networks with a *sense of direction* is developed when N is composite. The approach is based on a well-known *divide* and *conquer* paradigm for problem solving.

The objective in considering recursively scalable networks and H-meshes is to utilize the underlying communication structure as a guiding factor in the development of election and termination detection algorithms. Furthermore, these communication structures are recently analyzed and are employed as interprocessor communication architectures in the development of message passing multiprocessor systems.

An election algorithm in a synchronous recursively scalable network is designed by making use of its hierarchical communication structure. It requires less than $4.5N$ messages and the running time is at most $2.75\sqrt{N}$ where N is the number of processors in the network.

A termination detection algorithm in a H-mesh is developed. It is based on the propagation of passive state information and on the probe passing technique. Traffic rules for probes and passive state information are designed by making use of the structure of the H-mesh. The correctness proof for the termination detection algorithm is also presented.

Acknowledgements

I am grateful to my supervisor Professor Hasan Ural for his guidance, encouragement, support and kindness. It was a pleasure working with him.

I would like to express my deep sense of gratitude to Professor Nicola Santoro who spent considerable amount of his precious time guiding me through chapter 2 of this thesis. He was very kind and patient.

I wish to thank Professor Gerald Karam and Professor Dan Ionescu for their useful suggestions, comments and encouragement during my graduate study.

I am thankful to Professor Richard Tan and Professor Michael Loui for their useful comments and helpful suggestions on issues related to chapter 2 of this thesis. I also thank Professor H. T. Mouftah for a useful discussion on recursively scalable networks.

I am especially grateful to Professor Robert Probert, Professor Louis Birta, Professor George White and Professor Tuncer Ören for their generosity, help, kindness and support throughout my graduate studies.

My financial support during the course of my graduate studies from OGS, NSERC (through my supervisor's grant) and University of Ottawa Achievement award is gratefully acknowledged. I also thank the Computer Science department for providing the necessary facilities through the course of my graduate studies.

Finally I thank my friends and colleagues Nur Özmızrak and Murat Özmızrak for their encouragement and support during my graduate studies.

Contents

Chapter 1

Introduction and Literature Survey

1.1 Introduction

A *distributed system* is a collection of autonomous computing nodes which can communicate with each other and which cooperate on a common goal or task[49].

Distributed systems are often used to access remote shared resources. These systems, in addition to providing higher performance or throughput, also offer increased reliability or fault-tolerance. In such systems, when some of the computing nodes or communication links fail, the remaining connected part of the system can continue to operate perhaps with a degraded performance; however a component failure in a single computing node system is often disastrous.

From the perspective of modeling and analysis, a distributed system can be represented as an undirected graph, $G = (V, E)$ with $\mid V \mid = N$ where:

a) set V corresponds to the collection of autonomous computing nodes (also referred to as processors) and

b) set E corresponds to the set of communication links which enable the computing nodes to communicate with each other to carry out the task. For any $e = \{v_i, v_j\} \in E$ indicates that the processor $v_i \in V$ and $v_j \in V$ communicate using the communication link e.

Furthermore, each processor is assumed to have a local non-shared memory and clock. The communication between processors $v_i \in V$ and $v_j \in V$ connected by communication link $e \in E$ is characterized by sending messages and receiving messages along link e.

With respect to the issues addressed in this thesis, it would suffice to consider each processor $v_i \in V$ as an event-driven, finite-state entity. That is, informally, depending upon the occurance of a predefined *event* (such as a message is received, the clock reaches a predefined values, etc.) and depending upon its current state, each processor may perform operations such as *local computations*, *transmission of messages* and *change of state*.

Formal descriptions of distributed systems and the associated problems can be found in [17, 22, 23, 51, 124, 126].

Within the above context, one of the basic computations in the distributed system G is to *implement* the following specification:

Each processor $v_i \in V$ starts in the same initial state but exactly one processor $v^* \in V$ enters a predefined state called "Elected" and the rest of the processors $V - \{v^*\}$ enter a predefined state called "Defeated". There is no a-priori restriction as to which of the processors in V enters the "Elected" state.

This is referred to as "electing a leader" among a set of processors. The *leader election* occurs in token passing systems as follows. Token passing systems provide a

useful mechanism (called *mutual exclusion*) which ensures that at any time at most one processor out of a group of processors enters a critical section. The processors P_0 , P_1 , $\cdots$, P_{N-1} in a token passing system are arranged in a ring such that P_i receives messages from $P_{i\ominus 1}$ and sends messages to $P_{i\oplus 1}$ ($\oplus$ and $\ominus$ respectively are *modulo* N addition and subtraction). A single token circulates around the ring. A processor P_i wishing to enter its critical section waits until it receives the token from $P_{i\ominus 1}$. P_i then enters its critical section and it sends the token to $P_{i\oplus 1}$ upon completion of the execution of its critical section. A processor which does not wish to enter its critical section transmits the token upon receiving it.

A problem arises when the token is lost. In order to recover from this situation the processors elect a leader to issue a new token. Le Lann[87] suggested a solution for both token loss detection and token regeneration based on processor identities.

The leader election also occurs in clock synchronization when a new master among the slaves needs to be chosen after the crash of the machine on which the original master was running[63], or in a distributed database system to replace a malfunctioning central lock coordinator[102], or in replicated distributed file system to replace a primary site[6].

Of the states of each processor $v_i \in V$ in the distributed system G, one specific state is called "idle" state which is the state wherein v_i is *waiting* to receive messages (i.e., informally it has completed its assigned task). Furthermore, consider that each communication link $e = \{v_i, v_j\} \in E$ of the distributed system G is in one of two states; namely, "empty" (which indicates that there are no messages in e) or "non-empty" (which indicates that there are messages in e intended for either v_i or v_j). Within this framework, another important basic computation in the distributed

system G is to detect the following situation:

For every $v \in V$ state of v is "idle" and for every $e \in E$ state of e is "empty".

This is referred to as *termination detection*. The objective here is to determine whether the computation in G is terminated or not. For example, in distributed simulation scheme based on multiphase approach[27], the termination of a phase is required to be detected in a *distributed* manner by the processors before a next phase can be initiated. Also in an iterative solution of simultaneous equations in a distributed system[134], it is required to detect that every computing node is converged so that the entire computation can be terminated.

The complexity (or performance) of a computation in the distributed system G is measured in terms of three parameters; namely, the maximum number of messages sent during the computation, the maximum running time of the computation, and the size of the messages required in the computation.

An important component of the distributed system G is the set E which defines the interprocessor communication architecture. Several different interprocessor communication architectures; namely, ring, chordal ring, complete network, square mesh, recursively scalable network and H-mesh, are proposed and analyzed in the distributed systems literature.

The impact of interprocessor communication architecture on the basic computations such as leader election and termination detection is the subject of intensive research. In this thesis, we address leader election and termination detection in complete networks, recursively scalable networks and H-meshes.

1.2 Literature Survey

1.2.1 Election

Several election algorithms, both synchronous and asynchronous, have been designed and analyzed for various network interconnection schemes; namely, ring[33, 44, 58, 69, 85, 87, 106, 109, 111], chordal ring[15], complete network[3, 80, 89, 90, 112], and square mesh[2, 112]. In Table 1.1, we summarize the election algorithms and their worst case message complexities for different network interconnection schemes.

Burns[23], Frederickson and Lynch[56], and Pachl, Korach and Rotem[110] showed that election in synchronous rings requires $\Omega(N \log N)$ messages. Because synchronous rings are a special case of aynchronous rings, election algorithms for asynchronous rings also require $\Omega(N \log N)$ messages. Afek and Gafni[4] and Peterson[112] proved that election in synchronous and asynchronous complete networks requires $\Theta(N \log N)$ messages. Santoro[124] showed that the election in a general graph requires $\Theta(\mid E \mid + N \log N)$ messages where $\mid E \mid$ denotes the number of links in the graph. An excellant review of lower bound results for election algorithms can be found in [92].

Sense of direction[125] plays an important role in designing election algorithms. A complete network has a *sense of direction*[125] if there is a fixed directed Hamiltonian cycle such that at each processor, each link is labeled with the distance along this cycle to the processor at the other end of the link. Loui, Matsushita and West[89, 90] show that $O(N)$ messages suffice to elect the leader in an asynchronous complete network with a sense of direction. Peterson[112] designed an election algorithm in an

asynchrnous square mesh with a sense of direction which requires $O(N)$ messages.

In the case where the uniqueness of processor identity does not hold, the solutions to the election problem are discussed in [57, 86]

Dolev and Israeli[43] develop leader election protocols for uniform, dynamic and self-stabilizing distributed systems.

Processor and communication link failures present an additional challenge in devising solutions to the election problem. Several election algorithms under different failure modes are considered in [1, 2, 24, 35, 61, 75, 97].

An excellant discussion of the practical importance of elections in a distributed system is given by Le Lann[87] and Garcia-Molina[59]. To this end, several election protocols for general configuration networks in the absence of message loss[18, 59, 63, 77, 102] and in the presence of message loss and network partitioning[48, 78] during the election process are developed. [48] also carried out the performance analyis using simulation modeling techniques. [63] presented an election algorithm for TEMPO, a distributed network clock synchronizer for Berkeley UNIX 4.3BSD systems.

An interesting discussion of leader election is given in [128]. This takes the view that the leader should be elected based upon systems characteristics such as expected connectivity, end-to-end delays, workload and delay characteristics. The objective here is to elect the leader that is *available* (in terms of actual use) to as many nodes as possible.

Algorithm	Connection Topologies	Worst Case	
Dolev, Klawe, Rodeh[44]	Unidirectional Ring	$1.356N \log N$	
Peterson[111]	Unidirectional Ring	$1.44..N \log N$	
Moran, Shalom, Zaks[106]	Bidirectional Ring	$1.44...N \log N$	
Attiya, et. al.[15]	Chordal Ring Networks	$O(N)$	
Loui, Matshushita, West[89]	Complete Network	$3.62N$	
This Thesis — Chapter 2	Complete Network (N is composite, $N = dN'$)		d
		$3.310N$	2
		$3.540N$	3
		$3.155N$	4
		$3.324N$	5
		$3.270N$	6
Abu-Amara[2]	Synchronous Square Mesh	$\frac{229}{18}N$	
This Thesis — Chapter 3	Synchronous Recursively Scalable Network	$4.5N$	

Table 1.1: Election Algorithms and Their Worst Case Message Complexities

1.2.2 Termination Detection

Dijkstra and Scholten[41] proposed a very elegant termination detection algorithm for a specific class of computations, called *diffusing computations*. Since then it has been extended and applied to various problems[27, 28, 32, 47, 67, 104, 105, 121]. Based on the Dijkstra-Scholten scheme, Misra and Chandy presented distributed graph algorithms for knot detection[105] and for computing shortest paths from a single vertex to all other vertices in a weighted, directed graph, in the presence of negative cycles[28]. Misra and Chandy[104] also showed how to adopt the Dijkstra-Scholten scheme for termination detection in a diffusing computation, for detecting termination or deadlock in a network of communicating sequential processes as defined by Hoare[70]. Dijkstra-Scholten scheme is also used for deadlock detection in a distributed system[32, 47, 67] and for derivation of distributed enforcement algorithm[121]. The basic idea of Dijkstra and Scholten scheme is also utilized in the distributed simulation scheme developed by Chandy and Misra[27]. Natarajan[108] shows the connection between detecting termination and detecting communication deadlock.

Francez[52, 53, 54, 55] addressed the problem of distributed termination by augmenting the distributed computation with extra control communication. A rooted spanning tree of the communication graph is chosen which carries the control communication. The root of the tree generates control waves down the tree to check whether the processes are ready to terminate. When the wave reaches the leaf processes it starts going up collecting agreement (or disagreement) of processes to terminate. Misra[103] presented an algorithm which improves Francez's scheme. The algorithm

uses a single marker which repeatedly traverses the edges of the network until it detects termination. Cohen and Lehman[36] improved and extended Francez's scheme to dynamic systems.

Termination detection schemes of Dijkstra and Scholten[41] and Francez[52] rely on the existance of the presence of a special master processor in the network. This issue of asymmetry is addressed by several researchers [10, 11, 12, 14, 50, 64, 65, 71, 72, 113, 114, 116, 122, 129, 130, 136] who present symmetric (or uniform) distributed termination detection algorithms.

Tel et. al.[138, 140] show how to derive garbage collection algorithms from distributed termination protocols. Termination detection algorithm for Occam processes is presented in [135]. Termination detection of iterative solution of simultaneous equations in a distributed message passing system is presented in [133, 134]. Termination detection and abortion algorithms based on assigning weights to all processes and maintaining the invariant that the sum of the weights is zero are developed in [115]. A modification to this algorithm using graph searching techniques is given in [73]. Based on the idea of message counting, a class of termination detection algorithms is developed in [81, 82, 98, 99].

Termination detection of dynamic systems is carried out in [36, 83, 84, 91].

Formal proofs and systematic development of termination detection algorithms can be found in [7, 8, 9, 13, 14, 19, 21, 42, 50, 62, 71, 72, 100, 107, 117, 123, 137, 138, 139, 141, 142, 145].

[25, 29, 30, 99, 118, 119, 120] deal with the complexity analysis of termination detection algorithms.

Termination detection in the face of uncertainty is addressed in [5, 144].

Chandy and Lamport[26] have presented a very elegant and general scheme for detecting *stable properties* of a network. A property (proposition) is stable if it remains true once it becomes true. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared". The stable property detection problem is further pursued in [20, 31, 40, 66, 68, 88, 96, 101, 127, 131].

An excellant review of termination detection algorithms is given in [98, 136, 138].

1.3 Content of the Thesis

The remainder of the thesis is structured as follows. In Chapter 2, we develop an hybrid approach to designing election algorithm in a complete network with a sense of direction when N the size of the complete network is composite.

In Chapter 3, we develop an election algorithm in a synchronous recursively scalable network.

In Chapter 4, we present a termination detection algorithm in H-meshes. In Chapter 5, we summarize the results of the thesis and suggest possible problems for further research.

Chapter 2

An Hybrid Approach to Designing Election Algorithm in a Complete Network with a Sense of Direction

2.1 Introduction

Loui, Matsushita and West[89, 90] presented a distributed leader election algorithm (hereafter referred to as LMW algorithm) in a complete network with a *sense of direction*[125] which requires less than $3.62N$ messages where N is the number of processors in the network. Based on the *divide* and *conquer* paradigm (i.e., preprocessing and postprocessing) in this chapter, we develop an hybrid approach to designing election algorithm in a complete network with a sense of direction. We show the constant factor 3.62 can be lowered when N is composite (i.e., $N = dN'$). We present the effect of preprocessing (Table 2.1) for a set of values of d.

The improvement in the message complexity is achieved via a two-phase election algorithm. The first phase is called the *preprocessing* and the second phase is called the *postprocessing*. In the preprocessing phase, we solve the problem of electing

a leader efficiently on the smaller size subnetworks. The postprocessing phase is the application of the LMW algorithm on the leaders of the subnetworks from the preprocessing phase.

Due to the terminology used in the description of the two-phase election algorithm, we restate what do we mean by electing a leader as follows. Given a set of processors (with distinct *ids* chosen from a totally ordered set), each in the same initial state, exactly one processor enters the predesignated state called "Elected", whereas of the remaining processors, some (possibly all) enter the predesignated state called "Defeated" and others (possibly empty) are "Dreaming"of being "Elected". The "dreaming" processors eventually enter the "Defeated" state when they hear the results of the election or the message from the postprocessing phase.

Also recall that a complete network has a *sense of direction*[125] if there is a fixed directed Hamiltonian cycle such that at each processor, each link is labeled with the distance along this cycle to the processor at the other end of the link.

2.2 Algorithm and Analysis

Let d divide N. Let us suppose there exists an algorithm X which requires exactly Δ messages to elect a leader in a complete network of size d with a sense of direction. The examples of d and algorithm X are given in the next section. In our two-phase election algorithm, we make use of algorithm X and LMW algorithm as follows.

In the preprocessing phase, we apply the algorithm X simultaneously on $\frac{N}{d}$ sub-networks; each sub-network is a complete network of size d satisfying the following property. Let y be a processor in a subnetwork, S_i for $i \in [1, \frac{N}{d}]$. Then, the remaining

$d-1$ processors in S_i are at distances $\frac{N}{d}$, $2\frac{N}{d}$, $\cdots$, $(d-1)\frac{N}{d}$ from y along the fixed Hamiltonian cycle. The postprocessing phase is the LMW algorithm. Hence, instead of directly applying the LMW algorithm on N *candidate* processors, we apply it on the network which has $\frac{N}{d}$ *candidate* (local leaders) and $(d-1)\frac{N}{d}$ *non-candidate* processors. This results in the constant factor smaller than 3.62 provided d and Δ satisfy the following inequality (see the explanation below)

$$\Delta\frac{N}{d} + (d-1)\frac{N}{d} + 3.62\frac{N}{d} \;<\; 3.62N \tag{2.1}$$

This simplifies to

$$\Delta \;<\; 2.62(d-1) \tag{2.2}$$

The first term in the inequality (2.1) correspond to the cost of preprocessing. The second and the third terms denote the cost of applying the LMW algorithm on the complete network of size N in which $\frac{N}{d}$ processors are *candidates* to become the leader and $(d-1)\frac{N}{d}$ processors are *non-candidates*. The right side term refers to the cost of applying the LMW algorithm on a complete network of size N in which all N processors are *candidates*.

2.3 Examples of d and algorithm X

In this section, we show the existence of algorithms which satisfy the inequality(2.2) when d is 2 , 3 , 4 , 5 or 6. In Table 2.1, we present the effect of preprocessing on the Election Problem in a complete network with a sense of direction. The message complexity in Table 2.1 is computed from the left side of the inequality (2.1).

Theorem 2.1: The Election Problem in a complete network of size 2 can be solved with 2 message transmissions.

Proof: Trivial. Both processors exchange their *ids*. □

Theorem 2.2: The Election Problem in a complete network of size 3 can be solved with 5 message transmissions.

Proof: Refer to Figs. 2.1 and 2.2. We assume the processor *id* i is the largest. Initially, all three processors are *candidates* to become the leader and send their *ids* to their right.

A *candidate* processor upon receiving the message from the left remains as a *candidate* if its *id* is larger than the *id* in the message. Otherwise, it becomes *non-candidate* and forwards the *id* in the message. A *candidate* processor which receives its own *id* or remains as a *candidate* after receiving two messages becomes the leader.

The processor i in Fig. 2.1 and the processors i and k in Fig. 2.2 remain as *candidates* after receiving the first message. The *non-candidate* processors j , k in Fig. 2.1 and j in Fig. 2.2 respectively forward (to the right) the *ids* i, j and i. The processor i in Fig. 2.1 becomes the leader and the processor k in Fig. 2.2 becomes *non-candidate*. The *non-candidate* processor k in Fig. 2.2 now forwards *id* i to processor i which then becomes the leader. □

Theorem 2.3: The Election Problem in a complete network of size 4 can be solved with with 6 message transmissions.

Proof: Consider the complete network of size 4 in Fig. 2.3. Fix a directed Hamiltonian cycle (i, j, k, l, i) as shown by bold arcs. The labels on the links denote the distances between the processors joined by the links along the directed Hamiltonian cycle.

The processors i, j, k and l in Fig. 2.3 first send their *ids* to the processors k, l, i and j respectively. Now, the processors i and k know the *id* of each other and so also the processors j and l. We can logically group these processors as shown in Fig. 2.4. The labels on the link joining the two groups – (i,k) and (j,l) are the same labels joining each processors in Fig. 2.3. For example, the link from i to j has the label 1, the link from i to l has label 3, the link from k to j has label 3 and the link from k to l has label 1 and so on. Now, with two more message transmissions we can elect a leader as follows.

The processor with the smallest *id* from each group sends the largest *id* from its group along link 1. For example, the processor i sends *id* k to the processor j if $i < k$. Otherwise, the processor k sends *id* i to the processor l. Similarly, for the other group (j,l). The processor becomes the leader if the *id* in the message it received from the other group is smaller than the largest *id* of the group it belongs to.

Refer to Figs. 2.5 and 2.6. Assume, the processor *id* i is the largest. The processors i, j, k and l in Figs. 2.5 and 2.6 first send their *ids* respectively to the processors k, l, i and j. Then, the processors k and j in Fig. 2.5 respectively send *ids* i and l to the processors k and l. The processor k becomes the leader. Similarly, the processors k and l respectively send *ids* i and j to the processors l and i. The processor i becomes the leader. Observe that in Fig. 2.5 none of the processors i, j or l can become leader. For, $l < i$ and the processors i and j do not receive the second message. Similarly in Fig. 2.6 none of the processors j , k or l can become the leader. For, $j < i$ and the processors j and k do not receive the second message. □

Theorem 2.4: The Election Problem in a complete network of size 5 can be solved with 9 message transmissions.

Proof: Consider the Algorithm A. It realizes the result stated in Theorem 2.4. An informal description and the correctness of the Algorithm A are given in the subsequent paragraphs.

Algorithm A :

1. send(1; 0, my-id) /* send my-id to the right */
2. receive(msg_type, left-id) /* receive left-id from the left */

   if my-id > left-id ⟶ candidate := true

   □ my-id < left-id ⟶ candidate := false; /* become non-candidate */

   send(2; 0, my-id) /* send my-id along link 2 */

   fi
3. **if** candidate ∧ received(msg_type, left-left-id)

   ⟶ **if** msg_type = 0 ⟶ **if** left-id < left-left-id ⟶ "Elected"

   □ left-id > left-left-id ⟶

   send(2; 1, _); candidate := false

   /* send nomination messsage along link 2 */

   /* become non-candiadate */

   fi

   □ msg_type ≠ 0 ⟶ "Elected"

   fi

   □ ¬candidate ∧ received(msg_type, id) ⟶ skip

   fi

The communication primitives are **send(link; msg)** and **receive(msg)** where **link** is the link along which **msg** is to be sent. **msg** consists of two fields – **message_type** and **id**. The purpose of the **message_type** field is to nominate the receiving processor as a leader. A message whose **message_type** field is set to 1 is called a *nomination* message. Otherwise it is called a *non-nomination* message (i.e., **message_type** field is set 0). In a *nomination* message the **id** field is not used (denoted by underscore) whereas in a *non-nomination* message the **id** field contains the identity of the sending processor.

Consider the complete network of size 5 in Fig. 2.7. Fix a directed Hamiltonian cycle (i, j, k, l, m, i) as shown by bold arcs. The labels on the links denote the distances between the processors joined by the links along the directed Hamiltonian cycle. Initially, all 5 processors are *candidates* to become the leader and send their *ids* to their immediate right (link 1).

A *candidate* processor upon receiving the first message from the immediate left (link 4) remains as a *candidate* if its *id* is larger than the *id* in the message. Otherwise, it becomes *non-candidate* and sends its *id* along link 2. Furthermore, it discards any message if received.

A *candidate* processor upon receiving the second message along link 3 sends a nomination message along link 2 and becomes *non-candidate* only if the second message is not a *nomination* message and the *id* in the first message is larger than the *id* in the second message.

A *candidate* upon receiving the second message along link 3 becomes the leader if either it is a *nomination* message or *non-nomination* message but the *id* in the first message is smaller than the *id* in the second message.

Refer to Fig. 2.7. Without loss of generality assume the processor i becomes the leader. Then, we show that none of the processors j , k , l or m can become leader. Since the processor i becomes the leader then one of the following two conditions is true.

1. $i > m$ and the processor i has received a *nomination* message from the processor l.

2. $i > m$ and the processor i has deduced that $m < l$.

In either case, the processor i does not send the second message to the processor k. Hence, the processor k cannot become the leader.

Let the condtion 1 be true. This implies that the processors j and l are *non-candidates* and $k > j$. Since $k > j$, the processor k does not send the second message to the processor m and thus prevents the processor m from becoming the leader.

Let the condtion 2 be true. This implies that the processors l and m are *non-candidates*. Suppose $i < j$. The processor j upon receiving the second message from the processor m deduces that $i > m$. The processor j therefore sets itself as *non-candidate* and sends a *nomination* message to the processor l. But this nomination message is discarded by the processor l since it is a *non-candiadate*. □

Theorem 2.5: The Election Problem in a complete network of size 6 can be solved with 11 message transmissions.

Proof: Consider the complete network of size 6 in Fig. 2.8. Fix a directed Hamiltonian cycle (i, j, k, l, m, n, i) as shown by bold arcs. The labels on the links denote the distances between the processors joined by the links along the directed Hamiltonian cycle. As in Theorem 2.3, we group the processors into groups of 2 as shown in Fig. 2.9 and apply the ideas of Theorem 2.2 as follows.

The processor with the smallest *id* from each group sends the largest *id* from its group along link 1. For example, the processor i sends *id* l to the processor j if $i < l$. Otherwise, the processor l sends *id* i to the processor m. Similarly, for the other two groups (j, m) and (k, n). The processor x from the group (x, y) upon receiving the messsage from the immediate left (link 5) responds as follows. Let the *id* in message be z.

1. $x < y$. If $z > y$ then the processor x forwards the *id* z along link 1. Otherwise, the processor x waits to receive the second message either along link 5 or along link 2.

2. $x > y$. If $z > x$ then the processor x forwards the *id* z along link 4. (Because the processor y sends *id* x along link 1 and the label on the link from x to y is 3) Otherwise the processor x waits to receive the second message either along link 5 or along link 2.

The processor x from the group (x, y) becomes the leader if either it receives the message which contains the largest *id* of its group or it has received two messages such that the *ids* in both the messages are smaller than the largest *id* of its group.

The formal specification is given in Algorithm B. **msg-received(link; id)** is a predicate which is true only if the (preprocessing) message is received via the **link.**

Algorithm B :

```
1. send(3; my-id)        /*send to distance 3 */
2. receive(opp-id)       /* receive from link 3 */
   if    my-id > opp-id  ⟶  large := true
     □   my-id < opp-id  ⟶  large := false;
                            send(1; opp-id)
   fi
3. if    large ∧ msg-received(5; right-right-id)
         ⟶ if    my-id > right-right-id
                 ⟶ receive(any-id)
                    if    my-id > any-id ⟶ "Elected"
                      □   my-id < any-id ⟶ send(4; right-right-id)
                    fi
              □  my-id < right-right-id
                 ⟶ send(4; right-right-id);
                    ignore any incoming (preprocessing) msg
           fi
     □  ¬ large ∧ msg-received(5; right-right-id)
         ⟶ if    opp-id > right-right-id
                 ⟶ receive(any-id)
                    if    opp-id > any-id ⟶ "Elected"
                      □   opp-id < any-id ⟶ send(1; right-right-id)
                    fi
              □  opp-id < right-right-id
                 ⟶ send(1; right-right-id);
                    ignore any incoming (preprocessing) msg
           fi
   fi
```

We prove below that the Algorithm B realizes the result stated in Theorem 2.5.

Refer to Fig. 2.9. Without loss of genarality, assume the processor i in the group (i,l) becomes the leader. Then, we show that none of the processors j , k , l , m or n can become leader. Since the processor i becomes the leader then one of the following two conditions is true. $max(x,y)$ is a function which returns x if $x > y$ else returns y.

1. The processor i received $max(i,l)$ from one of the processors of the group (k,n).

2. The processor i received both $max(k,n)$ and $max(j,m)$ and deduced that $max(i,l) > max(j,m)$ and $max(i,l) > max(k,n)$.

Let the condition 1 be true. This implies that $max(i,l) > max(j,m) > max(k,n)$. Therefore, none of the above two conditions for becoming the leader can hold for any of the processors of the two groups (j,m) and (k,n).

Let the condition 2 be true. This implies that $max(i,l) > max(k,n) > max(j,m)$. Again, none of the above two conditions for becoming the leader can hold for any of the processors of the two groups (j,m) and (k,n).

In Fig. 2.9, exactly 5 messages are transmitted. Since, the processor i is the leader then it has received two messages from the group (k,n). The first message contains $max(k,n)$ and the second message contains either $max(j,m)$ or $max(i,l)$. From group (j,m) to group (k,n) two messages are sent – the first contains $max(j,m)$ and the second contains $max(i,l)$. From group (i,l) to group (j,m) one message containing $max(i,l)$ is sent. Therefore, the processor l can neither receive $max(i,l)$ nor $max(k,n)$ and $max(j,m)$ to deduce that $max(i,l) > max(j,m) > max(k,n)$. □

2.4 Complexity Analysis

1. **Δ values in Table 2.1 are optimal.**

   A processor is said to be a *candidate* if it "wishes" to become the leader. A *candidate* processor upon receiving the first message remains as a *candidate* if its *id* is larger than the *id* in the message. Otherwise it becomes *non-candidate*.

   Let a *scenario* be a tuple $< n_c, m_{nc} >$ such that $n + m = N$ where N is the number of *candidate* processors prior to the first message is received by any of the *candidate* processors, n_c and m_{nc} respectively denote that n processors remain as *candidate* and m processors become *non-candidate* upon receiving the first message by all the *candidate* processors.

   Theorem 2.6: Any distributed algorithm for electing a leader in a complete network of size N with a sense of direction requires at least $2N - 1$ message transmissions if N is prime and initially all N processors are *candidates* (to become the leader).

   Proof: Since, initially all N processors are *candidates* to become the leader then the first *step* (or *action*) of any election algorithm results in N message transmissions and in following possible *scenarios* — $< 1_c, N - 1_{nc} >, < 2_c, N - 2_{nc} >, \cdots, < N - 1_c, 1_{nc} >$.

   Observe that the leader election is not complete and the election algorithm needs to be extended. Two possible extensions are — (1) letting the *candidate* processors take the next *step* of sending a "message", (2) letting the *non-candidate* processors do the same. In either case, there exists a *scenario* — $< N - 1_c, 1_{nc} >$ in the first case or $< 1_c, N - 1_{nc} >$ in the second case, which

results in $N-1$ message transmissions. Note choosing both cases would result in N message transmissions. Hence, $2N-1$ message transmissions are at least required. □

For $d=2$, it is obvious that $\Delta=2$ is optimal. From Theorem 2.6, for $d=3$ and $d=5$, $\Delta=5$ and $\Delta=9$ respectively are optimal.

For $d=4$, refer to Fig. 2.3. If the first message containing the *id* of the sending processor is sent either along link 1 or along link 3 results in the following *scenarios* — $< 1_c, 3_{nc} >, < 2_c, 2_{nc} >, < 3_c, 1_{nc} >$. However, if the link 2 is selected results in the *scenario* - $< 2_c, 2_{nc} >$. Hence, $\Delta=6$ is optimal.

For $d=6$, refer to Fig. 2.8. If the message containing the *id* of the sending processor is sent either along link 1 or along link 5 results in the following possible *scenarios* — $< 1_c, 5_{nc} >, < 2_c, 4_{nc} >, < 3_c, 3_{nc} >, < 4_c, 2_{nc} >, < 5_c, 1_{nc} >$. Hence, 11 message transmissions are required.

If either link 2 or link 4 is selected results in the decomposition of the network into 2 subnetworks each of size 3. Such a decomposition requires $12(=5+5+2)$ messsage transmissions. Because, for $d=3$, $\Delta=5$ is optimal and for $d=2$, $\Delta=2$ is optimal.

However, if the link 3 is selected results in the decomposition of the network into 3 sub-networks each of size 2. Such a decomposition requires $11(=2+2+2+5)$ message transmissions. Because, for $d=2$, $\Delta=2$ is optimal and for $d=3$, $\Delta=5$ is optimal.

2. **Generalization of inequality (2.1)**

In the postprocessing phase, we used LMW algorithm which requires at the most

$3.62N$ messages. It is useful to determine for each d the minimum c such that if the postprocessing algorithm requires more than cN messages (on a network of size N) then the hybrid approach that uses preprocessing algorithms for d developed in this paper uses fewer than cN messages.

Replacing the constant factor 3.62 in equality(2.1) by a constant c, we obtain

$$\Delta\frac{N}{d} + (d-1)\frac{N}{d} + c\frac{N}{d} \;<\; cN \tag{2.3}$$

This simplifies to

$$c \;>\; 1 + \frac{\Delta}{d-1} \tag{2.4}$$

d	Δ	message complexity
2	2	3.310 N
3	5	3.540 N
4	6	3.155 N
5	9	3.324 N
6	11	3.270 N

Table 2.1: Effect of Preprocessing

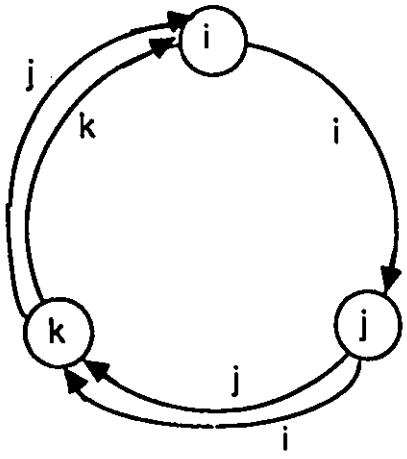


Fig. 2.1 k < j

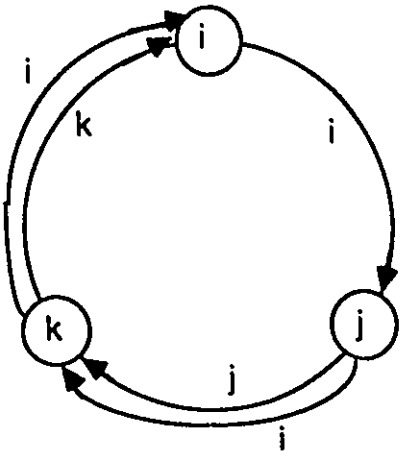


Fig. 2.2 k > j

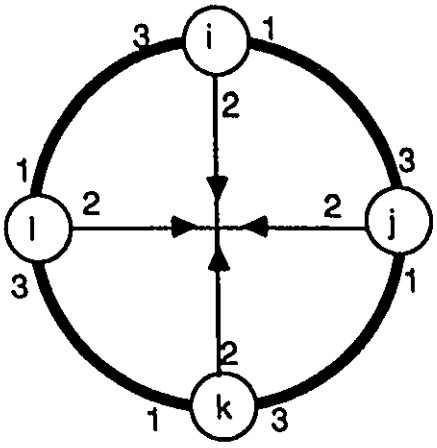


Fig. 2.3

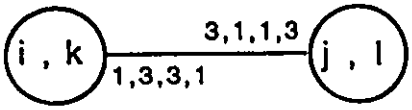


Fig. 2.4

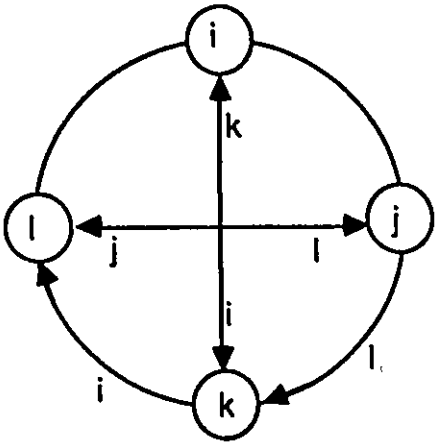


Fig. 2.5 j < l

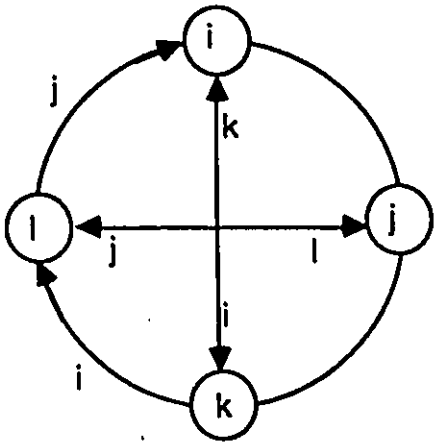


Fig. 2.6 j > l

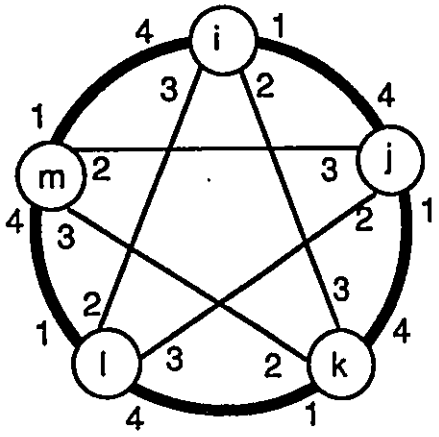


Fig. 2.7

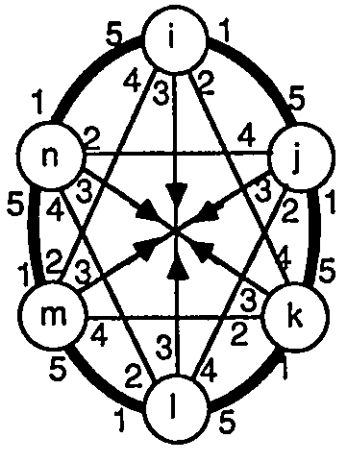


Fig. 2.8

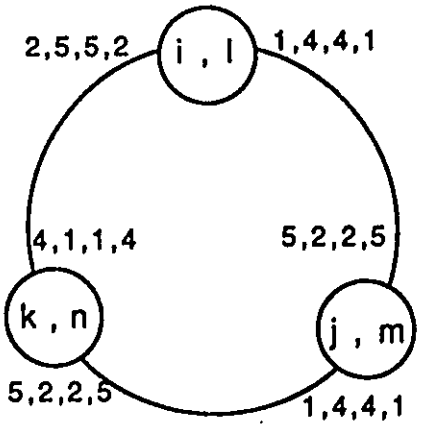


Fig. 2.9

Chapter 3

Election in a Synchronous Recursively Scalable Network

3.1 Introduction

In the past, a number of researchers[16, 37, 60, 79, 93, 143, 146] have investigated a new set of principles for program and structural organization of digital computers based on the concept of recursion. Such machines are referred to as *Recursive Machines*. Recently, also based on the concept of recursion, a general class of topologies for message passing architecture called *Recursively Scalable Networks* is examined in [38, 39].

Our interest in recursively scalable networks stems from our work (in the previous chapter) on election in complete networks. Presently, the leader election in a complete network with a sense of direction requires less than $3.62N$ messages where N is the size of the network[89]. In Chapter 2, we showed how to lower this constant factor 3.62 when N is composite. This is possible because we view the election as a two level process — in the first level, we divide the network into smaller-size subnetworks and solve the election problem efficiently on these subnetworks, and in the second level,

we apply the existing algorithm[89] on the leaders from the first level. This *divide and conquer* paradigm led us to address the election problem for networks which are decomposable. Hence, our attention is directed towards recursively scalable networks.

In this chapter, we develop a distributed leader finding algorithm in a synchronous recursively scalable network. It requires less than $4.5N$ messages where N is the number of processors in the network.

The remainder of the chapter is structured as follows. In section 3.2, we briefly describe recursively scalable networks. The algorithm to elect a leader in a synchronous recursively scalable network is given in section 3.3 and is followed by the complexity analysis in section 3.4.

3.2 Model

In this section, we informally describe a general class of topologies for message-passing architectures called *Recursively Scalable Networks* examined by Vecchia and Sanges[38, 39]. Very recently, stduies exploring the characteristics of recursively scalable networks and the suitability of constructing massively parallel computer systems based on recursively scalable networks are also carried out in the literature[74, 94, 95].

Definition 3.1: **(base structure or virtual node)**

Let K_n be a complete network of size n. Let v be a vertex in K_n. Deleting v results in a structure which is K_{n-1} and has $n-1$ free links, we define such a structure as *base structure* or *virtual node*.

Refer to Figs. 3.1(a)-3.1(c). Deleting the vertex v in Fig. 3.1(a) results in a structure as shown in Fig. 3.1(b). The vertex $\overline{v}$ is called the virtual node and is

formed by four completely connected real nodes — w , x , y and z. It serves as a base structure for constructing larger structures and is referred to as a 1^{st} level virtual node. Using $\overline{v}$, we can construct a 2^{nd} level virtual node consisting of $16(= 4 \times 4)$ real nodes as shown in Fig. 3.1(c). Analogously, we can build a l^{th} level virtual node. Such a class of topologies are called recursively scalable networks[38, 39].

Let k be the number of real nodes (or free links) in a base structure (i.e., 1^{st} level virtual node). Then, a l^{th} level virtual node consists of k, $(l-1)^{th}$ level virtual nodes. Observe that k is also the node degree. Furthermore, let N be the number of real nodes and L be the expansion level. Then, the analytical relation $L = \log_k N$ holds among N , k and L.

In the remainder of the chapter, we are only interested in networks of node degree four, i.e., $k = 4$. We assume a synchronous mode of communication. Let the delay on each link be at the most one time unit. We also assume that the processors have a *sense of direction*[125]. That is, informally, the processors have a uniform notion of which of their four links is assigned which value from the set of possible directions – {East, South-East, South, South-West, West, North-West, North, North-East}. For example, if a processor x sent a message M along its South-East link then the processor y that receives M knows that it received M from the North-West link. Furthermore, we assume that each processor (node) has a unique identifier (id) chosen from a totally ordered set. Initially, no processor knows the id of any other processor.

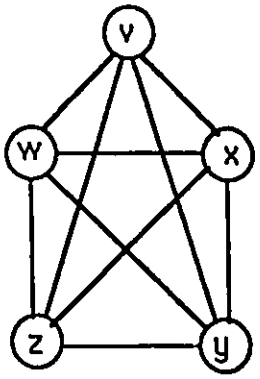


Fig. 3.1(a)

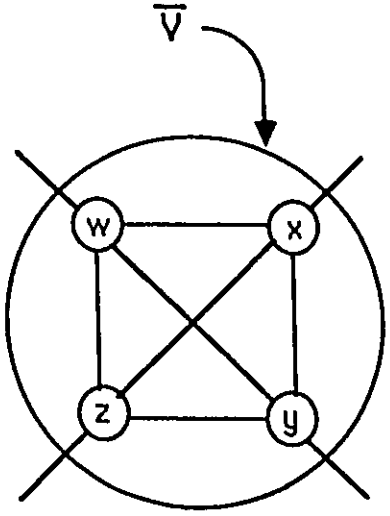


Fig. 3.1(b)

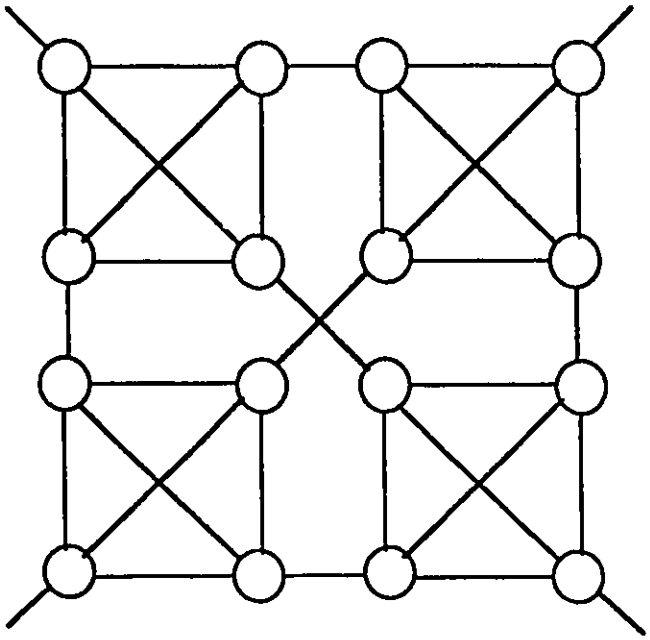


Fig. 3.1(c)

3.3 Algorithm

3.3.1 Informal Description

The computation to elect a leader is divided into rounds. It requires L rounds where L is the expansion level. In the first round, 4^{L-1} leaders are elected among level 1 nodes; in the second round, 4^{L-2} leaders are elected among level 2 nodes; and so on, finally in the L^{th} round, $4^{L-L} = 1$ leader is elected among level L nodes. We informally describe below the computational steps of the i^{th} round.

Fig. 3.2 refers to a $(i+1)^{th}$ level virtual node and we refer to it by its four end nodes — $\{c_1, c_5, c_8, w\}$. The number of such $(i+1)^{th}$ level virtual nodes in a L^{th} level virtual node is equal to 4^{L-i-1}. The $(i+1)^{th}$ level virtual node $\{c_1, c_5, c_8, w\}$ in turn consists of four i^{th} level virtual nodes — $\{c_1, c_2, p, c_3\}$, $\{c_4, c_5, c_6, r\}$, $\{q, c_7, c_8, c_9\}$ and $\{t, s, z, w\}$. Furthermore, each of these i^{th} level virtual nodes consists of four $(i-1)^{th}$ level virtual nodes. Let the nodes — a, b, c, d; e, f, g, h; j, k, l, m; and x, y, u, v respectively be the leaders (or 'playing' on behalf of the 'actual' leaders) of the $(i-1)^{th}$ level virtual nodes. In the i^{th} round, we find four leaders, one from each of these four groups of nodes and specify how to proceed to the $(i+1)^{th}$ round (i.e., the nodes p, q, r and s can start the $(i+1)^{th}$ round). Note that these computational steps are also carried out simultaneously in the remaining $(i+1)^{th}$ level virtual nodes.

We describe next a procedure to elect a leader among four nodes — x, y, u, v and notify the node s to start the $(i+1)^{th}$ round. It consists of four phases. Let us divide these four nodes into two groups — $< x, y >$ and $< u, v >$. In the first phase, x and y, u and v exchange their *ids*. In the second phase, the node with the largest *id* from each group sends its *id* to the other group along the bold path. In the third phase,

either x or y, and, either u or v respectively notify the nodes z and w, the largest *id* between the two groups and also the indication as to whether the largest *id* is from its group or not.

In the fourth phase, the nodes w and z decide which node can start the $(i+1)^{th}$ round. However, a node can start the $(i+1)^{th}$ round only after it has received a 'go' type message and it has either pair of links — <South-East , North-West> , <South-West , North-East> (we refer to such a pair of links as diagonal links). Hence, a node which has received the *id* of the leader of the i^{th} level virtual node and does not have diagonal links sends either a 'go' and a 'kill', or a 'go' type messages depending upon its adjacency pattern. Exact conditions for sending these messages are given in the formal specification of the algorithm. Therefore, the node z sends a 'go' type message to the node s and a 'kill' type message to the node w. Irrespective of what the node w does, it can not start the $(i+1)^{th}$ round. Similarly, the nodes p, q and r each receive a 'go' type message.

Observe that in the fourth phase of the L^{th} round, there are no 'go' and 'kill' type messages. For, let us assume that nodes c_1, c_5, c_8 and w (Fig. 3.2) are the end nodes of the L^{th} level virtual node. Each of these end nodes has one free link which can be used to construct the $(L+1)^{th}$ virtual node. We assume that the nodes c_1 and c_8 are directly connected and so also the nodes c_5 and w, with the convention that the South-East link of the node c_8 is connected to the North-West link of the node c_1 and likewise for the nodes c_5 and w. Now, at the end of the third phase of the L^{th} round, the nodes w and c_8 have the *id* of the leader and also the indication as to which group — $< p, q >$ or $< r, s >$, the leader belongs to. In the fourth phase, the nodes w and c_8 do not send any 'go' and 'kill' type messages. Instead, if they do not

receive any message in 2^L time units, one of the two declares itself as the leader (i.e., w is elected as the leader if the largest *id* is from the group $< r, s >$ and likewise c_8 is elected as the leader if the largest *id* is from the group $< p, q >$) and the other sets itself as defeated. Finally, the nodes c_8 and w respectively notify the results of the election to the nodes c_1 and c_5.

Observe that there exist 1^{st} level virtual nodes which require $12(= 4 + 2 + 2 + 4)$ messages if we employ the above described four phases of computation in the first round. However, we can optimize the 1^{st} round by deleting the third phase and not generating the 'kill' type messages in the fourth phase. Hence, any 1^{st} level virtual node generates at most 8 messages.

3.3.2 Formal Specification

The algorithm is divided into four segments — **Initialization, Round-1, Round-i** and **Fulfill-Obligation.** Every processor starts in the **Initialization** segment followed by **Round-1.** Depending upon its adjacency pattern, the processor either enters **Round-i** segment or **Fulfill-Obligation** segment. A processor enters **Round-i** segment at most once. The algorithm is specified using Dijkstra's guarded commands. To facilitate the jumping to the non-consecutive segments, we have also included a command — **Skip to label** where **label** is the segment label. Observe that there is no communication in the **Initialization** segment. Every processor depending upon its adjacency pattern initializes certain variables which will be used later.

To facilitate the specification of the algorithm, we employ few useful notations. A 3-tuple (*dir-1, dir-2, dir-3*) is a logical variable which is true for a processor x if three of its links are *dir-1*, *dir-2* and *dir-3*. Obvious mnemonics are used for directions,

e.g., SE for South-East, etc. The function $max(x, y)$ returns x if $x > y$ else y. The communication primitives are : **Send(link, msg)** and **Receive(msg)**, where **link** is the link on which **msg** is to be sent. **msg** is a 5-tuple < **dist, msg-type, node-id, flag, round** >, where **dist** indicates how many hops away the destination node is; **msg-type** $\in$ {play, go, kill, end}; **flag** indicates the largest *id* for a given round is from its group or not and **round** is the round number. In many situations, e.g., **Round-1** etc., there is no need for all the fields; they are hence dropped. For ease of expressing, the last three fields – **node-id, flag, round** are grouped and is referred to as **msg-con. received(msg, time)** is a communication predicate which is true if any message is received within **time** units otherwise false. **received(msg)** is also a communication predicate but there is no upper limit on time. When a processor is in **Fulfill-Obligation** segment, it fulfills obligation only if **received(msg)** is true.

Initialization :

```
if    (S,E,SE)   ⟶  link-next ⟵ East
                    if      ¬NW-link ⟶
                              if      North-link ⟶ link-go ⟵ East
                                  □   West-link  ⟶ link-go ⟵ South
                              fi
                              link-i ⟵ nil , links-used ⟵ true
                        □     NW-link ⟶
                              link-go ⟵ nil , link-i ⟵ NW , links-used ⟵ false
                    fi
  □ (S,W,SW)   ⟶  link-next ⟵ West
                    if      ¬NE-link ⟶
                              if      North-link ⟶ link-go ⟵ West
                                  □   East-link  ⟶ link-go ⟵ South
                              fi
                              link-i ⟵ nil , links-used ⟵ true
                        □     NE-link ⟶
                              link-go ⟵ nil , link-i ⟵ NE , links-used ⟵ false
                    fi
  □ (N,E,NE)   ⟶  link-next ⟵ East
                    if      ¬SW-link ⟶
                              if      South-link → link-go ← East , link-kill ← nil
                                  □   West-link  → link-go ← North , link-kill ← East
                              fi
```

```
                                link-i ⟵ nil , links-used ⟵ true
                        □       SW-link ⟶
                                link-go ⟵ nil , link-i ⟵ SW , links-used ⟵ false
                    fi
□  (N,W,NW) ⟶       link-next ⟵ West
                    if          ¬SE-link ⟶
                                if      South-link → link-go ← West , link-kill ← nil
                                    □   East-link  → link-go ← North , link-kill ← West
                                fi
                                link-i ⟵ nil , links-used ⟵ true
                        □       SE-link ⟶
                                link-go ⟵ nil , link-i ⟵ SE , links-used ⟵ false
                    fi
fi
```

Round-1 :

Round ⟵ 1

if (S,E,SE) ⟶ Send(SE, my-id)

□ (S,W,SW) ⟶ Send(SW, my-id)

□ (N,E,NE) ⟶ Send(NE, my-id)

□ (N,W,NW) ⟶ Send(NW, my-id)

fi

Receive(opp-id)

if my-id > opp-id ⟶ Send(link-next, my-id) , first-id ⟵ my-id

□ my-id < opp-id ⟶ first-id ⟵ opp-id

fi

if received(opp-opp-id, 1) ⟶

if link-go ≠ nil ⟶ Send(link-go, 'play', max(first-id, opp-opp-id), Round)

□ link-go = nil ⟶ **Skip**

fi

□ ¬received(–, 1) ⟶ **Skip**

fi

if received(<'play', play-id, Round>, 1) ⟶

if link-i ≠ nil → Selected-dir ← link-i , links-used ← true , **Skip to Round-i**

□ link-i = nil → **Skip to Fulfill-Obligation**

fi

□ ¬received(–, 1) ⟶ **Skip to Fulfill-Obligation**

fi

Round-i :

Send(Selected-dir, play-id) , Receive(opp-play-id)

if play-id > opp-play-id $\longrightarrow$

first-id $\longleftarrow$ play-id

if North-link $\longrightarrow$ Send(North, $< 2^{\text{Round}} - 1$, play-id>)

$\square$ South-link $\longrightarrow$ Send(South, $< 2^{\text{Round}} - 1$, play-id>)

fi

$\square$ play-id < opp-play-id $\longrightarrow$ first-id $\longleftarrow$ opp-play-id

fi

if received(<opp-opp-play-id>, $2^{\text{Round}} - 1$) $\longrightarrow$

second-id $\longleftarrow$ max(first-id, opp-opp-play-id)

if first-id = second-id $\longrightarrow$ flag $\longleftarrow$ true

$\square$ first-id $\neq$ second-id $\longrightarrow$ flag $\longleftarrow$ false

fi

if SE-link $\longrightarrow$

if Selected-dir = SE

$\longrightarrow$ Send(SE, $< 2^{\text{Round}-1}$, 'play', second-id, flag, Round>)

$\square$ Selected-dir = NW

$\longrightarrow$ Send(SE, $< 2^{\text{Round}-1} - 1$, 'play', second-id, flag, Round>)

fi

$\square$ SW-link $\longrightarrow$

if Selected-dir = SW

$\longrightarrow$ Send(SW, $< 2^{\text{Round}-1}$, 'play', second-id, flag, Round>)

$\square$ Selected-dir = NE

$\longrightarrow$ Send(SW, $< 2^{\text{Round}-1} - 1$, 'play', second-id, flag, Round>)

fi

fi

$\square$ $\neg$received(–, $2^{\text{Round}} - 1$) $\longrightarrow$ **Skip**

fi

Fulfill-Obligation :

pre-condition : received(<dist, msg-type, msg-con>) is true

```
if      dist > 1 ⟶
        if      msg-type ∈ {go, kill, play} → Send(¬ from, <dist - 1, msg-type, msg-con>)
           □    msg-type ∉ {go, kill, play} →
                if      from = South ⟶
                        if      North-link → Send(North, <dist-1, msg-type, msg-con>)
                           □    ¬North-link →
                                if      (S,W,SW) → Send(East, <dist-1, msg-type, msg-con>
                                   □    (S,E,SE) → Send(West, <dist-1, msg-type, msg-con>)
                                fi
                        fi
                   □    from = North ⟶
                        if      South-link → Send(South, <dist-1, msg-type, msg-con>)
                           □    ¬South-Link ⟶
                                if      (N,W,NW) → Send(East, <dist-1, msg-type, msg-con>
                                   □    (N,E,NE) → Send(West, <dist-1, msg-type, msg-con>
                                fi
                        fi
                   □    from = East ⟶
                        if      (S,W,SW) → Send(South, <dist-1, msg-type, msg-con>)
                           □    (N,W,NW) → Send(North, <dist-1, msg-type, msg-con>)
```

```
                    fi
                □ from = West ⟶
                    if      (S,E,SE) → Send(South, <dist-1, msg-type, msg-con>)
                        □   (N,E,NE) → Send(North, <dist-1, msg-type, msg-con>)
                    fi
            fi
        fi
□ dist ≯ 1 ⟶
    if      msg-type = 'End'    ⟶  links-used ⟵ true
        □ msg-type ≠ 'End'      ⟶
            if      (SE-link ∨ SW-link) ⟶
                    if      received('kill', 2^Round) ⟶  Skip
                        □   received('go', 2^Round)   ⟶
                            Selected-dir ⟵ link-i , links-used ⟵ true
                            Round ⟵ Round + 1 , Skip to Round-i
                        □ ¬received(–, 2^Round) ⟶
                            if      flag ⟶ Elected
                                □ ¬flag ⟶ Defeated
                            fi
                            if      SE-link ⟶ Send(SE, 'End')
                                □   SW-link ⟶ Send(SW, 'End')
                            fi
                    fi
                □ ¬(SE-link ∨ SW-link) ⟶ Send(link-go, < 2^Round – 1, 'go'>)
```

```
                    if      link-kill ≠ nil ⟶ Send(link-kill, < 2^Round − 1, 'kill'>)
                        □   link-kill = nil ⟶ Skip
                    fi
                fi
            fi
        fi
```

3.4 Complexity Analysis

Theorem 3.1: The algorithm requires at the most $2N$ messages in round 1.

Proof: Let L be the expansion level. We have, therefore, 4^{L-1} 1^{st} level virtual nodes. In round 1 of the algorithm, we simultaneously elect a leader in each 1^{st} level virtual node. At the most 8 messages are required to elect a leader in a 1^{st} level virtual node. Since $L = \log_4 N$, the total number of messages required are : $8 \times 4^{L-1} = 2 \times 4^L = 2N$ □

Theorem 3.2: Let m_i be the number of message exchanges in round i. Then,

$$\sum_{i=2}^{L-1} m_i \leq \frac{5}{2}N - 12\sqrt{N} + 2\log_4 N + 4$$

Proof: Consider an i^{th} level virtual node $\{t, s, z, w\}$ in Fig. 3.2. There are 4^{L-i} number of i^{th} level virtual nodes. From the specification of the algorithm, the i^{th} round consists of 4 phases of computation which are carried out simultaneously in all i^{th} level virtual nodes. Let us assume the worst case – $x > y$ and $u > v$. (Note that the name of the node is also its id).

The first phase requires 4 messages. The second phase requires $2(2^{i-1} - 1 + 2^{i-1} -$

$1+1) = 2(2^i - 1)$ messages whereas the third requires $2^{i-1} - 1 + 1 + 2^{i-1} - 1 + 1 = 2 \times 2^{i-1} = 2^i$ messages.

In the fourth phase, depending upon their adjacency pattern the nodes like w and z generate "go" and "kill" type messages which are propagated $2^i - 1$ distance. Let $f(i)$ denote the total number of "go" and "kill" type messages generated in the i^{th} round. Then, the fourth phase of the i^{th} round requires $f(i)(2^i - 1)$ messages. We derive $f(i)$ below.

Refer to Fig. 3.3. From the specification of the algorithm, in the 4^{th} phase of the $(L-1)^{th}$ round, the nodes c_{11}, c_{14}, c_{22} and c_{23} each send a "go" type message and the nodes c_{22} and c_{23} each also send a "kill" type message. Therefore, $f(L-1) = 1 + 1 + [1+1] + [1+1] = 6$. Observe $f(L-1)$ is written as the sum of four terms to emphasize that there are four $(L-1)^{th}$ level virtual nodes. This characterization facilitates the derivation of recurrence relation for $f(i)$.

Refer to Fig. 3.4. Observe that each $(L-1)^{th}$ level virtual node consists four $(L-2)^{th}$ level virtual nodes. From the specification of the algorithm, we find,

$$f(L-2) = [f(L-1)+1] + [f(L-1)+1] + [f(L-1)+2] + [f(L-1)+2] = 30$$

That is, $f(L-2) = 4f(L-1) + 6$.

Similarly, we can obtain the recurrence relations for $f(L-3)$, $f(L-4)$, and so on. Hence,

$$\begin{aligned} f(L-1) &= 6 \\ f(L-i) &= 4f(L-i+1) + f(L-1) \qquad for \quad i > 1 \end{aligned}$$

Hence, the total number of messages are :

$$\sum_{i=2}^{L-1} 4^{L-i}[4 + 2(2^i - 1) + 2^i] + \sum_{i=2}^{L-1} f(i)(2^i - 1)$$

$$= 3N\sum_{i=2}^{L-1}\frac{1}{2^i} + 2N\sum_{i=2}^{L-1}\frac{1}{4^i} + \sum_{i=2}^{L-1} f(i)(2^i-1) \quad = \quad \frac{5}{2}N - 12\sqrt{N} + 2\log_4 N + 4$$

where,

$$\sum_{i=2}^{L-1}\frac{1}{2^i} = \frac{1}{2} - \frac{2}{\sqrt{N}} \quad , \quad \sum_{i=2}^{L-1}\frac{1}{4^i} = \frac{1}{12} - \frac{4}{3N}$$

$$\sum_{i=2}^{L-1} f(i)(2^i-1) = \frac{5}{6}N - 6\sqrt{N} + 2\log_4 N + \frac{20}{3} \qquad \square$$

Theorem 3.3: The algorithm requires at most $3\sqrt{N}+4$ messages in round L.

Proof: Observe that there is only one L^{th} level virtual node since L is the expansion level. There are no "go" and "kill" type messages. Hence, the number of messages required is at most equal to $4 + 2(2^L - 1) + 2 \times 2^{L-1} + 2 \quad = \quad 3\sqrt{N} + 4 \qquad \square$

Theorem 3.4: The algorithm requires at most $4.5N$ messages.

Proof: Adding the messages from Theorems 3.1-3.3, we obtain,

$$2N + \frac{5}{2}N - 12\sqrt{N} + 2\log_4 N + 4 \quad < \quad 4.5N \qquad \square$$

Theorem 3.5: The algorithm runs in time at most $2.75\sqrt{N}$.

Proof: From the specification of the algorithm, round $i \in [2, L-1]$ consists of 4 phases. The first phase requires 1 time unit, the second phase $2^i - 1$ time units, the third phase 2^{i-1} time units and the fourth phase $2^i - 1$ time units. Observe that there is no fourth phase in the L^{th} round and the first round requires 2 time units. Hence, the algorithm runs in time

$$2 + \sum_{i=2}^{L-2}[1 + 2^i - 1 + 2^{i-1} + 2^i - 1] + 2^L + 2^{L-1} \quad < \quad 2.75\sqrt{N} \qquad \square$$

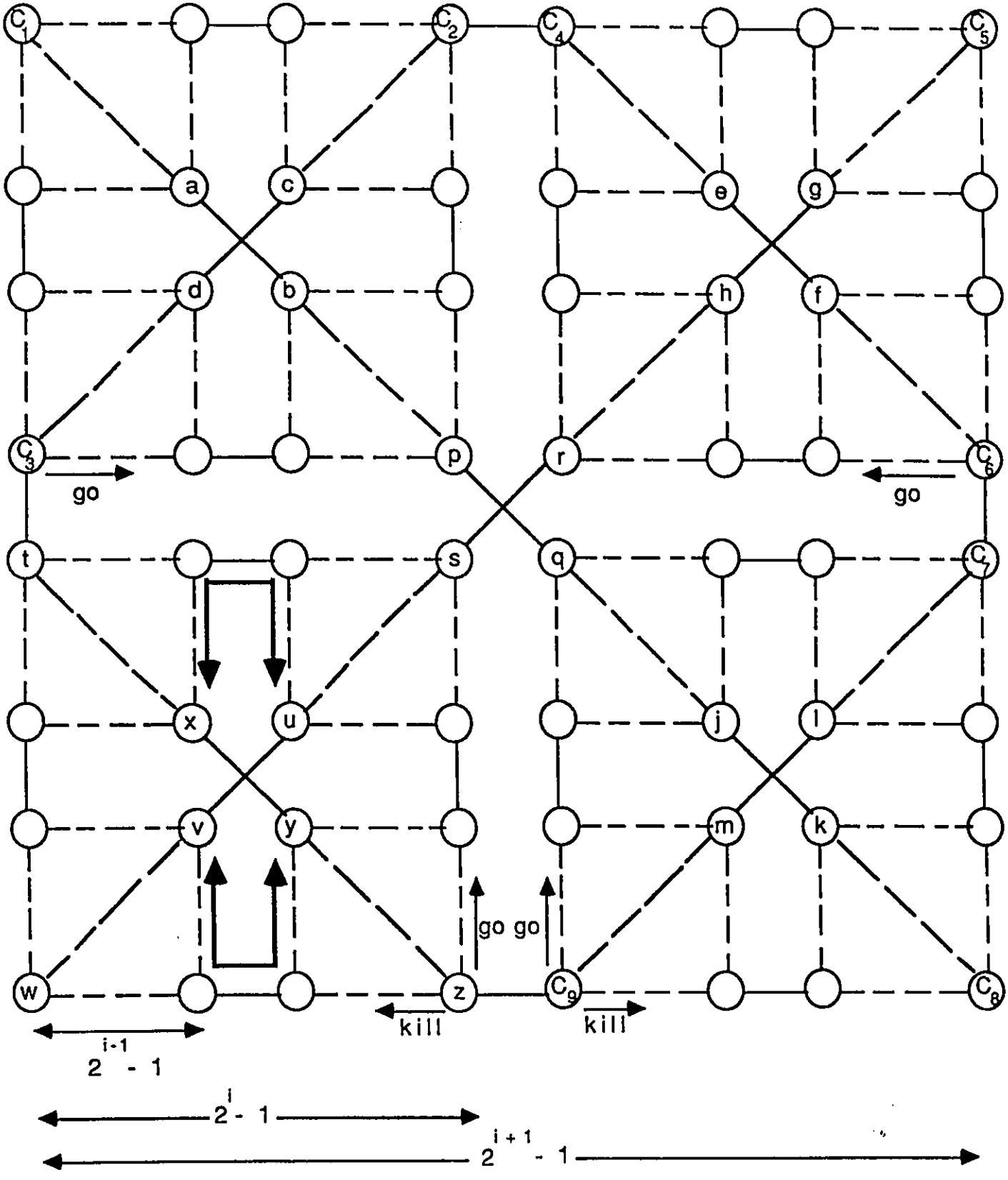

C1
C2
C4
C5
a
c
e
g
d
b
h
f
C3
p
r
C6
go
go
t
s
q
C7
x
u
j
l
v
y
m
k
go go
w
z
C9
C8
kill
kill
i-1
2 - 1
i
2 - 1
i + 1
2 - 1

Fig. 3.2

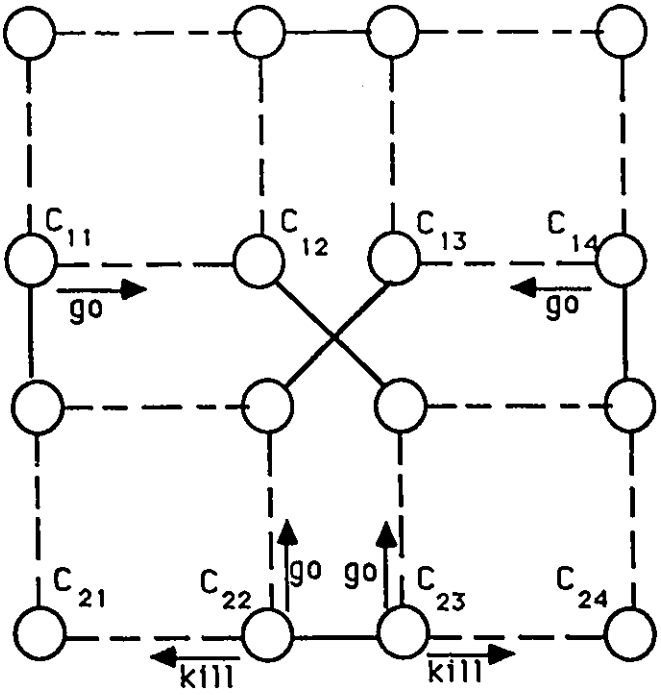

C11
C12
C13
C14
go
go
C21
C22
go
go
C23
C24
kill
kill

Fig. 3.3

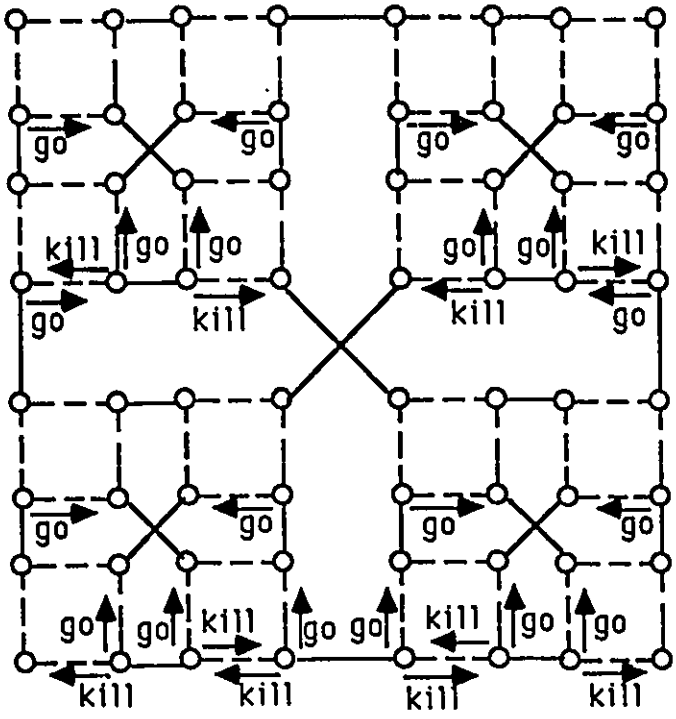

go
go
go
go
kill
go
go
go
go
kill
go
kill
kill
go
go
go
go
go
go
go
go
kill
go
go
kill
go
go
kill
go
go
kill
kill
kill
kill

Fig. 3.4

Chapter 4

Termination Detection in H-meshes

4.1 Introduction

Termination detection like *election* is a fundamental problem in a distributed computing system. It is subjected to intensive investigation by several researchers since it was first formulated and solved by Dijkstra and Scholten[41] and Francez[52]. The termination detection problem is the problem of detecting the following situation:

i) every node in a network is *idle*, i.e., waiting for messages to engage in further computations, and

ii) channels are empty, i.e., all messages that have been sent have been received.

The objective of this chapter is to develop a termination detection algorithm in H-meshes (an abbreviation for Hexagonal meshes). H-meshes are recently considered as a candidate multiprocessor architecture[34, 45, 46, 76, 132]. Topological properties of H-meshes are formally examined in [34].

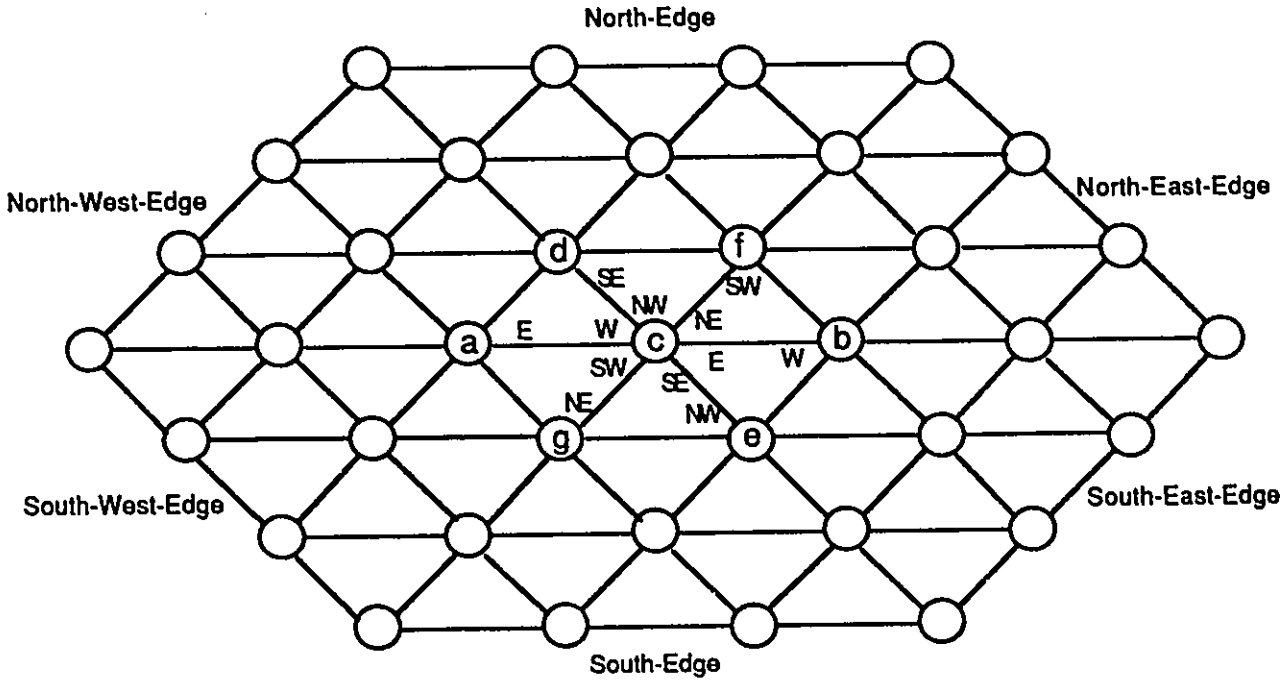
We propose a probe (or token) passing method for detecting the termination of computations in H-meshes. Six probes one for each edge of the H-mesh are initiated to elicit the knowledge about the state of the H-mesh. We define invariants for probes which capture the system state. We then design traffic rules for probes so as to maintain the invariants. We present two sufficient conditions which guarantee that the computation in the H-mesh is terminated if any one of the two conditions is met.

The remainder of this chapter is structured as follows. In section 4.2 we discuss the model. In section 4.3 we present various assumptions, definitions and notations used in the derivation of the proposed termination detection algorithm in H-meshes. In section 4.4 we develop a termination detection algorithm in H-meshes and establish several properties related to the algorithm. We present the correctness proof of the algorithm in section 4.5.

4.2 Model

An H-mesh of size of 4 (denoted by H_4) is shown in Figure 4.1. The size of an H-mesh is defined as the number of nodes (processors) on each peripheral edge of the H-mesh[132].

We assume that the processors have a *sense of direction*[125]. That is, informally, the processors have a uniform notion of which of their six links is assigned which value from the set of possible directions - { East, South-East, South-West, West, North-West, North-East }. For example, if a processor x sent a message M along its South-East link then the processor y that receives M knows that it received M from the North-West link. Figure 4.1 shows the labeling of links of the processor c. Similar



Labels : E , W , NE , SW , NW and SE respectively stand for East-link, West-link, North-East-link, South-West-link, North-West-link and South-East-link

Figure 4.1 An H-mesh of size 4

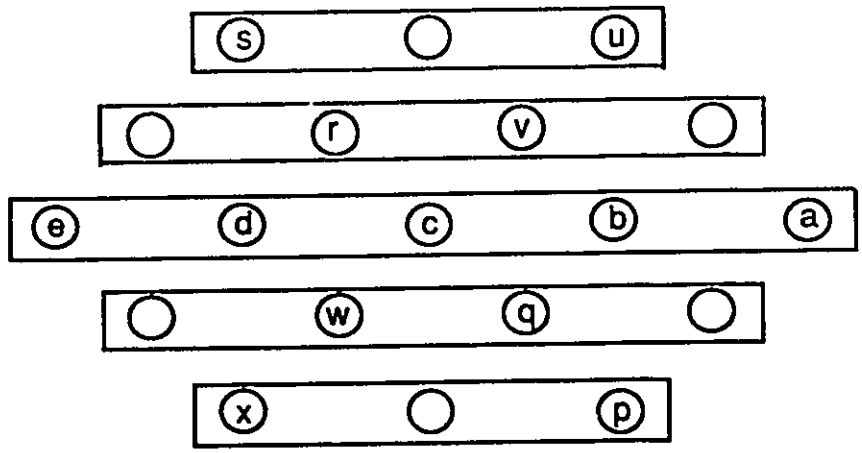


Figure 4.2(a) Partitioned rows of an H_3 in (East, West) direction

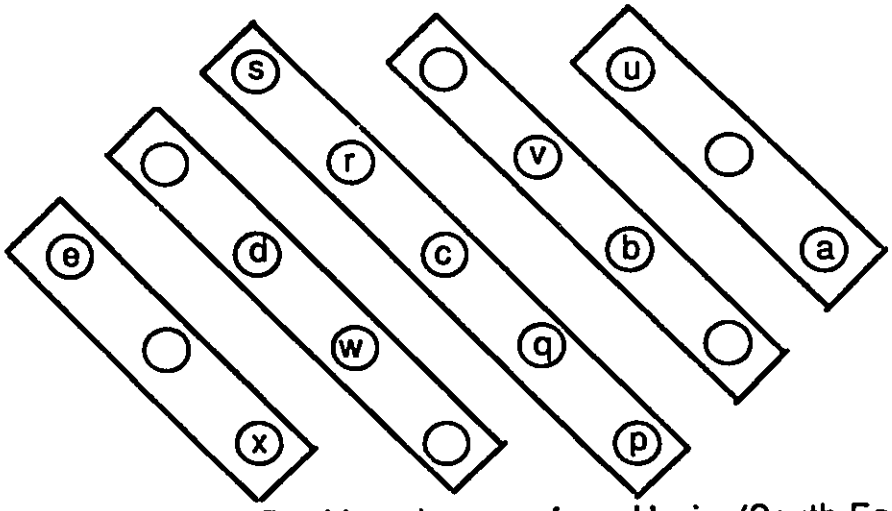


Figure 4.2(b) Partitioned rows of an H_3 in (South-East, North-West) direction

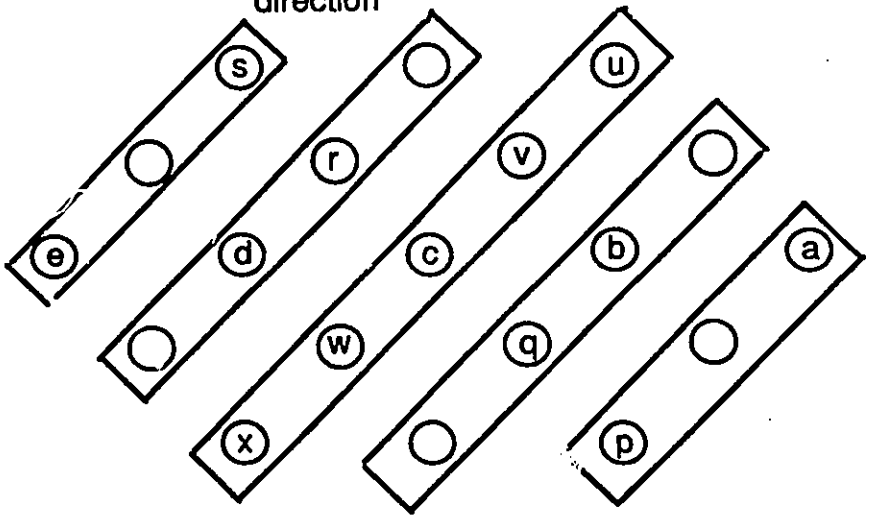
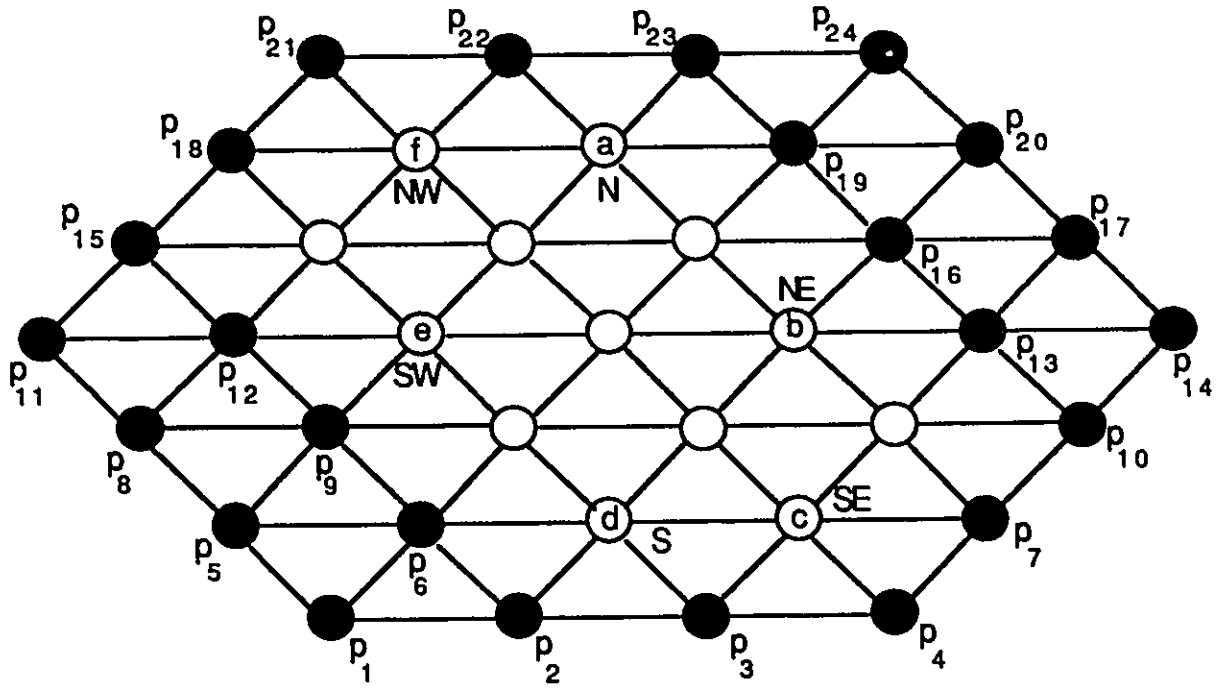


Figure 4.2(c) Partitioned rows of an H_3 in (North-East, South-West) direction



S , N , SW , NE , NW and SE respectively stand for S-probe, N-probe, SW-probe, NE-probe, NW-probe and SE-probe.

Figure 4.3 Communication structure and Notations

convention applies to the remaining processors.

An H-mesh of size n can be partitioned into $2n-1$ rows with respect to any of the three pairs of directions; namely, (East, West), (South-East, North-West) and (North-East, South-West). Figures 4.2(a)-(c) show the rows of H_3 which are partitioned with respect to these three pairs of directions. The row of processors $\{a, b, c, d, e\}$ is referred to as (East, West) row. Similarly, the rows of processors $\{p, q, c, r, s\}$ and $\{u, v, c, w, x\}$ are respectively referred to as (South-East, North-West) row and (North-East, South-West) row.

We refer to the six sides of the H-mesh (Figure 4.1) as North-Edge, North-East-Edge, South-East-Edge, South-Edge, South-West-Edge and North-West-Edge.

4.3 Assumptions, Definitions and Notations

A processor is said to be *active* if it is computing on its assigned task and *passive* if it is completed with its assigned task. For the purpose of illustration (Figure 4.3), a passive processor is denoted by blackened circle whereas white circle denotes an active processor.

A message is either related to the computation of task assigned to each processor or related to the detection of termination of the computation. The former is called the *data* message whereas the latter is called *control* message. We assume that:

a) Each processor is in one of two states; namely, active and passive.

b) A passive processor is activated either by a *data* message from an active processor or by *software loading*.

c) A passive processor does not send *data* messages to its neighbours. A passive processor may communicate via *control* messages with its neighbours in detecting termination.

d) Message transmission is instantaneous as in Dijkstra, Feijen, and Scholten[42].

The computation whose termination is to be detected is referred to as *underlying computation*. The proposed termination detection algorithm is based on two principles; namely, the propagation of passive state of processors and the elicitation of the knowledge about the state of the H-mesh.

Passive state propagation is carried out along the rows which partition the H-mesh; namely, (East, West) row, (South-East, North-West) row and (North-East, South-West) row. The elicitation of the knowledge about the state of the H-mesh is accomplished by six probes; namely, S-probe, N-probe, SW-probe, NE-probe, NW-probe and SE-probe.

The knowledge about the state of the H-mesh can be inferred by designing traffic rules for these probes such that an invariant which captures the system state is preserved. To this end, we can let each probe start from one of the six edges of the H-mesh and travel towards the respective opposite edges. That is, the S-probe can start from the South-Edge and move towards the North-Edge. On the other hand, the N-probe can start from the North-Edge and move towards the South-Edge. The SW-probe can start from the South-West-Edge and move towards the North-East-Edge; conversely NE-probe can start from the North-East-Edge and move towards South-West-Edge. Similarly, the SE-probe can start from the South-East-Edge and move towards the North-West-Edge whereas the NW-probe can start from the North-West-Edge and move towards South-East-Edge. Naturally, we desire that each movement (from a

processor to a processor) of every probe maintains an invariant. Before formally defining invariants for probes, we first describe what does each probe signify.

The S-probe indicates that the processors in all the (East, West) rows South of (not including) the (East, West) row where the S-probe resides, are passive. Similarly, the N-probe indicates that the processors in all the (East, West) rows North of (not including) the (East, West) row where the N-probe resides, are passive. For example, in Figure 4.3 the S-probe is at the processor d. Observe that the processors $\{p_1, p_2, p_3, p_4\}$ in the (East, West) row South of processor d are passive. The N-probe is at the processor a and the processors $\{p_{21}, p_{22}, p_{23}, p_{24}\}$ in (East, West) row North of processor a are passive.

The SW-probe indicates that the processors in all the (South-East, North-West) rows South-West of (not including) the (South-East, North-West) row where the SW-probe resides, are passive. Similarly, the NE-probe indicates that the processors in all the (South-East, North-West) rows North-East of (not including) the (South-East, North-West) row where the NE-probe resides, are passive. For example, in Figure 4.3 the SW-probe is at the processor e. Observe that the processors $\{p_1, p_5, p_8, p_{11}\}$ and $\{p_2, p_6, p_9, p_{12}, p_{15}\}$ in two (South-East, North-West) rows South-West of processor e are passive. The NE-probe is at the processor b and the processors $\{p_{10}, p_{13}, p_{16}, p_{19}, p_{23}\}$ and $\{p_{14}, p_{17}, p_{20}, p_{24}\}$ in two (South-East, North-West) rows North-East of processor b are passive.

The NW-probe indicates that the processors in all the (North-East, South-West) rows North-West of (not including) the (North-East, South-West) row where the NW-probe resides, are passive. Similarly, the SE-probe indicates that the processors in all the (North-East, South-West) rows South-East of (not including) the (North-East,

South-West) row where the SE-probe resides, are passive. For example, in Figure 4.3 the NW-probe is at the processor f. Observe that the processors $\{p_{11}, p_{15}, p_{18}, p_{21}\}$ in the (North-East, South-West) row North-West of processor f are passive. The SE-probe is at the processor c and the processors $\{p_4, p_7, p_{10}, p_{14}\}$ in the (North-East, South-West) row South-East of processor c are passive.

Let R_x^i for $0 \leq i \leq 2n-2$ denote the (East, West) rows of processors of the H-mesh of size n where R_x^0 and R_x^{2n-2} respectively correspond to the South-Edge and the North-Edge. The index i denotes the row number of the (East, West) row R_x^i. For every processor in R_x^i the (East, West) row number is i. Let t_S denote the (East, West) row number of the processor at which the S-probe resides. The S-probe ends with $t_S = 2n-2$. Similarly, t_N denotes the (East, West) row number of the processor at which the N-probe resides. The N-probe ends with $t_N = 0$.

Let R_y^j for $0 \leq j \leq 2n-2$ denote the (South-East, North-West) rows of processors of the H-mesh of size n where R_y^0 and R_y^{2n-2} respectively correspond to the South-West-Edge and the North-East-Edge. The index j denotes the row number of the (South-East, North-West) row R_y^j. For every processor in R_y^j the (South-East, North-West) row number is j. Let t_{SW} denote the (South-East, North-West) row number of the processor at which the SW-probe resides. The SW-probe ends with $t_{SW} = 2n-2$. Similarly, t_{NE} denotes the (South-East, North-West) row number of the processor at which the NE-probe resides. The NE-probe ends with $t_{NE} = 0$.

Let R_z^k for $0 \leq k \leq 2n-2$ denote the (North-East, South-West) rows of processors of the H-mesh of size n where R_z^0 and R_z^{2n-2} respectively correspond to the South-East-Edge and the North-West-Edge. The index k denotes the row number of the (North-East, South-West) row R_z^k. For every processor in R_z^k the (North-East, South-

West) row number is k. Let t_{SE} denote the (North-East, South-West) row number of the processor at which the SE-probe resides. The SE-probe ends with $t_{SE} = 2n - 2$. Similarly, t_{NW} denotes the (North-East, South-West) row number of the processor at which the NW-probe resides. The NW-probe ends with $t_{NW} = 0$.

The invariants for the probes are given by:

$$P_S \equiv \forall i : 0 \leq i < t_S : \text{row } R_x^i \text{ of processors is passive}$$

$$P_N \equiv \forall i : t_N < i \leq 2n-2 : \text{row } R_x^i \text{ of processors is passive}$$

$$P_{SW} \equiv \forall j : 0 \leq j < t_{SW} : \text{row } R_y^j \text{ of processors is passive}$$

$$P_{NE} \equiv \forall j : t_{NE} < j \leq 2n-2 : \text{row } R_y^j \text{ of processors is passive}$$

$$P_{SE} \equiv \forall k : 0 \leq k < t_{SE} : \text{row } R_z^k \text{ of processors is passive}$$

$$P_{NW} \equiv \forall k : t_{NW} < k \leq 2n-2 : \text{row } R_z^k \text{ of processors is passive}$$

We can design traffic rules for the probes so as to keep P_S, P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW} invariant, which is given in the next section.

The following definitions are used in the description of the proposed termination detection algorithm in section 4.4.

- **computing**(P) is a boolean function intended to express the notion of whether a processor P is working on its assigned task (returns TRUE) or not (returns FALSE).

- **data_msg_received**(P) is a boolean function intended to express the notion of whether a processor P received a *data* message from any of its neighbours (returns TRUE) or not (returns FALSE).

- **East-link** of P refers to the communication link connecting P to its neighbour on the east. Similar convention applies to West-link, South-East-link, South-West-link, North-East-link and North-West-link.

- **has_probe**(P) is a boolean function intended to express the notion of whether P has at least one probe (returns TRUE) or not (returns FALSE).

- **neighbour**(P)[East-link] returns the state of processor P's east neighbour. Similar definition applies to neighbour(P)[West-link],neighbour(P)[North-East-link], neighbour(P)[South-West-link],neighbour(P)[North-West-link] and neighbour(P)[South-East-link].

- **neighbour_reactivated**(P)[East-link] is a boolean function intended to express the notion of whether P reactivated its passive east neighbour (returns TRUE) or not (returns FALSE). Similar definition applies to neighbour_reactivated(P) [West-link], neighbour_reactivated(P) [North-East-link], neighbour_reactivated(P) [South-West-link], neighbour_reactivated(P) [North-West-link] and neighbour_reactivated(P) [South-East-link].

- **North-East-link**(P) is a boolean function intended to express the notion of whether a peripheral processor P has the north east neighbour (returns TRUE) or not (returns nil). Similar definition applies to South-West-link(P), North-West-link(P) and South-East-link(P).

- **opposite_edge**(P) returns the edge which is opposite to the edge to which the peripheral processor P belongs to. If $P \in$ South-Edge then opposite_edge(P) is defined to be North-Edge. Similar definition applies to P when P is in North-

Edge, South-East-Edge, South-West-Edge, North-East-Edge or North-West-Edge. Note that in the case where a peripheral processor P belongs to two edges opposite_edge(P) returns two opposite edges. There are six peripheral processors in a H-mesh which belongs to two edges of the H-mesh.

- **originated_edge($\mathcal{P}$)** returns the originated edge of a probe $\mathcal{P}$. The originated edges of S-probe, N-probe, SW-probe, NE-probe, SE-probe and NW-probe are defined to be South-Edge, North-Edge, South-West-Edge, North-East-Edge, South-East-Edge and North-West-Edge respectively.

- **probe(P)** returns the probe a processor P has with it. Note that when P has more than one probe with it, probe(P) returns arbitrarily one of its probes.

- **probe_count(P)** returns the number of probes that have visited a processor P.

- **send**(link, msg) is a communication primitive used for sending a message 'msg' on the communication link 'link'.

- **set_of_links** = { East-link, West-link, North-East-link, South-West-link, North-West-link, South-East-link }.

- **software_load(P)** is a boolean function intended to express the notion of whether the underlying computation has begun (returns TRUE) or not (returns FALSE).

- **South-Edge** refers to the set of processors which are on the South-Edge of the H-mesh. Similar convention applies to North-Edge, South-West-Edge, North-East-Edge, South-East-Edge and North-West-Edge.

- **state(P)** returns the state of a processor P which is either passive or active.

4.4 Termination Detection Algorithm

The proposed termination detection algorithm consists of specifying the following four sets of rules:

A) Processor State Transition Rules

B) Passive State Propagation Rules

C) Probe Propagation Rules

D) Termination Detection Criteria

To begin with we restrict to the case of detecting the termination of computations in which no *data* messages are present. In section 4.4.2, we handle the case where *data* messages are present. Passive state information and the probes are *control* messages.

4.4.1 Absence of Data Messages

The consequence of the absence of *data* messages in the underlying computation is that an active processor can become passive, but a passive processor can not become active again.

A) Processor State Transition Rules

A processor is in one of two states; namely, active and passive. We describe below the state transition rules with respect to peripheral processors and non-peripheral processors.

A.1) Peripheral Processors

Peripheral processors are assumed to be passive.

A.2) Non-peripheral Processors

There are three rules which describe the state transition of a non-peripheral processor P. Transition rules (A.2.1), (A.2.2) and (A.2.3) are intended to express respectively:

i) A passive processor P becomes active when the underlying computation begins.

ii) An active processor P remains active until its assigned task is completed.

iii) An active processor P becomes passive if it is completed its assigned task.

A.2.1) **If** state(P) = passive $\wedge$ software_load(P)
Then state(P) $\longleftarrow$ active.

A.2.2) **If** state(P) = active $\wedge$ computing(P)
Then state(P) $\longleftarrow$ active.

A.2.3) **If** state(P) = active $\wedge$ $\neg$ computing(P)
Then state(P) $\longleftarrow$ passive.

B) Passive State Propagation Rules

The objective is to propagate the passive state information along the rows; namely, (East, West) row, (South-East, North-West) row and (North-East, South-West) row. The passive state propagation rules with respect to peripheral and non-peripheral processors is described below.

B.1) Peripheral Processors

Peripheral processors send their passive state information to their neighbours after the computation has begun.

B.2) Non-peripheral Processors

Passive non-peripheral processors propagate the passive state information received from their neighbours in the opposite directions; i.e., a passive non-peripheral processor x receives passive state information from its North-East neighbour then the processor x sends passive state information to its South-West neighbour. Note that for the set of possible directions – { East, South-East, South-West, West, North-West, North-East }, the opposite directions are defined as { West, North-West, North-East, East, South-East, South-West }.

Active non-peripheral processors do not participate in the propagation of passive state information until they become passive. That is, passive state information received from its neighbours when a non-peripheral processor is active remain with it and are propagated in the opposite directions only after it becomes passive. The order of the propagation of passive state information is arbitrary.

Rule (B.1) assures the starting of the termination detection process. Later on in this section, we prove certain properties (Lemmas 4.1 - 4.18 and Corollaries 4.1 - 4.3) related to the passive state propagation rules (B.1) and (B.2). These properties provide a formal basis for the derivation of probe propagation rules which are presented below.

C) Probe Propagation Rules

There are two ways by which a given probe on a given row can be propagated; namely, i) along the given row and ii) across the given row (i.e., the next row parallel to the given row). In method i) the corresponding invariants (P_S and P_N) or (P_{SW} and P_{NE}) or (P_{SE} and P_{NW}) will not be falsified. However in method ii) the corresponding invariants (P_S and P_N) or (P_{SW} and P_{NE}) or (P_{SE} and P_{NW}) can be falsified if

not all processors in the given row are passive. Depending upon the passive state information available to the processor at which a probe resides either method i) or method ii) can be chosen. More specifically, method i) is used when it can not be inferred that all processors in the given row are passive and method ii) is used when it can be inferred that all processors in the given row are passive. In fact, properties established in Lemmas 4.1 - 4.18 and Corollaries 4.1 - 4.3 provide a formal basis for choosing between these two methods of probe propagation. Theorem 4.1 establishes that the following probe propagation rules keep P_S, P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW} invariant.

Six probes (i.e., S-probe, N-probe, SW-probe, NE-probe, SE-probe and NW-probe) are initially located on peripheral processors respectively belonging to six edges (i.e., South-Edge, North-Edge, South-West-Edge, North-East-Edge, South-East-Edge and North-West-Edge). Since, a peripheral processor may belong to two edges, it may contain two probes initially. In fact, the initial location of probes is arbitrary as long as they are on their corresponding edges; i.e., S-probe is on South-Edge, N-probe is on North-Edge, SW-probe is on South-West-Edge, NE-probe is on North-East-Edge, SE-probe is on South-East-Edge and NW-probe is on North-West-Edge.

Probe propagation rules with respect to peripheral processors and non-peripheral processors are described below.

C.1) Peripheral Processors

Peripheral processors perform two distinct tasks related to the propagation of probes; namely,

a) Initiation of the probe(s)

b) Response to the receipt of a probe

C.1a) Initiation of the probe(s)

After the computation has begun, if a peripheral processor P has a probe then it is sent to its neighbour as specified below. In the case where a peripheral processor has two probes, both probes are transmitted using the following specification and the order of transmission of the probes is arbitrary.

if has_probe(P) $\longrightarrow$

  if probe(P) = S-probe $\wedge$ $P \in$ South-Edge

    $\longrightarrow$ **send** (North-East-link, S-probe)

  $\square$ probe(P) = N-probe $\wedge$ $P \in$ North-Edge

    $\longrightarrow$ **send** (South-West-link, N-probe)

  $\square$ probe(P) = SW-probe $\wedge$ $P \in$ South-West-Edge

    $\longrightarrow$ **send** (North-East-link, SW-probe)

  $\square$ probe(P) = NE-probe $\wedge$ $P \in$ North-East-Edge

    $\longrightarrow$ **send** (South-West-link, NE-probe)

  $\square$ probe(P) = SE-probe $\wedge$ $P \in$ South-East-Edge

    $\longrightarrow$ **send** (North-West-link, SE-probe)

  $\square$ probe(P) = NW-probe $\wedge$ $P \in$ North-West-Edge

    $\longrightarrow$ **send** (South-East-link, NW-probe)

  fi

$\square$ $\neg$ has_probe(P) $\longrightarrow$ **skip**

fi

C.1b) Response to the receipt of a probe

We specify below the response of a peripheral processor P (which is to propagate

the probe) when a probe is received. For the sake of clarity, we specify the response to each event (i.e., the receipt of each probe) separately.

C.1b.1) S-probe is received

```
if      P ∉ North-Edge ⟶
        if      P ∈ South-West-Edge ⋃ North-West-Edge
                ⟶ if      neighbour(P) [East-link] = passive
                          ⟶ send (North-East-link, S-probe)
                  □ neighbour(P) [East-link] ≠ passive
                          ⟶ send (East-link, S-probe)
                fi
          □ P ∈ South-East-Edge ⋃ North-East-Edge
                ⟶ if      neighbour(P) [West-link] = passive
                          ⟶ if      North-East-link(P) = nil
                                    ⟶ send (North-West-link, S-probe)
                            □ North-East-link(P) ≠ nil
                                    ⟶ send (North-East-link, S-probe)
                            fi
                  □ neighbour(P) [West-link] ≠ passive
                          ⟶ send (West-link, S-probe)
                fi
        fi
    □ P ∈ North-Edge ⟶ skip /* termination detected */
fi
```

C.1b.2) N-probe is received

```
if      P ∉ South-Edge ⟶
        if      P ∈ North-East-Edge ⋃ South-East-Edge
                ⟶ if      neighbour(P) [West-link] = passive
                          ⟶ send (South-West-link, N-probe)
                  □ neighbour(P) [West-link] ≠ passive
                          ⟶ send (West-link, N-probe)
                fi
          □ P' ∈ North-West-Edge ⋃ South-West-Edge
                ⟶ if      neighbour(P) [East-link] = passive
                          ⟶ if      South-West-link(P) = nil
                                    ⟶ send (South-East-link, N-probe)
                            □ South-West-link(P) ≠ nil
                                    ⟶ send (South-West-link, N-probe)
                            fi
                  □ neighbour(P) [East-link] ≠ passive
                          ⟶ send (East-link, N-probe)
                fi
        fi
  □ P ∈ South-Edge ⟶ skip /* termination detected */
fi
```

C.1b.3) SW-probe is received

```
if    P ∉ North-East-Edge ⟶
      if    P ∈ South-Edge ⋃ South-East-Edge
            ⟶ if    neighbour(P) [North-West-link] = passive
                    ⟶ send (North-East-link, SW-probe)
              □ neighbour(P) [North-West-link] ≠ passive
                    ⟶ send (North-West-link, SW-probe)
            fi
        □ P ∈ North-West-Edge ⋃ North-Edge
            ⟶ if    neighbour(P) [South-East-link] = passive
                    ⟶ if    North-East-link(P) = nil
                            ⟶ send (East-link, SW-probe)
                      □ North-East-link(P) ≠ nil
                            ⟶ send (North-East-link, SW-probe)
                    fi
              □ neighbour(P) [South-East-link] ≠ passive
                    ⟶ send (South-East-link, S-probe)
            fi
      fi
  □ P ∈ North-East-Edge ⟶ skip /* termination detected */
fi
```

C.1b.4) NE-probe is received

```
if      P ∉ South-West-Edge ⟶
        if      P ∈ North-Edge ⋃ North-West-Edge
                ⟶ if      neighbour(P) [South-East-link] = passive
                          ⟶ send (South-West-link, NE-probe)
                  □ neighbour(P) [South-East-link] ≠ passive
                          ⟶ send (South-East-link, NE-probe)
                fi
          □ P ∈ South-East-Edge ⋃ South-Edge
                ⟶ if      neighbour(P) [North-West-link] = passive
                          ⟶ if      South-West-link(P) = nil
                                    ⟶ send (West-link, NE-probe)
                            □ South-West-link(P) ≠ nil
                                    ⟶ send (South-West-link, NE-probe)
                            fi
                  □ neighbour(P) [North-West-link] ≠ passive
                          ⟶ send (North-West-link, S-probe)
                fi
        fi
   □ P ∈ South-West-Edge ⟶ skip /* termination detected */
fi
```

C.1b.5) SE-probe is received

```
if    P ∉ North-West-Edge ⟶
      if    P ∈ South-Edge ⋃ South-West-Edge
            ⟶ if    neighbour(P) [North-East-link] = passive
                    ⟶ send (North-West-link, SE-probe)
              □ neighbour(P) [North-East-link] ≠ passive
                    ⟶ send (North-East-link, SE-probe)
            fi
        □ P ∈ North-East-Edge ⋃ North-Edge
            ⟶ if    neighbour(P) [South-West-link] = passive
                    ⟶ if    North-West-link(P) = nil
                            ⟶ send (West-link, SE-probe)
                      □ North-West-link(P) ≠ nil
                            ⟶ send (North-West-link, SE-probe)
                    fi
              □ neighbour(P) [South-West-link] ≠ passive
                    ⟶ send (South-West-link, SE-probe)
            fi
      fi
  □ P ∈ North-West-Edge ⟶ skip /* termination detected */
fi
```

C.1b.6) NW-probe is received

```
if    P ∉ South-East-Edge ⟶
      if    P ∈ North-Edge ⋃ North-East-Edge
            ⟶ if    neighbour(P) [South-West-link] = passive
                    ⟶ send (South-East-link, NW-probe)
              □ neighbour(P) [South-West-link] ≠ passive
                    ⟶ send (South-West-link, NW-probe)
            fi
        □ P ∈ South-West-Edge ⋃ South-Edge
            ⟶ if    neighbour(P) [North-East-link] = passive
                    ⟶ if    South-East-link(P) = nil
                            ⟶ send (East-link, NW-probe)
                      □ South-East-link(P) ≠ nil
                            ⟶ send (South-East-link, NW-probe)
                    fi
              □ neighbour(P) [North-East-link] ≠ passive
                    ⟶ send (North-East-link, NW-probe)
            fi
      fi
  □ P ∈ South-East-Edge ⟶ skip /* termination detected */
fi
```

C.2) Non-peripheral Processors

Active non-peripheral processors do not participate in the propagation of probes until they become passive. That is, probes received when a non-peripheral processor P is active remain with P and are propagated according to the following specifications only after P becomes passive. The order of the propagation of probes is arbitrary.

C.2.1) S-probe is received

if neighbour(P) [East-link] = passive $\wedge$
neighbour(P) [West-link] $\neq$ passive
$\longrightarrow$ **send** (West-link, S-probe)

$\square$ neighbour(P) [East-link] $\neq$ passive $\wedge$
neighbour(P) [West-link] = passive
$\longrightarrow$ **send** (East-link, S-probe)

$\square$ neighbour(P) [East-link] = passive $\wedge$
neighbour(P) [West-link] = passive
$\longrightarrow$ **send** (North-East-link, S-probe)

$\square$ neighbour(P) [East-link] $\neq$ passive $\wedge$
neighbour(P) [West-link] $\neq$ passive
$\longrightarrow$ **skip**

fi

C.2.2) N-probe is received

if neighbour(P) [East-link] = passive $\wedge$

neighbour(P) [West-link] $\neq$ passive

$\longrightarrow$ **send** (West-link, N-probe)

$\square$ neighbour(P) [East-link] $\neq$ passive $\wedge$

neighbour(P) [West-link] = passive

$\longrightarrow$ **send** (East-link, N-probe)

$\square$ neighbour(P) [East-link] = passive $\wedge$

neighbour(P) [West-link] = passive

$\longrightarrow$ **send** (South-West-link, N-probe)

$\square$ neighbour(P) [East-link] $\neq$ passive $\wedge$

neighbour(P) [West-link] $\neq$ passive

$\longrightarrow$ **skip**

fi

C.2.3) SW-probe is received

if neighbour(P) [North-West-link] = passive $\wedge$

neighbour(P) [South-East-link] $\neq$ passive

$\longrightarrow$ **send** (South-East-link, SW-probe)

$\square$ neighbour(P) [North-West-link] $\neq$ passive $\wedge$

neighbour(P) [South-East-link] = passive

$\longrightarrow$ **send** (North-West-link, SW-probe)

$\square$ neighbour(P) [North-West-link] = passive $\wedge$

neighbour(P) [South-East-link] = passive

$\longrightarrow$ **send** (North-East-link, SW-probe)

□ neighbour(P) [North-West-link] $\neq$ passive $\wedge$
neighbour(P) [South-East-link] $\neq$ passive
$\longrightarrow$ **skip**

fi

C.2.4) NE-probe is received

if neighbour(P) [North-West-link] = passive $\wedge$
neighbour(P) [South-East-link] $\neq$ passive
$\longrightarrow$ **send** (South-East-link, NE-probe)
□ neighbour(P) [North-West-link] $\neq$ passive $\wedge$
neighbour(P) [South-East-link] = passive
$\longrightarrow$ **send** (North-West-link, NE-probe)
□ neighbour(P) [North-West-link] = passive $\wedge$
neighbour(P) [South-East-link] = passive
$\longrightarrow$ **send** (South-West-link, NE-probe)
□ neighbour(P) [North-West-link] $\neq$ passive $\wedge$
neighbour(P) [South-East-link] $\neq$ passive
$\longrightarrow$ **skip**

fi

C.2.5) SE-probe is received

if neighbour(P) [North-East-link] = passive $\wedge$

neighbour(P) [South-West-link] $\neq$ passive

$\longrightarrow$ **send** (South-West-link, SE-probe)

□ neighbour(P) [North-East-link] $\neq$ passive $\wedge$

neighbour(P) [South-West-link] = passive

$\longrightarrow$ **send** (North-East-link, SE-probe)

□ neighbour(P) [North-East-link] = passive $\wedge$

neighbour(P) [South-West-link] = passive

$\longrightarrow$ **send** (North-West-link, SE-probe)

□ neighbour(P) [North-East-link] $\neq$ passive $\wedge$

neighbour(P) [South-West-link] $\neq$ passive

$\longrightarrow$ **skip**

fi

C.2.6) NW-probe is received

if neighbour(P) [North-East-link] = passive $\wedge$

neighbour(P) [South-West-link] $\neq$ passive

$\longrightarrow$ **send** (South-West-link, NW-probe)

□ neighbour(P) [North-East-link] $\neq$ passive $\wedge$

neighbour(P) [South-West-link] = passive

$\longrightarrow$ **send** (North-East-link, NW-probe)

□ neighbour(P) [North-East-link] = passive $\wedge$

neighbour(P) [South-West-link] = passive

$\longrightarrow$ **send** (South-East-link, NW-probe)

□ neighbour(P) [North-East-link] $\neq$ passive $\wedge$

neighbour(P) [South-West-link] $\neq$ passive

$\longrightarrow$ **skip**

fi

D) Termination Detection Criteria

We specify below sufficiency conditions for peripheral processors and non-peripheral processors to infer that the computation is terminated.

D.1) A peripheral processor x can infer that the computation is terminated if it receives a probe p such that originated_edge(p) = opposite_edge(x).

D.2) A non-peripheral passive processor y can infer that the computation is terminated if probe_count(y) = 6.

Recall that in the case where a peripheral processor x belongs to two edges of the H-mesh, opposite_edge(x) returns two opposite edges. There are six peripheral processors in a H-mesh which belong to two edges of the H-mesh. For such peripheral processors, the equality relationship originated_edge(p) = opposite_edge(x) in rule (D.1) above is said to hold if it holds for any one of the two values returned by the opposite_edge(x). The correctness of criteria (D.1) and (D.2) are demonstrated in section 4.5.

We establish below certain properties related to the passive state propagation rules (B.1) and (B.2) which facilitate the proof of invariance of P_S, P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW}.

Lemma 4.1: Let R_x denote the set of processors in a (East, West) row (other than North-Edge and South-Edge) of processors of a H-mesh. Suppose $u \in R_x$ be a non-peripheral processor such that neighbour(u) [East-link] = passive and neighbour(u) [West-link] = passive. Let $R'_x = \{v \in R_x : v \neq u\}$ (i.e., $R'_x = R_x - \{u\}$). Then for every $w \in R'_x$ state(w) = passive.

Proof: Let $R_x = \{p_1, p_2, \cdots, p_{m-1}, p_m\}$. By definition two processors in R_x are peripheral processors and the rest are non-peripheral processors. Without loss of generality, let p_1 and p_m be peripheral processors, and p_i for $2 \leq i \leq m-1$ be non-peripheral processors. Furthermore, assume that for every $i \in [2, m-1]$ p_{i-1} is the west neighbour of p_i and p_{i+1} is the east neighbour of p_i, p_2 is the east neighbour of p_1 and p_{m-1} is the west neighbour of p_m.

From Rules - (B.1) and (B.2), the passive state propagation is carried out along two directions; namely, from p_1 to p_m along the east links (i.e., $p_1 \rightarrow p_2 \rightarrow \cdots p_{i-1} \rightarrow p_i \rightarrow p_{i+1} \rightarrow \cdots \rightarrow p_m$) and from p_m to p_1 along the west links (i.e., $p_1 \leftarrow p_2 \leftarrow \cdots p_{i-1} \leftarrow p_i \leftarrow p_{i+1} \leftarrow \cdots \leftarrow p_m$). From the assumption, there exists $u \in \{p_2, p_3, \cdots, p_{i-1}, p_i, p_{i+1}, \cdots, p_{m-1}\}$ such that neighbour(u) [East-link] = neighbour(u) [West-link] = passive, then this implies that $\forall w \in R^l_x = \{p_1, \cdots, u\} - \{u\}$, w is passive and $\forall w' \in R^r_x = \{u, \cdots, p_m\} - \{u\}$, w' is passive. Observe that $R'_x = R^l_x \bigcup R^r_x$.

□

Corollary 4.1: Let R_x denote the set of processors in a (East, West) row (other than North-Edge and South-Edge) of processors of a H-mesh. Suppose $u \in R_x$ be a non-peripheral processor such that state(u) = passive, neighbour(u) [East-link] = passive and neighbour(u) [West-link] = passive. Then for every $w \in R_x$ state(w) = passive.

Proof: Follows directly from Lemma 4.1 and state(u) = passive. □

Lemma 4.2: Let R_x denote the set of processors in a (East, West) row (other than North-Edge and South-Edge) of processors of a H-mesh. Suppose $u \in R_x$ be a peripheral processor such that $u \in$ South-West-Edge $\bigcup$ North-West-Edge and neighbour(u) [East-link] = passive. Then for every $w \in R_x$ state(w) = passive.

Proof: Let $R_x = \{u = p_1, p_2, \cdots, p_{m-1}, p_m\}$. Without loss of generality, let p_m be the other peripheral processor, and p_i for $2 \leq i \leq m-1$ be non-peripheral processors. Furthermore, assume that for every $i \in [2, m-1]$ p_{i-1} is the west neighbour of p_i and p_{i+1} is the east neighbour of p_i, p_2 is the east neighbour of p_1 and p_{m-1} is the west neighbour of p_m.

Since neighbour(p_1) [East-link] = passive, then this implies from passive state propagation rules - (B.1) and (B.2) that for every $w \in \{p_2, \cdots, p_m\}$ state(w) = passive. Furthermore, from Rule (B.1) state(p_1) = passive. Therefore, for every $w \in R_x$ state(w) = passive. □

Lemma 4.3: Let R_x denote the set of processors in a (East, West) row (other than North-Edge and South-Edge) of processors of a H-mesh. Suppose $u \in R_x$ be a peripheral processor such that $u \in$ South-East-Edge $\bigcup$ North-East-Edge and neighbour(u) [West-link] = passive. Then for every $w \in R_x$ state(w) = passive.

Proof: Let $R_x = \{p_1, p_2, \cdots, p_{m-1}, p_m = u\}$. Without loss of generality, let p_1 be the other peripheral processor, and p_i for $2 \leq i \leq m-1$ be non-peripheral processors. Furthermore, assume that for every $i \in [2, m-1]$ p_{i-1} is the west neighbour of p_i and p_{i+1} is the east neighbour of p_i, p_2 is the east neighbour of p_1 and p_{m-1} is the west neighbour of p_m.

Since neighbour(p_m) [West-link] = passive, then this implies from passive state propagation rules - (B.1) and (B.2) that for every $w \in \{p_1, \cdots, p_{m-1}\}$ state(w) = passive. Furthermore, from Rule (B.1) state(p_m) = passive. Therefore, for every $w \in R_x$ state(w) = passive. □

Lemma 4.4: Let R_x denote the set of processors in a (East, West) row (other than North-Edge and South-Edge) of processors of a H-mesh. Suppose $u \in R_x$ be a non-peripheral processor such that state(u) = active. Then there does not exist a non-peripheral processor $v \neq u$ in R_x such that neighbour(v) [East-link] = passive and neighbour(v) [West-link] = passive.

Proof: Let $R_x = \{p_1, p_2, \cdots, p_{m-1}, p_m\}$. By definition two processors in R_x are peripheral processors and the rest are non-peripheral processors. Without loss of generality, let p_1 and p_m be peripheral processors, and p_i for $2 \leq i \leq m-1$ be non-peripheral processors. Furthermore, assume that for every $i \in [2, m-1]$ p_{i-1} is the west neighbour of p_i and p_{i+1} is the east neighbour of p_i, p_2 is the east neighbour of p_1 and p_{m-1} is the west neighbour of p_m.

From Rules - (B.1) and (B.2), the passive state propagation is carried out along two directions; namely, from p_1 to p_m along the east links (i.e., $p_1 \rightarrow p_2 \rightarrow \cdots p_{i-1} \rightarrow p_i \rightarrow p_{i+1} \rightarrow \cdots \rightarrow p_m$) and from p_m to p_1 along the west links (i.e., $p_1 \leftarrow p_2 \leftarrow \cdots p_{i-1} \leftarrow p_i \leftarrow p_{i+1} \leftarrow \cdots \leftarrow p_m$). Since for $u \in \{p_2, \cdots, p_{m-1}\}$ state(u) = active, this implies that the passive state propagation along the east links will not be carried out beyond u towards p_m until state(u) = passive. Similarly, the passive state propagation along the west links will not be carried out beyond u towards p_1 until state(u) = passive. Therefore, for every $w \in \{p_1, \cdots, u\} - \{u\}$ neighbour(w) [East-link] $\neq$ passive and also for every $w' \in \{u, \cdots, p_m\} - \{u\}$ neighbour(w') [West-link] $\neq$ passive. Hence,

in particular there does not exist a non-peripheral processor $v \neq u$ in R_x such that neighbour(v) [East-link] = passive and neighbour(v) [West-link] = passive. □

Lemma 4.5: Let R_x denote the set of processors in a (East, West) row (other than North-Edge and South-Edge) of processors of a H-mesh. Let $u \in R_x$ be a peripheral processor such that $u \in$ South-West-Edge $\bigcup$ North-West-Edge. Suppose $v \in R_x$ be a non-peripheral processor such that state(v) = active. Then neighbour(u) [East-link] $\neq$ passive.

Proof: Let $R_x = \{u = p_1, p_2, \cdots, p_{m-1}, p_m\}$. Without loss of generality, let p_m be the other peripheral processor, and p_i for $2 \leq i \leq m-1$ be non-peripheral processors. Furthermore, assume that for every $i \in [2, m-1]$ p_{i-1} is the west neighbour of p_i and p_{i+1} is the east neighbour of p_i, p_2 is the east neighbour of p_1 and p_{m-1} is the west neighbour of p_m.

Since for $v \in \{p_2, \cdots, p_{m-1}\}$ state(v) = active, this implies from rules - (B.1) and (B.2) that the passive state propagation along the west links will not be carried out beyond v towards p_1 until state(v) = passive. Therefore, for every $w \in \{p_1, \cdots, v\} - \{v\}$ neighbour(w) [East-link] $\neq$ passive. Hence, in particular neighbour(p_1) [East-link] $\neq$ passive. □

Lemma 4.6: Let R_x denote the set of processors in a (East, West) row (other than North-Edge and South-Edge) of processors of a H-mesh. Let $u \in R_x$ be a peripheral processor such that $u \in$ South-East-Edge $\bigcup$ North-East-Edge. Suppose $v \in R_x$ be a non-peripheral processor such that state(v) = active. Then neighbour(u) [West-link] $\neq$ passive.

Proof: Let $R_x = \{p_1, p_2, \cdots, p_{m-1}, p_m = u\}$. Without loss of generality, let p_1 be

the other peripheral processor, and p_i for $2 \leq i \leq m-1$ be non-peripheral processors. Furthermore, assume that for every $i \in [2, m-1]$ p_{i-1} is the west neighbour of p_i and p_{i+1} is the east neighbour of p_i, p_2 is the east neighbour of p_1 and p_{m-1} is the west neighbour of p_m.

Since for $v \in \{p_2, \cdots, p_{m-1}\}$ state(v) = active, this implies from rules - (B.1) and (B.2) that the passive state propagation along the east links will not be carried out beyond v towards p_m until state(v) = passive. Therefore, for every $w \in \{v, \cdots, p_m\} - \{v\}$ neighbour(w) [West-link] $\neq$ passive. Hence, in particular neighbour(p_m) [West-link] $\neq$ passive. □

Lemma 4.7: Let R_y denote the set of processors in a (South-East, North-West) row (other than South-West-Edge and North-East-Edge) of processors of a H-mesh. Suppose $u \in R_y$ be a non-peripheral processor such that neighbour(u) [South-East-link] = passive and neighbour(u) [North-West-link] = passive. Let $R'_y = \{v \in R_y : v \neq u\}$ (i.e., $R'_y = R_y - \{u\}$). Then for every $w \in R'_y$ state(w) = passive.

Proof: Similar to that of Lemma 4.1 □

Corollary 4.2: Let R_y denote the set of processors in a (South-East, North-West) row (other than South-West-Edge and North-East-Edge) of processors of a H-mesh. Suppose $u \in R_y$ be a non-peripheral processor such that state(u) = passive, neighbour(u) [South-East-link] = passive and neighbour(u) [North-West-link] = passive. Then for every $w \in R_y$ state(w) = passive.

Proof: Similar to that of Corollary 4.1. □

Lemma 4.8: Let R_y denote the set of processors in a (South-East, North-West)

row (other than South-West-Edge and North-East-Edge) of processors of a H-mesh. Suppose $u \in R_y$ be a peripheral processor such that $u \in$ North-West-Edge $\bigcup$ North-Edge and neighbour(u) [South-East-link] = passive. Then for every $w \in R_y$ state(w) = passive.

Proof: Similar to that of Lemma 4.2. □

Lemma 4.9: Let R_y denote the set of processors in a (South-East, North-West) row (other than South-West-Edge and North-East-Edge) of processors of a H-mesh. Suppose $u \in R_y$ be a peripheral processor such that $u \in$ South-East-Edge $\bigcup$ South-Edge and neighbour(u) [North-West-link] = passive. Then for every $w \in R_y$ state(w) = passive.

Proof: Similar to that of Lemma 4.3. □

Lemma 4.10: Let R_y denote the set of processors in a (South-East, North-West) row (other than South-West-Edge and North-East-Edge) of processors of a H-mesh. Suppose $u \in R_y$ be a non-peripheral processor such that state(u) = active. Then there does not exist a non-peripheral processor $v \neq u$ in R_y such that neighbour(v) [South-East-link] = passive and neighbour(v) [North-West-link] = passive.

Proof: Similar to that of Lemma 4.4. □

Lemma 4.11: Let R_y denote the set of processors in a (South-East, North-West) row (other than South-West-Edge and North-East-Edge) of processors of a H-mesh. Let $u \in R_y$ be a peripheral processor such that $u \in$ North-West-Edge $\bigcup$ North-Edge. Suppose $v \in R_y$ be a non-peripheral processor such that state(v) = active. Then neighbour(u) [South-East-link] $\neq$ passive.

Proof: Similar to that Lemma 4.5. □

Lemma 4.12: Let R_y denote the set of processors in a (South-East, North-West) row (other than South-West-Edge and North-East-Edge) of processors of a H-mesh. Let $u \in R_y$ be a peripheral processor such that $u \in$ South-East-Edge $\bigcup$ South-Edge. Suppose $v \in R_y$ be a non-peripheral processor such that state(v) = active. Then neighbour(u) [North-West-link] $\neq$ passive.

Proof: Similar to that of Lemma 4.6. □

Lemma 4.13: Let R_z denote the set of processors in a (North-East, South-West) row (other than South-East-Edge and North-West-Edge) of processors of a H-mesh. Suppose $u \in R_z$ be a non-peripheral processor such that neighbour(u) [North-East-link] = passive and neighbour(u) [South-West-link] = passive. Let $R'_z = \{v \in R_z : v \neq u\}$ (i.e., $R'_z = R_z - \{u\}$). Then for every $w \in R'_z$ state(w) = passive.

Proof: Similar to that of Lemma 4.1. □

Corollary 4.3: Let R_z denote the set of processors in a (North-East, South-West) row (other than South-East-Edge and North-West-Edge) of processors of a H-mesh. Suppose $u \in R_z$ be a non-peripheral processor such that state(u) = passive, neighbour(u) [North-East-link] = passive and neighbour(u) [South-West-link] = passive. Then for every $w \in R_z$ state(w) = passive.

Proof: Similar to that of Corollary 4.1. □

Lemma 4.14: Let R_z denote the set of processors in a (North-East, South-West) row (other than South-East-Edge and North-West-Edge) of processors of a H-mesh. Suppose $u \in R_z$ be a peripheral processor such that $u \in$ South-West-Edge $\bigcup$ South-

Edge and neighbour(u) [North-East-link] = passive. Then for every $w \in R_z$ state(w) = passive.

Proof: Similar to that of Lemma 4.2. □

Lemma 4.15: Let R_z denote the set of processors in a (North-East, South-West) row (other than South-East-Edge and North-West-Edge) of processors of a H-mesh. Suppose $u \in R_z$ be a peripheral processor such that $u \in$ North-East-Edge $\bigcup$ North-Edge and neighbour(u) [South-West-link] = passive. Then for every $w \in R_z$ state(w) = passive.

Proof: Similar to that of Lemma 4.3. □

Lemma 4.16: Let R_z denote the set of processors in a (North-East, South-West) row (other than South-East-Edge and North-West-Edge) of processors of a H-mesh. Suppose $u \in R_z$ be a non-peripheral processor such that state(u) = active. Then there does not exist a non-peripheral processor $v \neq u$ in R_z such that neighbour(v) [North-East-link] = passive and neighbour(v) [South-West-link] = passive.

Proof: Similar to that of Lemma 4.4. □

Lemma 4.17: Let R_z denote the set of processors in a (North-East, South-West) row (other than South-East-Edge and North-West-Edge) of processors of a H-mesh. Let $u \in R_z$ be a peripheral processor such that $u \in$ South-West-Edge $\bigcup$ South-Edge. Suppose $v \in R_z$ be a non-peripheral processor such that state(v) = active. Then neighbour(u) [North-East-link] $\neq$ passive.

Proof: Similar to that Lemma 4.5. □

Lemma 4.18: Let R_z denote the set of processors in a (North-East, South-West) row (other than South-East-Edge and North-West-Edge) of processors of a H-mesh. Let $u \in R_z$ be a peripheral processor such that $u \in$ North-East-Edge $\bigcup$ North-Edge. Suppose $v \in R_z$ be a non-peripheral processor such that state(v) = active. Then neighbour(u) [South-West-link] $\neq$ passive.

Proof: Similar to that of Lemma 4.6. □

Theorem 4.1: Traffic rules (C.1) and (C.2) for the probes maintain the invariance of P_S, P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW}.

Proof: The initiation of the probes (i.e., rule C.1a) establishes P_S, P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW} because of rule (B.1).

From Lemmas 4.1 - 4.6 and Corollary 4.1, traffic rules - (C.1b.1), (C.1b.2), (C.2.1) and (C.2.2) establish P_S and P_N.

From Lemmas 4.7 - 4.12 and Corollary 4.2, traffic rules - (C.1b.3), (C.1b.4), (C.2.3) and (C.2.4) establish P_{SW} and P_{NE}.

From Lemmas 4.13 - 4.18 and Corollary 4.3, traffic rules - (C.1b.5), (C.1b.6), (C.2.5) and (C.2.6) establish P_{SE} and P_{NW}.

Therefore, traffic rules - (C.1) and (C.2) for the probes maintain the invariance of P_S, P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW}. □

Observe that whenever $t_S = 2n-2$, $t_N = 0$, $t_{SW} = 2n-2$, $t_{NE} = 0$, $t_{SE} = 2n-2$, or $t_{NW} = 0$ establishes that all processors are passive. However in the presence of *data* messages P_S, P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW} can be falsified; for example, when a processor on the (East, West) row R_x^i with $t_N < i$ becomes active on account of the receipt of a *data* message falsifies P_N. Next, we present rules which will prevent P_S,

P_N, P_{SW}, P_{NE}, P_{SE} and P_{NW} from being falsified in the presence of *data* messages.

4.4.2 Presence of Data Messages

The consequence of the presence of *data* messages in the underlying computation is that a passive processor P_p is reactivated (i.e., becomes active again) on account of the receipt of a *data* message (from an active processor P_a; because only active processors send *data* messages). This situation can be handled by extending the behaviour of P_a and P_p as follows:

Processor P_a does not become passive until P_p becomes passive (again) (and correspondingly processor P_p informs P_a that it is passive again after completing the actions triggered by the receipt of the *data* message from P_a).

This modification ensures that neither P_a nor any other processor on three rows of processors (i.e., (East, West) row, (South-East, North-West) row and (North-East, South-West) row) where P_a resides, propagate the probes across three rows of processors unless P_p becomes passive again.

In light of this, processor state transition rules (A.2.1) and (A.2.3) are modified and a new transition rule (A.2.4) is added as follows:

A.2.1) **If** state(P) = passive $\wedge$ (software_load(P) $\vee$ data_msg_received(P))
Then state(P) $\longleftarrow$ active.

A.2.3) **If** state(P) = active $\wedge$ $\neg$ computing(P) $\wedge$ $\nexists$ link $\in$ set_of_links :
neighbour_reactivated(P) [link]
Then state(P) $\longleftarrow$ passive.

A.2.4) **If** state(P) = active $\wedge$ $\neg$ computing(P) $\wedge$ $\exists$ link $\in$ set_of_links :
neighbour_reactivated(P) [link]
Then state(P) $\longleftarrow$ active.

Furthermore, the passive state propagation rule is modified as follows:

B.3) If a passive processor P receives a *data* message from its neighbour P' and P has propagated its passive state information to P', then P acknowledges by sending its passive state information (again) after it becomes passive again (i.e., after completing the actions triggered by the receipt of the *data* message from P').

4.5 Correctness Proof

We show below the correctness of termination detection criteria (D.1) and (D.2) (section 4.3).

Theorem 4.2: Let x be a peripheral processor. Suppose x has received a probe p such that originated_edge(p) = opposite_edge(x). Then, the underlying computation is terminated.

Proof: Suppose that the underlying computation is not terminated. This implies that there exists a processor z such that state(z) = active. From Rule (A.1), z is not a peripheral processor.

Six cases to examine since probe p can be any one of the possible six probes.

Case (i): Let p = S-probe.

Since originated_edge(p) = opposite_edge(x) and by definition, originated_edge(S-probe) = South-Edge. This implies that $x \in$ North-Edge.

Let R_x^i denote the (East, West) row on which z resides. Since z is a non-peripheral processor then there exists at least one more (East, West) row of processors to the north of R_x^i. Let R_x^{i+1} denote the first (East, West) row to the north of R_x^i. Note that the North-Edge is considered to be the last (East, West) row to the north of R_x^i.

From Rules - (A.2.4), (B.3) and (C.2), S-probe can not be propagated to R_x^{i+1} until state(z) = passive. But from the assumption, $x \in$ North-Edge has received the S-probe which implies that state(z) = passive. This contradicts the assumption that state(z) = active.

Case (ii): Let p = N-probe.

Since originated_edge(p) = opposite_edge(x) and by definition, originated_edge(N-probe) = North-Edge. This implies that $x \in$ South-Edge.

Let R_x^i denote the (East, West) row on which z resides. Since z is a non-peripheral processor then there exists at least one more (East, West) row of processors to the south of R_x^i. Let R_x^{i-1} denote the first (East, West) row to the south of R_x^i. Note that the South-Edge is considered to be the last (East, West) row to the south of R_x^i.

From Rules - (A.2.4), (B.3) and (C.2), N-probe can not be propagated to R_x^{i-1} until state(z) = passive. But from the assumption, $x \in$ South-Edge has received the N-probe which implies that state(z) = passive. This contradicts the assumption that state(z) = active.

Case (iii): Let p = SW-probe.

Since originated_edge(p) = opposite_edge(x) and by definition, originated_edge(SW-probe) = South-West-Edge. This implies that $x \in$ North-East-Edge.

Let R_y^i denote the (South-East, North-West) row on which z resides. Since z is a non-peripheral processor then there exists at least one more (South-East, North-West) row of processors to the north east of R_y^i. Let R_y^{i+1} denote the first (South-East, North-West) row to the north east of R_y^i. Note that the North-East-Edge is considered to be the last (South-East, North-West) row to the north east of R_y^i.

From Rules - (A.2.4), (B.3) and (C.2), SW-probe can not be propagated to R_y^{i+1} until state(z) = passive. But from the assumption, $x \in$ North-East-Edge has received the SW-probe which implies that state(z) = passive. This contradicts the assumption that state(z) = active.

Case (iv): Let p = NE-probe.

Since originated_edge(p) = opposite_edge(x) and by definition, originated_edge(NE-probe) = North-East-Edge. This implies that $x \in$ South-West-Edge.

Let R_y^i denote the (South-East, North-West) row on which z resides. Since z is a non-peripheral processor then there exists at least one more (South-East, North-West) row of processors to the south west of R_y^i. Let R_y^{i-1} denote the first (South-East, North-West) row to the south west of R_y^i. Note that the South-West-Edge is considered to be the last (South-East, North-West) row to the south west of R_y^i.

From Rules - (A.2.4), (B.3) and (C.2), NE-probe can not be propagated to R_y^{i-1} until state(z) = passive. But from the assumption, $x \in$ South-West-Edge has received the NE-probe which implies that state(z) = passive. This contradicts the assumption that state(z) = active.

Case (v): Let p = SE-probe.

Since originated_edge(p) = opposite_edge(x) and by definition, originated_edge(SE-probe) = South-East-Edge. This implies that $x \in$ North-West-Edge.

Let R_z^i denote the (North-East, South-West) row on which z resides. Since z is a non-peripheral processor then there exists at least one more (North-East, South-West) row of processors to the north west of R_z^i. Let R_z^{i+1} denote the first (North-East, South-West) row to the north west of R_z^i. Note that the North-West-Edge is considered to be the last (North-East, South-West) row to the north west of R_z^i.

From Rules - (A.2.4), (B.3) and (C.2), SE-probe can not be propagated to R_z^{i+1} until state(z) = passive. But from the assumption, $x \in$ North-West-Edge has received the SE-probe which implies that state(z) = passive. This contradicts the assumption that state(z) = active.

Case (vi): Let p = NW-probe.

Since originated_edge(p) = opposite_edge(x) and by definition, originated_edge(NW-probe) = North-West-Edge. This implies that $x \in$ South-East-Edge.

Let R_z^i denote the (North-East, South-West) row on which z resides. Since z is a non-peripheral processor then there exists at least one more (North-East, South-West) row of processors to the south east of R_z^i. Let R_z^{i-1} denote the first (North-East, South-West) row to the south east of R_z^i. Note that the South-East-Edge is considered to be the last (North-East, South-West) row to the south east of R_z^i.

From Rules - (A.2.4), (B.3) and (C.2), NW-probe can not be propagated to R_z^{i-1} until state(z) = passive. But from the assumption, $x \in$ South-East-Edge has received the NW-probe which implies that state(z) = passive. This contradicts the assumption that state(z) = active. □

Theorem 4.3: Let y be a non-peripheral processor such that probe_count(y) = 6 and state(y) = passive. Then, the underlying computation is terminated.

Proof: Suppose that the underlying computation is not terminated. This implies that there exists a processor z such that state(z) = active. From Rule (A.1), z is not a peripheral processor.

From Rules - (A.2.4), (B.3) and (C.2), we have:

a) S-probe can not be propagated to any (East, West) row of processors to the north of z until state(z) = passive. Therefore, for any processor w in these rows of processors probe_count(w) < 6.

b) N-probe can not be propagated to any (East, West) row of processors to the south of z until state(z) = passive. Therefore, for any processor w in these rows

of processors probe_count(w) < 6.

c) SW-probe can not be propagated to any (South-East, North-West) row of processors to the north east of z until state(z) = passive. Therefore, for any processor w in these rows of processors probe_count(w) < 6.

d) NE-probe can not be propagated to any (South-East, North-West) row of processors to the south west of z until state(z) = passive. Therefore, for any processor w in these rows of processors probe_count(w) < 6.

e) SE-probe can not be propagated to any (North-East, South-West) row of processors to the north west of z until state(z) = passive. Therefore, for any processor w in these rows of processors probe_count(w) < 6.

f) NW-probe can not be propagated to any (North-East, South-West) row of processors to the south east of z until state(z) = passive. Therefore, for any processor w in these rows of processors probe_count(w) < 6.

From a), b), c), d), e), and f), this implies that there does not exist a processor w such that probe_count(w) = 6. This contradicts the assumption that probe_count(y) = 6. □

Chapter 5

Conclusions and Further Study

In this thesis, we developed leader election algorithms for complete networks and recursively scalable networks, and a termination detection algorithm in H-meshes.

Specifically, an hybrid approach to designing election algorithm in a complete network with a sense of direction is developed. The objective of this approach is to investigate the possibility of improving the message complexity of an existing election algorithm due to Loui, Matsushita and West[89, 90] which requires less than $3.62N$ messages where N is the number of processors in a complete network. We showed that the constant factor 3.62 can be lowered when N is composite. We demonstrated it by designing election algorithms when N is a multiple of 2, 3, 4, 5, or 6.

The improvement in message complexity is due to our hybrid approach which views the election as a two level process — in the first level, we divide the network into smaller-size subnetworks and solve the election problem efficiently on these subnetworks, and in the second level, we apply the algorithm due to Loui, Matsushita and West[89, 90] on the leaders from the first level.

Next, we focused our attention on a general class of topologies for message passing

architectures called Recursively Scalable Networks. We developed a distributed leader finding algorithm in a synchronous recursively scalable network. We showed that it requires less than $4.5N$ messages and the running time is at the most $2.75\sqrt{N}$ where N is the number of processors in the network.

Finally, based on passive state propagation and probe passing technique we designed a termination detection algorithm in a H-mesh. We also developed the correctness proof of the algorithm.

We suggest the following problems for further study:

a) The election algorithm for recursively scalable networks developed in chapter 3 assumes a synchronous mode of communication. It may be interesting to examine the effect of asynchronicity on the message complexity.

b) It may be useful to study whether election algorithms for square or rectangular meshes can be modified to solve the election problem in H-meshes.

c) The termination detection algorithm for H-meshes developed in chapter 4 assumes that the message transmission is instantaneous. The algorithm may be extended to the case where message transmission is not instantaneous. This may require a processor to expect an acknowledgement for each *data* message sent out by it.

d) The developement of a termination detection algorithm for a recursively scalable network making use of its communication structure may be interesting.

e) The design of election and termination detection algorithms for DeBruijn graphs may also be interesting.

Bibliography

[1] H. H. Abu-Amara, "Fault-Tolerant Distributed Algorithms for Election in Complete Networks," IEEE Trans. Comput., vol. 37, no. 4, 1988, pp. 449-453.

[2] H. H. Abu-Amara, "Fault-Tolerant Distributed Algorithms for Agreement and Election," Ph.D. Thesis UILU-ENG-88-2242, ACT-95, Univ. Illinois at Urbana-Champaign, 1988.

[3] Y. Afek and E. Gafni, "Simple and efficient distributed algorithms for election in complete networks," in Proc. 22nd Ann. Allerton Conf. on Communication, Control, and Computing, 1984, pp. 689-698.

[4] Y. Afek and E. Gafni, "Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks," in Proc. 4th ACM Symp. Principles of Distrib. Comput., 1985, pp. 186-195.

[5] Y. Afek and M. Saks, "Detecting Global Termination Conditions in the Face of Uncertainty," in Proc. 6th ACM Symp. on Principles of Distrib. Comput., 1987, pp. 109-124.

[6] P. A. Alsberg and J. D. Day, "A Principle for Sharing of Distributed Resources," in Proc. 2nd Int. Conf. Soft. Engg., 1976, pp. 627-644.

[7] K. R. Apt, "Correctness Proofs of Distributed Termination Algorithms," ACM Trans. Programming Languages and Systems, vol. 8, no. 3, 1986, pp. 388-405.

[8] K. R. Apt and N. Francez, "Modeling the Distributed Termination Convention of CSP," ACM Trans. Programming Languages and Systems, vol. 6, no. 3, 1984, pp. 370-379.

[9] K. R. Apt and J-L. Richier, "Real Time Clocks versus Virtual Clocks," in Control Flow and Data Flow: Concepts of Distributed Programming, Edited by M. Broy, Springer-Verlag 1985, pp. 475-501.

[10] R. K. Arora, S. P. Rana and M. N. Gupta, "Distributed Termination Detection Algorithm for Distributed Computations," Inform. Process. Lett., vol. 22, May 30 1986, pp. 311-314.

[11] R. K. Arora and M. N. Gupta, " More Comments on Distributed Termination Detection Algorithm for Distributed Computations," Inform. Process. Lett., vol. 29, 1988, pp. 53-55.

[12] R. K. Arora, S. P. Rana and M. N. Gupta, "Ring Based Termination Detection Algorithm for Distributed Computations," Microprocessing and Microprogramming, vol. 19, 1987, pp. 219-226.

[13] R. K. Arora and M. N. Gupta, "An Algorithm for Solving Distributed Termination Problem," Microprocessing and Microprogramming, vol. 22, 1988, pp. 263-271.

[14] R. K. Arora and M. N. Gupta, "An Generalized Framework for Derving Ring Based Termination Detection Algorithms," Journal of Microcomputer, vol. 12, 1989, pp. 147-157.

[15] H. Attiya, J. van Leeuwen, N. Santoro and S. Zaks, "Efficient Elections in Chordal Ring Networks," Algorithmica, vol. 4, 1989, pp. 437-446.

[16] R. W. Baldwin, "An Evaluation of the Recursive Machine Architecture," Ph.D. Thesis, Dept. Elec. Engg. and Comp. Sci., M.I.T., 1982.

[17] A. J. Bernstein, "Predicate Transfer and Timeout in Message Passing Systems," Inform. Process. Lett., vol. 24, 1987, pp. 43-52.

[18] P. Bernstein, V. Hadzilacos and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[19] P. Blanc, "Distributed Termination Detection when Messages Arrive out of Sequences," in Parallel Processing Edited by M. Cosnard, M. H. Barton and M. Vanneschi Elsvier Science Publishers B. V. (North-Holland), 1988, pp. 347-360.

[20] L. Bouge, "Repeated Snapshots in Distributed Systems with Synchronous Communications and their Implementation in CSP," Theoretical Computer Science, vol. 49, 1987, pp. 145-169.

[21] L. Bouge and N. Francez, "A Compositional Approach to Superimposition," in Proc. 15th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages, 1988, pp. 240-249.

[22] J. E. Burns, "A Formal Model for Message Passing Systems," Tech. Rept. No. 91, Comp. Sci., Indiana Univ., 1980.

[23] J. E. Burns, "Complexity of Communications among Asynchronous Parallel Processes," Ph.D. Diss., Georgia Inst. Tech., 1981.

[24] M. Y. Chan and F. Y. L. Chin, "Optimal Resilient Ring Election Algorithms," in Proc. 3rd Int. Workshop, Distributed Algorithms on Graphs, 1989, pp. 345-354.

[25] S. Chandrasekaran and S. Venkatesan, "A Message-Optimal Algorithm for Distributed Termination Detection," Journal of Parallel and Distrib. Comput., vol. 8, 1990, pp. 245-252.

[26] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," ACM Trans. Computer Systems, vol. 3, no. 1, 1985, pp. 63-75.

[27] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," Communications of the ACM, vol. 24, no. 4, 1981, pp. 198-206.

[28] K. M. Chandy and J. Misra, "Distributed Computation on Graphs: Shortest Path Algorithms," Communications of the ACM, vol. 25, no. 11, 1982, pp. 833-837.

[29] K. M. Chandy and J. Misra, "How processes learn," in Proc. 4th ACM Symp. Principles of Distrib. Comput., 1985, pp. 204-214.

[30] K. M. Chandy and J. Misra, "How processes learn," Distrib. Comput., vol. 1, 1986, pp. 40-52.

[31] K. M. Chandy and J. Misra, "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," ACM Trans. Programming Languages and Systems, vol. 8, no. 3, 1986, pp. 326-343.

[32] K. M. Chandy, J. Misra and L. M. Haas, " Distributed Deadlock Detection," ACM Trans. Computer Systems, vol. 1, no. 2, 1983, pp. 144-156.

[33] E. Chang and R. Roberts, "An Improved Algorithm for Decentralized Extrema-finding in Circular Configurations of Processes," Communications of the ACM, vol. 22, 1979, pp. 281-283.

[34] M. -S. Chen, K. G. Shin and D. D. Kandlur, "Addressing, Routing, and Broadcasting in Hexagonal Mesh Multiprocessors," IEEE Trans. Comput., vol. 39, no. 1, 1990, pp. 10-18.

[35] I. A. Cimet and P. R. S. Kumar, "A Resilient Distributed Protocol for Network Synchronization," in Proc. ACM SIGCOMM Symp. Commun. Arch. Protocols, 1986, pp. 358-367.

[36] S. Cohen and D. Lehman, "Dynamic Systems and their Distributed Termination," in Proc. 1st ACM Symp. Principles of Distrib. Comput., 1982, pp. 29-33.

[37] A. L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," SIGARCH Newsletter, vol. 6, no. 7, 1978, pp. 210-215.

[38] G. Della Vecchia and C. Sanges, "Recursively Scalable Networks for Message Passing Architectures," in Parallel Processing and Applications, Edited by E. Chiricozzi and A. D'Amico, Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 33-40.

[39] G. Della Vecchia and C. Sanges, "An Optimized Broadcasting Technique for WK-Recursive Topologies," in Proc. 12th IMACS World Congress on Sci. Comput., 1988, pp. 291-293.

[40] E. W. Dijkstra, "Distributed Snapshot of K. M. Chandy and L. Lamport," in Control Flow and Data Flow: Concepts of Distributed Programming, Edited by M. Broy, Springer-Verlag 1985, pp. 513-517.

[41] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations ," Inform. Process. Lett., vol. 11, no. 1, 1980, pp. 1-4.

[42] E. W. Dijkstra, W.H.J. Feijen and A. J. M. van Gasteren, "Derivation of a Termination Detection Algorithm for Distributed Computations ," Inform. Process. Lett., vol. 16, 1983, pp. 217-219.

[43] S. Dolev and A. Israli, "Uniform Dynamic Self-stabilizing Leader Election," in Proc. 5th Int. Workshop, Distributed Algorithms on Graphs, 1991, pp. 167-180.

[44] D. Dolev, M. Klawe and M. Rodeh, "An O(n logn) Unidirectional Distributed Algorithm for Extrema Finding in a Circle," Journal of Algorithm, vol. 3, 1982, pp. 245-260.

[45] J. W. Dolter, P. Ramanathan and K. G. Shin, "Performance Analysis of Virtual Cut-Through Switching in HARTS: A Hexagonal Mesh Multicomputer," IEEE Trans. Comput., vol. 40, no. 6, 1991, pp. 669-680.

[46] J. W. Dolter, P. Ramanathan and K. G. Shin, "A Microprogrammable VLSI Routing Controller for HARTS," in Proc. Int. Conf. Comp. Design: VLSI in Comput., i989, pp. 160-163.

[47] A. K. Dutta and S. Ghosh, "Deadlock Detection in Distributed Systems," in Proc. 9th IEEE Int. Phoenix Conf. on Computer and Communication, 1990, pp. 131-136.

[48] M. El-Ruby, et.al., "Leader Election in Distributed Computing Systems," in Proc. 1st Great Lakes Comp. Sci., Conf., Computing in the 90's, 1989, pp. 350-356.

[49] P. H. Enslow, "What is a Distributed Data Processing System," Computer, vol. 27, no. 1, 1978, pp. 13-21.

[50] O. Eriksen, "A Termination Detection Protocol and its Formal Verification," Journal of Parallel and Distributed Computing, vol. 5, 1988, pp. 82-91.

[51] J. A. Feldman and Nigam, "A Model and Proof Technique for Message-Based Systems," SIAM Journal of Comput., vol. 9, no. 4, 1980, pp. 768-784.

[52] N. Francez, "Distributed Termination," ACM Trans. on Programming Languages and Systems, vol. 2, no. 1, 1980, pp. 42-55.

[53] N. Francez, "On Francez's Distributed Termination," ACM Trans. Programming Languages and Systems, vol. 3, no. 1, 1981, pp. 112-113.

[54] N. Francez and M. Rodeh, "Acheiveing Distributed Termination without Freezing," IEEE Trans. Soft. Engg., vol. 8, no. 3, 1982, pp. 287-292.

[55] N. Francez, M. Rodeh and M. Sintzoff, "Distributed Termination with Interval Assertions," in Proc. Int. Colloq Formalization of Programming Concepts, Edited by J. Diaz and I. Ramos, Springer-Verlog, pp. 280-289.

[56] G. N. Frederickson and N. A. Lynch, "Electing a Leader in a Synchronous Ring," Journal of Assoc. Comput. Mach., vol. 34, 1987, pp. 98-115.

[57] G. N. Frederickson and N. Santoro, "Breaking Symmetry in Synchronous Networks," Lecture Notes in Comp. Sci., vol. 227, 1986, pp. 26-33.

[58] E. Gafni, "Improvements in the Complexity of Two Message Optimal Election Algorithms," in Proc. 4th ACM Symp. Principles of Distrib. Comput., 1985, pp. 175-185.

[59] H. Garcia-Molina, "Elections in a Distributed Computing System," IEEE Trans. Comput., vol. 31, 11982, pp. 48-59.

[60] V. M. Glushkov, et.al. "Recursive Machines," in *Information Processing* 74, Amsterdam, The Netherlands: North-Holland, 1974, pp. 65-70.

[61] O. Goldreich and L. Shrira, "Electing a Leader in a Ring with Link Failures," Acta Informatica, vol. 24, 1987, pp. 79-91.

[62] D. Gomm and R. Walter, "The Distributed Termination Problem: Formal Solution and Correctness Based on Petri Nets," in Proc. 6th Int. Meeting of Young Computer Scientists, Aspects and Prospects of Theoretical Computer Sci., 1990, pp. 159-162.

[63] R. Gusella and S. Zatti, "An Election Algorithm for a Distributed Clock Synchronization Program," in Proc. IEEE 6th Int. Conf. Distrib. Comput. Systems, 1986, pp. 364-371.

[64] S. Haldar and D. K. Subramanian, "Ring Based Termination Detection Algorithm for Distributed Computations," Inform. Process. Lett., vol. 29, 1988, pp. 149-153.

[65] C. Hazari and H. Zedan, "A Distributed Algorithm for Distributed Termination," Inform. Process. Lett., vol. 24, 1987, pp. 293-297.

[66] J-M. Helary, C. Jard, N. Plozeau and M. Raynal, "Detection of Stable Properties in Distributed Applications," in Proc. 6th ACM Symp. Principles of Distrib. Comput., 1987, pp. 125-136.

[67] J-M. Helary, A. Maddi and M. Raynal, "Controlling Information Transfers in Distributed Algorithms Application to Deadlock Detection," in Parallel Processing and Applications, Edited by E. Chiricozzi and A. D'Amico, Elsevier Science Publishers B. V. (North-Holland), 1988, pp. 85-92.

[68] J-M. Helary and N. Plozeau, "Computing Paricular Snapshots in Distributed Systems," in Proc. 9th IEEE Int. Phoenix Conf. on Computer and Communication, 1990, pp. 116-123.

[69] D. S. Hirschberg and J. B. Sinclair, "Decentralized Extrema-finding in Circular Configurations of Processors," Communications of the ACM, vol. 23, 1980, pp. 627-628.

[70] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall Int., New Jersey, 1985.

[71] S-T. Huang, "A Fully Distributed Termination Detection Scheme," Inform. Process. Lett., vol. 29, 1988, pp. 13-18.

[72] S-T. Huang, "Termination Detection by using Distributed Snapshots," Inform. Process. Lett., vol. 32, 1989, pp. 113-119.

[73] S-T. Huang, "Detecting Termination of Distributed Computations by External Agents," in Proc. IEEE 9th Int. Conf. Distrib. Comput. Systems, 1989, pp. 79-84.

[74] H. Iizuka, "Interconnection Networks with a Limited Number of Links," in Proc. Canadian Conf. Electrical and Computer Engineering, 1990, pp. 67.4.1-67.4.4.

[75] A. Itai et.al., "Optimal Distributed t-Resilient Election in Complete Networks," IEEE Trans. Soft. Engg., vol. 16, no. 4, 1990, pp. 415-420.

[76] A. D. Kandlur and K. G. Shin, "Traffic Routing for Multicomputer Networks with Virtual Cut-Through Capability," IEEE Trans. Comput., vol. 41, no. 10, 1992, pp. 1257-1270.

[77] W. Kim, "AUDITOR: A Framework for High Availability of DB/DC Systems," IBM Res. Rep. RJ3512, 1982.

[78] J. Kim and G. Belford, "A Robust, Distributed Election Protocol," in Proc. IEEE 7th Symp. Reliable Distrib. Systems., 1988, pp. 54-60.

[79] C. Kirner, "Design of a Recursively Structured Parallel Computer," in Proc. 17th Ann. ACM Comp. Sci. Conf., 1989, p. 416.

[80] E. Korach, S. Moran and S. Zaks, "Tight lower and upper bounds for some distributed algorithms for a complete network of processors," Proc. 3rd ACM Symp. on Principles of Distrib. Comput., 1984, pp. 199-207.

[81] D. Kumar, "A Class of Termination Detection Algorithms for Distributed Computations," in Conf. on Foundations of Software Technology and Theoretical Computer Science, 1985, pp. 73-100.

[82] D. Kumar, "Development of a Class of Distributed Termination Detection Algorithms," IEEE Trans. on Knowledge and Data Engineering, vol. 4, no. 2, 1992, pp. 145-155.

[83] T-H. Lai, "A Termination Detection for Static and Dynamic Distributed Systems with Asynchronous Non-first-in-first-out Communication," Lecture Notes in Computer Science, vol. 226, 1986, pp. 196-205.

[84] T-H. Lai, "Termination Detection for Dynamically Distributed Systems with Non-first-in-first-out Communication," Journal of Parallel and Distrib. Comput., vol. 3, 1986, pp. 577-599.

[85] J. van Leeuwen and R. B. Tan, "An Improved Upperbound for Distributed Election in Bidirectional Rings of Processors," Distrib. Comput., vol. 2, 1987, pp. 149-160.

[86] J. van Leeuwen, N. Santoro, J. Urrutia and S. Zaks, "Guessing Games and Distributed Computations in Synchronous Networks," in Proc. 14th Int. Conf. Coll. on Automata, Languages and Programming, 1987, pp. 347-356.

[87] G. Le Lann, "Distributed Systems – Towards a Formal Approach," in *Information Processing* 77, Edited by B. Gilchrist, Amsterdam, The Netherlands: North-Holland, 1977, pp. 155-160.

[88] H. F. Li, T. Radhakrishnan and K. Venkatesh, "Global State Detection in Non-FIFO Networks," in Proc. IEEE 7th Int. Conf. Distrib. Comput. Systems, 1987, pp. 364-370.

[89] M. C. Loui, T. A. Matsushita and D. B. West, "Election in a Complete Network with a Sense Direction," Inform. Process. Lett., vol. 22, no. 4, 1986, pp. 185-187.

[90] M.C.Loui, T.A.Matsushita and D.B.West, Corringendum - Election in a complete network with a sense of direction, Inform. Process. Lett., vol. 28, 1988, p. 327.

[91] E. L. Lozinski, "A Remark on Distributed Termination," in Proc. IEEE 5th Int. Conf. Distrib. Comput. Systems, 1985, pp. 416-419.

[92] N. Lynch, "A Hundred Impossibility Proofs for Distributed Computing," in Proc. Eighth Ann. ACM Symp. Principles of Distrib. Comput., 1989, pp. 1-27.

[93] G. A. Mago, "A Network of Microprocessors to Execute Reduction Languages – Part I and II," Int. Journal Comput. Inform. Sci., vol. 8, 1979, no. 5, pp. 349-385, no. 6, pp. 435-471.

[94] A. I. Mahdaly, H. T. Mouftah and N. N. Hanna, "Topological Properties of WK-recursive Networks," in Proc. 2nd IEEE Workshop on Future Trends of Distributed Computing Systems, 1990.

[95] A. I. Mahdaly, H. T. Mouftah and N. N. Hanna, "A Transputer-Based Emulator for Message-Passing Highly Concurrent Networks," in Proc. Canadian Conf. Electrical and Computer Engineering, 1990, pp. 31.5.1-31.5.4.

[96] K. Marzullo and L. Sabel, "Using Consistent Subcuts for Detecting Stable Properties," in Proc. 5th int. Workshop, Distributed Algorithms on Graphs, 1991, pp. 273-288.

[97] T. Masuzawa et. al., "Optimal Fault-Tolerant Distributed Algorithms for Election in Complete Networks with a Global Sense of Direction," in Proc. 3rd Int. Workshop, Distributed Algorithms on Graphs, 1991, pp. 171-182.

[98] F. Mattern, "Algorithms for Distributed Termination Detection," Distrib. Comput., vo. 2, 1987, pp. 161-175.

[99] F. Mattern, "Experience with a New Distributed Termination Detection Algorithm," Lecture Notes in Computer Science, vol. 312, 1988, pp. 127-143.

[100] F. Mattern, "An Efficient Distributed Termination Test," Inform. Process. Lett., vol. 31, 1989, pp. 203-208.

[101] F. Mattern, "Global Quiescence Detection Based on Credit Distribution and Recovery," Inform. Process. Lett., vol. 30, 1989, pp. 195-200.

[102] D. A. Menasce and R. R. Muntz, "Locking Protocol for Resource Coordination in Distributed Databases," in Proc. SIGMOD Int. Conf. Management of Data, 1978, pp. 1-14.

[103] J. Misra, "Detecting Termination of Distributed Computations using Markers," in Proc. 2nd ACM Symp. Principles of Distrib. Comput., 1983, pp. 290-294.

[104] J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes," ACM Trans. Programming Languages and Systems, vol. 4, no. 1, 1982, pp. 37-43.

[105] J. Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection," ACM Trans. Programming Languages and Systems, vol. 4, no. 4, 1982, pp. 678-686.

[106] S. Moran, M. Shalom and S. Zaks, "An 1.44...nlogn Algorithm for Distributed Leader Finding in Bidirectional Rings of Processors," IBM Research report, RC 11933, 1986.

[107] H. Muller, "High Level Petri Nets and Distributed Termination," in Concurrency Nets, Edited by K. Voss, H. J. Genrich and G. Rozenberg, Springer, 1987, pp. 349-361.

[108] N. Natarajan, "A Distributed Scheme for Detecting Communication Deadlocks," IEEE Trans. Soft. Engg., vol. 12, no. 4, 1986, pp. 531-537.

[109] M. Overmars and N. Santoro, "Time vs Bits, an Improved Algorithm for Leader Election in Synchronous Rings," in Proc. 6th Ann. Symp. Theoretical Aspects of Comp. Sci., 1989, pp. 282-293.

[110] J. Pachl, E. Korach and D. Rotem, "Lower Bounds for Distributed Maximum-finding Algorithms," Journal of the Assoc. Comput., Mach., vol. 31, 1984, pp. 905-918.

[111] G. L. Peterson, "An $O(n \log n)$ unidirectional algorithm for the circular extrema problem," ACM Trans. Programming Languages and Systems., vol. 4, 1982, pp. 758-762.

[112] G. L. Peterson, "Efficient algorithms for elections in meshes and complete networks," Tech. Rept. TR-140, Dept. of Computer Science, Univ. of Rochester, 1984.

[113] S. P. Rana, "A Distributed Solution of the Distributed Termination Problem," Inform. Process. Lett., vol. 17, 1983, pp. 43-46.

[114] J-L. Richier, "Distributed Termination in CSP: Symmetric Solutions with Minimal Storage," in Proc. 2nd Ann. Symp. on Theoretical Aspects of Computer Science (STACS 85), July 1985, pp. 267-278.

[115] K. Rokusawa, N. Ichiyoshi, Chikayama and H. Nakashima, "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems," in Proc. Int. Conf. on Parallel Processing, 1988, pp. 18-22.

[116] S. Ronn and H. Saikkonen, "Distributed Termination Detection with Counters," Inform. Process. Lett., vol. 34, 1990, pp. 223-227.

[117] G. Roucairol, "On the Construction of Distributed Programs," in Distributed Operating Systems, Theory and Practice, Edited by Y. Paker et al., Springer-Verlag, 1987, pp. 47-65.

[118] B. Rozoy, "Model and Complexity of Termination for Distributed Computations," Lecture Notes in Computer Science, vol. 233, 1986, pp. 564-572.

[119] B. Rozoy, "Solution for the Distributed Termination Problem," Lecture Notes in Computer Science, vol. 269, 1987, pp. 114-121.

[120] B. Rozoy, "Termination for Distributed Systems with Asynchronous Message Passing: Model and Cost," Computers and Artificial Intelligence, vol. 7, no. 1, 1988, pp. 1-23.

[121] M. Rudalics, "Distributed Termination Enforcement," in Proc. 4th Int. Conf. on Parallel Architectures and Languages, 1992, pp. 353-378.

[122] H. Saikkonen and S. Ronn, "Distributed Termination on a Ring," BIT, vol. 26, 1986, pp. 188-194.

[123] B. A. Sanders, "A Method for the Construction of Probe-Based Termination Detection Algorithms," in Distributed Processing, Edited by M. H. Barton, E. L. Dagless and G. L. Reijins, Elsevier Science Publishers B. V. (North-Holland), 1988, pp. 249-257.

[124] N. Santoro, "On the Message Complexity of Distributed Problems," Int. J. Comput. Inform. Sci., vol. 13, no. 3, 1984, pp. 131-147.

[125] N. Santoro, "Sense of Direction, Topological Awareness and Communication Complexity," SIGACT News, vol. 16, no. 2, 1984, pp. 52-56.

[126] R. D. Schlichting and F. B. Schneider, "Using Message Passing for Distributed Programming: Proof Rules and Disciplines," ACM Trans. Programming Languages Systems, vol. 6, no. 3, 1984, pp. 402-431.

[127] N. Shavit and N. Francez, "A New Approach to Detection of Locally Indicative Stability," Lecture Notes in Computer Science, vol. 226, 1986, pp. 344-358.

[128] S. Singh, "Expected Connectivity and Leader Election in Unreliable Networks," Inform. Process. Lett., vol. 42, 1992, pp. 282-285.

[129] S. Skyum and O. Eriksen, "Symmetric Distributed Termination," in The Book of L, Edited by G. Rozenberg and A. Salomaa, Springer-Verlag, 1986, pp. 427-430.

[130] J. Song and L. Kinney, "Distributed Termination on Mesh," in Proc. Int. Conf. on Parallel Processing, 1988, pp. 83-85.

[131] M. Spezialetti and P. Kearns, "Efficient Distributed snapshots," in Proc. IEEE 6th Int. Conf. Distributed Computing Systems, 1986, pp. 382-388.

[132] K. S. Stevens, S. V. Robinson and A. L. Davis, "The Post Office - Communication Support for Distributed Ensemble Architectures," in Proc. Six Int. Conf. Distrib. Comput. Systems, 1986, pp. 160-166.

[133] B. Szymanski, Y. Shi and N. Prywes, "Synchronized Distributed Termination," IEEE Trans. Soft. Engg., vol. 11, no. 10, 1985, pp. 1136–1140.

[134] B. Szymanski, Y. Shi and N. Prywes, "Terminating Iterative Solution of simultaneous Equations in Distributed Message Passing Systems," in Proc. 4th ACM Symp. Principles of Distrib. Comput., 1985, pp. 287-292.

[135] D. Talia, "Notes on Termination of Occam Processes," SIGPLAN Notices, vol. 25, no. 9, 1990, pp. 17-24.

[136] R. B. Tan and J. van Leeuwen, "General Symmetric Distributed Termination Detection," Tech. Rept. RUU-CS-86-2, Univ. of Utrecht, Utrecht, 1986.

[137] R. B. Tan, G. Tel and J. van Leeuwen, "Comments on Distributed Termination Detection Algorithm for Distributed Computations," Inform. Process. Lett., vol. 23, 1986, p. 163.

[138] G. Tel, "The Structure of Distributed Algorithms," Ph.D. Thesis, University of Utrecht, Utrecht, 1988.

[139] G. Tel and J. van Leeuwen, "Comments on a Distributed Algorithm for Distributed Termination," Inform. Process. Lett., vol. 25, 1987, p. 349.

[140] G. Tel, R. B. Tan and J. van Leeuwen, "The Derivation of Graph Marking Algorithms from Distributed Termination Detection Protocols," Science of Computer Programming, vol. 10, 1988, pp. 107-137.

[141] G. Tel and Mattern, "Comments on Ring Based Termination Detection Algorithm for Distributed Computations," Inform. Process. Lett., vol. 31, 1989, pp. 127-128.

[142] R. W. Topor, "Termination Detection for Distributed Computations," Inform. Process. Lett., vol. 18, 1984, pp. 33-36.

[143] P. C. Treleaven and R. P. Hopkins, "A Recursive Computer Architecture for VLSI," SIGARCH Newsletter, vol. 13, no. 3, 1982, pp. 229-238.

[144] S. Venkatesan, "Reliable Protocols for Distributed Termination Detection," IEEE Trans. on Reliability, vol. 38, no. 1, 1989, pp. 103-110.

[145] J. P. Verjus, "On the Proof of a Distributed Algorithm," Inform. Process. Lett., vol. 25, May 29 1987, pp. 145-147.

[146] W. T. Wilner, "Recursive Machines," in Algorithmically Specialized Parallel Computers, Edited by L. Snyder, et.al., Academic Press, Inc., 1985, pp. 95-104.