INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

HIGH PERFORMANCE TERMINATION DETECTION TECHNIQUES SUPPORTING MULTITHREADED EXECUTION

by

YILI TSENG
B.S.M.E. National Taiwan University, Republic of China, 1985
M.S. University of Florida, 1990
M.S. University of Central Florida, 1995

A dissertation submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2000

Major Professor: Ronald F. DeMara

ABSTRACT

Efficient detection of execution termination is essential for optimizing throughput of multithreaded parallel computer architectures. In particular, an ensemble of processing elements (PEs) is said to have reached termination of processing upon completion of each interval of concurrent activity. Points at which synchronization occur are referred to as *synchronization barriers*. The design objective is to minimize the amount of overhead required to enforce completion of each barrier prior to the resumption of subsequent processing.

This dissertation begins by developing a novel taxonomy for termination detection techniques based on thread allocation strategy and degree of processor reactivation support. A capability class hierarchy ranging from *Static-Binding Idle-Tasking* to *Dynamic-Binding Any-Tasking* is derived as a result of the taxonomy. Together they assist significantly in identification of properties which facilitate algorithm assessment and refinement. A message, bit, time, and space optimality analysis indicates that as few as *(E-N)* additional messages can be utilized to realize dynamic binding rather than static

binding of threads on N PEs with E reporting events. These results are assessed against those for the CV, LTD, Credit, and Tiered algorithms to demonstrate the corresponding time and space performance which are achievable in practice.

The *Tiered Detection Algorithm* is shown to approach practical efficiency limits and is further refined in terms of its global invariant across non-serializable message communication channels. By attaching the level of thread nesting to thread consumption and production counts, it prevents false termination hazards. Its advantage in detection delay is revealed in average and worst cases over CV and LTD algorithms concerning the traversal of processor hierarchy and implementation performance of the Credit Algorithm.

The Tiered algorithm is then extended to a hardware-based approach, which is shown to be time and wire-efficient. The *Distributed-Sum Bit-Comparison (DSBC)* logic developed is capable of supporting dynamic allocation of tasks for multithreaded execution on shared-memory, message-passing, and/or single-chip multiprocessors. For a system of N PEs, a single instance of global logic and N instances of local logic interconnected by $3N$ wires are shown to provide direct support to the compiler and programmer for any arbitrary number of barriers. DSBC detection time upon completion of the last task is shown to scale linearly in terms of the number of active barriers in the

system. Comparison to Wired-NOR hardware and shared-lock software approaches demonstrate reduced barrier detection time, decreased inter-PE wiring requirements, and increased functionality. Finally, a version is designed using *Null Convention Logic* to provide a delay-insensitive alternative implementation that eliminates race conditions and timing considerations in distributed environments.

Dedicated to my father and mother,

Mr. Pao-Tung Tseng and Mrs. Hsueh-Ju Liu

高效能支援多軌執行之偵測程式終結技術

謹以此博士論文獻給我的雙親 — 曾保堂先生與劉雪如女士

曾屹立

ACKNOWLEDGMENTS

First, I would like to appreciate my parents, Mr. Pao-Tung Tseng and Mrs. Hsueh-Ju Liu. Without their love and financial support during the course of my pursuit of the Ph.D. degree, there will be no existence of this dissertation. I will make them proud of this son by my future contribution to this universe.

Next, I want to acknowledge my advisor, Dr. Ronald. F. DeMara. His considerateness and supportive personality gave me a precious opportunity to tackle my health problem before working on the academic research. Not every advisor owns this noble character and that proves to be more valuable than the academic ability is. His master style instruction gave me the greatest freedom to challenge myself and explore the unknown space of knowledge while his insight provided a clear direction. It is not only beneficial but also pleasant to practice researching under his instruction. I will reward him by my future achievements.

I also would like to express my gratitude to an unsung heroine, my wife, Chyong-Ru. I appreciate her support and companion through the ordeal of my work toward the Ph.D. degree. My unusual experience is not worth mentioning to others, however it provides useful practice to both of us, which will benefit the rest of our life. The adversity and adversary in the past turned out to be stepping stones rather than obstacles in the road.

The success of my elder brother and sister-in-law, Drs. Yi-Ping Tseng and Huei-Chu

Liao inspired my desire to explore more knowledge. Their encouragements keep me motivated in difficult times. They will always be my role models.

As for the roles of my sons, David and Daniel, the usual scenario in a dissertation's acknowledgments is " I appreciate my sons' cooperation by going to bed at 9 P.M. everyday so that I have time to finish my research. ". Since that never applies to them, I have made the following decision. If they want their names to be inscribed in any dissertation, they have to write their own!

銘謝

首先我要感謝我的雙親，曾保堂先生與劉雪如女士。在攻讀博士學位過程中，如無他們精神上與經濟上的支持，今日即無此博士論文之存在，我將以未來對宇宙之貢獻榮耀他們。

其次我要感謝我的指導教授，隆納德 •狄邁拉博士，其熱心助人及體諒他人之個性，使我有寶貴之機會調養好身體健康再進行學術研究。並非每位指導教授都有此高貴情操，且此情操證明比學術能力更有價值。他的洞察力及眼光指點了明確的研究方向，但其大師式的教導讓我有極大自由挑戰自我及探索未知之知識空間。在其指導下作科學研究，非但獲益良多且是令人愉快之人生經驗。我將以未來成就回報。

我也要對一位幕後英雄表達謝意，即內子莊瓊如。感謝她的支持與伴隨我走過邁向博士之路之嚴苛考驗。我比他人艱辛的境遇不足為外人道，但卻提供了彌珍足貴之人生歷練，將使我們夫婦終生受益。過去的逆境與阻撓者證明不成為路上之障礙物，反是助我成功之踏腳石！

我的大哥與大嫂，曾一平博士與廖惠珠博士，當年他們的成就激發了我追求更多知識之欲望，他們的鼓勵與支持使我在低潮時候依然鬥志昂揚，他們永遠是我的楷模。

至於我兒，曾翔緯與曾勛熠，尋常博士論文銘謝頁都寫道：『感謝我子女之合作，每晚九時準時就寢，使我有餘暇完成博士學位。』然因此情從未在我二子身上發生，故我決定，如果他們想要名字被寫入任何博士論文，他們必須撰寫各自之博士論文！

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Efficient barrier synchronization and termination detection techniques are essential for optimizing throughput in multiple processor architectures. An ensemble of processing elements (PEs) is said to be synchronized, or to have reached a quiescent state [9], upon completion of each interval of concurrent activity. Points at which synchronization occur are referred to as a barriers [20][21][18]. The design objective is to minimize the overhead required to enforce completion of each barrier prior to the resumption of subsequent processing.

1.1 The Barrier Synchronization Problem

Figure1.1 shows a code fragment containing three statements, labeled S1, S2, and S3, which invokes three distinct processes labeled P1, P2, and P3. Let I(S) and O(S) denote the set of input and output variables, respectively, of statement S. Statements

```
      cobegin;
S1:         x:=P1(a);
S2:         y:=P2(b);
      coend; ⇐ "Barrier" (point at which interprocess synchronization must occur)
S3:   P3(x,y);
```

Figure 1.1: Parallelizable Code Fragment Requiring Synchronization

S1 and S2 have no input, output, nor control dependencies, and hence by Bernstein's conditions [19]:

$$I(S1) \cap O(S2) = I(S2) \cap O(S1) = O(S1) \cap O(S2) = \emptyset$$

An empty intersection implies that S1 and S2 can be executed simultaneously on separate processors. On the other hand, statement S3 can only be executed correctly after both S1 and S2 have terminated since:

$$I(S3) \cap O(S1) = \{x\} \quad and \quad I(S3) \cap O(S2) = \{y\}$$

The barrier which corresponds to the completion of the concurrent processing, which must occur before $S3$ is initiated, is indicated by the **coend** statement shown in Figure 1. The processing tasks between barriers are executed by multiple Processing Elements (PE's) within the machine. PE's may execute these tasks simultaneously without impacting correctness, but only if the barriers are properly enforced. Since some barriers may only involve a subset of the processes or resources in the system, those which actually take part in a specific barrier are delineated as *participating tasks* or *participating PEs*, accordingly.

1.2 Significance of Synchronization and Quiescence Detection

Parallel and distributed processing techniques frequently offer cost-effective ways to boost throughput.[8][52][47] As networking technology matures and environments such as the Internet rapidly expand, distributed computing is an effective method to fully utilize the available resources to increase throughput [3][1][2]. Synchronization is a fundamental issue to both parallel and distributed computation. Its performance effects the overall performance the parallel and distributed multiprocessor systems profoundly since any idle processor in the system cannot proceed to execute the next

procedure before the synchronization has been completed. Even if the processors can be reactivated to process tasks of another application to utilize the processing cycles, overheads are incurred. An ineffective termination detection scheme will exchange more messages which congest the communication channels and effect the transmission of messages required by the underlying computation. Therefore the quiescence detection process plays an important role in parallel and distributed computing.

1.3 Application-Driven Synchronization Requirements

Characteristics which influence selection of a barrier mechanism include the application's task granularity between barriers, number of simultaneous barriers, and task creation/allocation strategy.

1.3.1 Granularity of the Application Tasks

Task granularity refers to the number and relative complexity of the operations within each concurrent process. The coarseness or fineness of granularity determines the interval of productive execution between barriers. As the tasks requiring synchronization become increasingly fine-grained, the relative impact of synchronization overhead on processing throughput becomes magnified. Thus, frequently synchronized applications are less able to tolerate the latency at which barriers are detected and may require hardware solutions to the synchronization problem.

1.3.2 Degree of Thread Concurrency

Singly-threaded applications require at most one barrier at any instant while *multi-threaded* applications may take advantage of concurrent barriers which are active simultaneously. For example, in a multi-user environment, each user's job involves

tasks contributing towards distinct barriers. Since there are no data dependencies between tasks from different users, these tasks could be executed simultaneously if the synchronization mechanism could distinguish between barrier signals. Likewise, single-user applications may also contain multiple sets of tasks contributing to a different active barrier for each of its threads.

1.3.3 Apriori Knowledge of PE Participation

Knowledge of whether a PE will participate in a barrier may not be readily available at compile-time. Applications in which the number of participating tasks and/or their processor binding can be determined prior to execution are said to exhibit *procedural task creation*. On the other hand, applications which dynamically select the PE's which will participate in the barrier and/or generate new processes based on run-time conditions are capable of *adaptive task creation*. Applications requiring synchronization support for adaptive process creation include Remote Procedure Calls, recursive algorithms, and dynamic search strategies.

Additionally, adaptive process creation may create a *launch-in-transit hazard*. This refers to the situation when all processors are idle, yet a message is in transit from one PE to another that will launch a new task or subprocess upon arrival at its destination. While all processors appear to be idle, the barrier is not actually reached. Launch-in-transit messages can be difficult to track, yet their proper accounting is vital for enforcing the barrier and ensuring correctness of program execution. Launch-in-transit hazards can arise on distributed-memory architectures such the iPSC hypercube, nCUBE, and others where the synchronization technique lacks a global snapshot of processor activity.

1.4 Architecture-Driven Synchronization Requirements

Each phase of the synchronization algorithm must accommodate the primary architectural features of the target machine such as its communication mechanism, level of hardware support for synchronization, and various machine-specific parameters.

1.4.1 Interprocessor Communication Strategy

Since shared-memory architectures provide a common region of the address space which can be accessed by multiple PE's, applicable barrier techniques involve the use of *global synchronization variables.* The design objectives involve minimizing contention for access to these variables. On the other hand, distributed memory machines must exchange *synchronization messages* through the machine's interconnection network. Thus, design objectives for distributed-memory synchronization schemes involve minimizing message traffic, transit times, and computational overhead required to process these messages.

1.4.2 Machine-Specific Configuration Parameters

Irregardless of whether a shared or distributed memory model is used, quantities such as the ratio of computation-to-communication speed can be determining factors in the applicability of a barrier technique. For instance, a synchronization algorithm may be applicable to a distributed- memory architecture, but the relative cost of communication on a particular machine may make certain approaches intractable. Similarly, the number of PE's in the machine, PE interrupt support, and spinlock availability will influence which barrier approaches are appropriate.

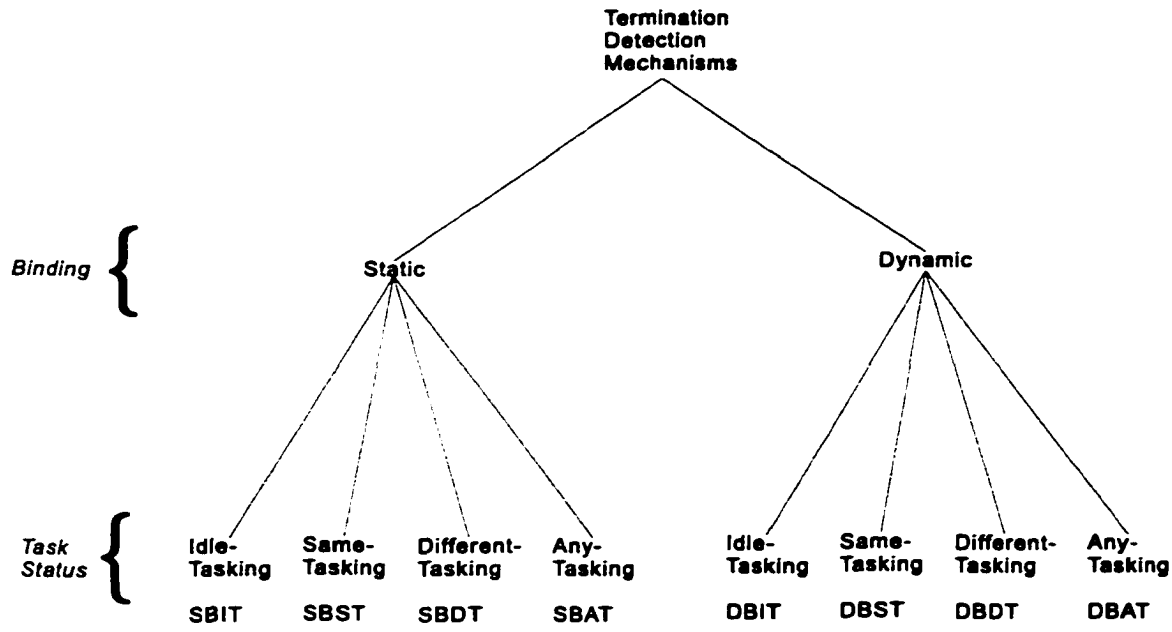


Figure 1.2: Classification Scheme based on Functionality of Barrier

1.4.3 Availability of Barrier Hardware

The use of dedicated hardware to enforce barriers can significantly reduce synchronization latencies. The hardware design issues involve minimizing the logic requirements per PE and reducing interprocessor wiring complexity, while optimizing flexibility. Use of dedicated barrier hardware can be prohibitive except in new machine designs since many commercial systems offer little hardware support and retrofitting may sacrifice the application's portability.

1.5 Taxonomy of Termination Detection Techniques

1.5.1 Capability Categories

After extensive studying of various barrier synchronization mechanisms, we propose the novel categorization of them as in Figure 1.2 based on their features. First they are grossly classified as *static-binding* and *dynamic-binding* according to the way

in which they allocate PEs and schedule processes. Only mechanisms which can both allocate PEs and schedule processes dynamically are categorized as dynamic-binding. Although some approaches schedule processes dynamically, they fulfill the tasks on fixed tree of PEs; hence they are still classified as static-binding. The barrier synchronization mechanisms are further classified as *idle-tasking*, *same-tasking*, *different-tasking*, and *any-tasking* under each binding scheme based on how they behave after they enter the barrier. If all joining PEs cannot be reactivated for other tasks after they enter the barrier, the mechanism is classified as idle-tasking capable. If joining PEs can be reactivated for other tasks in the same barrier after they enter the barrier, the mechanism is classified as same-tasking capable. If joining PEs can only be reactivated for other tasks in other barriers after they enter the barrier, the mechanism is classified as different-tasking capable. If joining PEs can be reactivated for other tasks in either the same barrier or other barrier after they enter the barrier, the mechanism is classified as any-tasking capable. Combined with binding calassification, there are eight categories for barrier synchronization mechanisms.

1.5.2 Class Hierarchy

Figure 1.3 shows a hierarchy of capabilities for the barrier classes defined in the previous section. In particular, class A is said to *subsume* class B if the mechanisms in class A can perform the operations of those in class B. For example, a technique in the DBAT class can correctly execute any synchronization operation supported by any other class. Therefore, the DBAT class subsumes all other classes. If one can realize DBAT class capability with a cost and efficiency comparable to any other class, then he will have provided a general technique for the barrier synchronization problem.

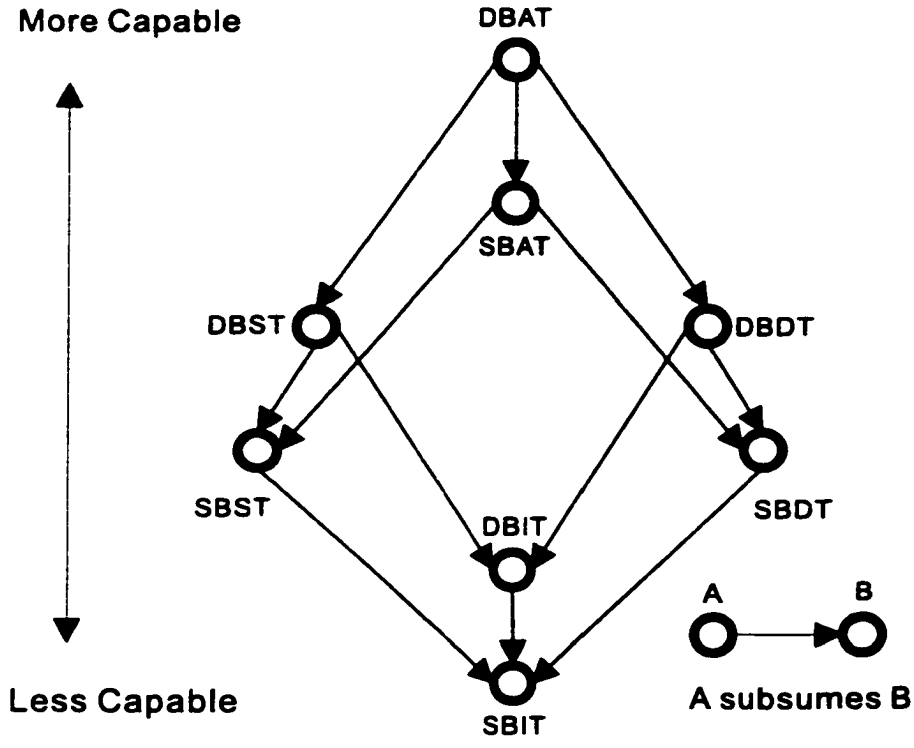


Figure 1.3: Hierarchy of Barrier Classes

1.6 Organization of the Dissertation

A novel capability taxonomy of termination detection algorithms along with a capability class hierarchy resulting from it are proposed in Chapter 1 to facilitate the analysis of thirteen popular termination detection algorithms which are introduced and investigated in Chapter 2. The classification assists in identifying the capability of existing termination detection techniques and helps to shape the requirements for an optimal algorithm. Next, the optimality of termination detection algorithms is studied and derived in Chapter 3 by refining existing lower bound of message complexity in termination detection algorithms. A new lower bound of termination detection algorithms is proposed as a result of the optimality analysis, which also serves as the ultimate goal of our approaches for new termination detection algorithms. A software approach referred to as the *Tiered Algorithm* is designed and presented in Chapter 4 following integration of our refinements along with the advantageous features extracted from the examination of other efficient termination detection algorithms. Its performance is shown to approach practical efficiency through comparison with three major algorithms which are intended to be optimal in terms of message complexity. An extension of the same fundamental concept is realized by hardware approach in Chapter 5. The advantages of the *Distributed-Sum Bit-Comparison* Logic are revealed by contrasting its performance with those of a software scheme and another major hardware design. A delay-insensitive version of the DSBC Logic, which eliminates timing concerns, is also developed with *NULL Convention Logic* technique. In Chapter 6, conclusion is summarized and a direction for future work is outlined.

CHAPTER 2

PREVIOUS WORK

2.1 Overview

The termination detection issue in parallel and distributed computations has been extensively researched in the past and many termination detection algorithms have been proposed, both in software and hardware. Major designs in the literature are introduced and classified according to the capability category proposed in previous Chapter in the following sections. The evolution of termination detection techniques is implicitly covered and sheds a light to the requirements of an optimal termination detection algorithm.

2.2 Static-Binding Idle-Tasking Capable Techniques

2.2.1 Butterfly Barrier

The Butterfly Barrier [20] [21] [18] [6] is an approach to barrier synchronization which is free of hot spots and incurs a delay which grows logarithmically with the number of processors. This technique builds upon a two-processor synchronization kernel which is illustrated in Figure 2.1. Statement S1 guarantee that each processor will not continue to S2 until the other processor has completed S4 from the previous

P1	P2
S1: while($f0 \neq 0$);	S1: while($f1 \neq 0$);
S2: $f0 = 1$;	S2: $f1 = 1$;
S3: while($f1 \neq 1$);	S3: while($f0 \neq 1$);
S4: $f1 = 0$;	S4: $f0 = 0$;

Figure 2.1: 2-Process Butterfly Barrier

barrier. This prevents a race condition which can occur in the presence of very short code segments or with processors which are subject to program interruption. S2 signals entry of the barrier code to the other processor. In S3, the processor waits until S2 has been executed by the other processor. Finally, S4 is used re-initialize the flags $f0$ and $f1$ for the next barrier.

The author proposed using this two-processor Butterfly lock to synchronize three or more processors using the structure shown in Figure 2.2. Multiple instances of the two processor lock are employed to prevent any processor from proceeding beyond the barrier until all processors have reached the barrier. This structure can be readily expanded to synchronize 2^i or more processors where $i > 1$. If the number of processors is not a power of 2, then it is possible to circumvent this restriction by having processors in the network stand-in for missing processors.

Note that this barrier synchronization technique does not rely on accessing a shared variable common to all processors. Each flag modified by a single process is polled by only one other process. However, it is important to consider the location of the set of synchronization flags used. If all of the flags are stored in a region of memory which requires shared hardware for access (i.e. buffers, busses, memory devices, etc.) then a contention problem may still occur. This barrier detection method is classified as SBIT. The technique is static-binding because it must embed the barrier synchronization codes in each process; that means each process has to be

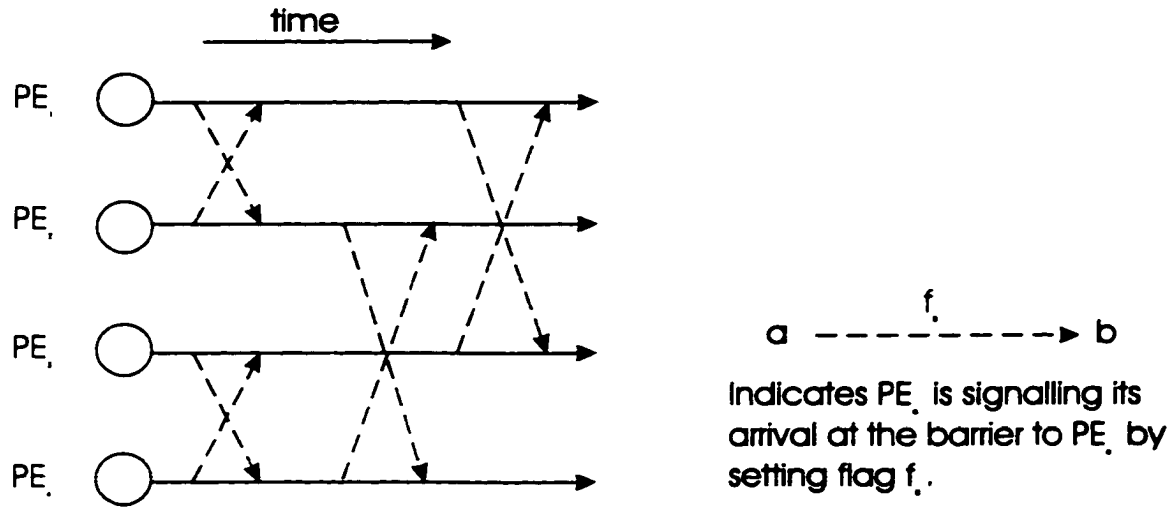


Figure 2.2: Butterfly Barrier Expanded to Support Multiple Processors

known in advance. It is classified as idle-tasking since all PEs reaching the barrier code shown in Figure 2.1 must wait at S3 for the paired PE to execute S2.

2.2.2 U-cube Tree Algorithm

The U-cube Tree Algorithm [23] is designed for the wormhole-routed [39] hypercube multicomputers by taking advantage of the feature that message latency is almost insensitive to the distance between the source and destination nodes in wormhole routing. Therefore it may not be efficient if implemented on other interconnection network. The algorithm uses a barrier processor to do termination detection, which can be a joining PE or a dedicated processor. The algorithm takes part in both the distribution phase, in which the barrier processor either broadcast or multicast the message to all the joining PEs, and the reduction phase, in which the joining PEs report to the barrier processor for termination detection. The algorithm first organizes the joining PEs as a dimension-ordered chain which will be explained shortly. In the distribution phase, the barrier processor unicasts to one of the joining PE first, then

every PE which has received a message unicasts the message to one of the PEs which have not received messages yet in the following steps until all joining PEs receive a message. It requires exactly $k = \lceil \lg(m+1) \rceil$ steps. The reason for using dimension-ordered chain is to guarantee that the paths followed by concurrent messages in the U-cube tree do not go through any common channel. The dimension-ordered chain is formed by the three following definitions [23]. Let $\sigma_{n-1}(x)\sigma_{n-2}(x)\ldots\sigma_0(x)$ represent the binary address of a node.

Definition 1 *The binary relation "dimension order," denoted $<_d$, is defined between two nodes x and y as follows: $x <_d y$ if and only if either $x = y$ or there exists a j such that $\sigma_j(x) < \sigma_j(y)$ and $\sigma_i(x) = \sigma_i(y)$ for all $i, 0 \leq i \leq 1$.*

Definition 2 *A sequence $\{d_1, d_2, d_3, \ldots, d_m)$ is a dimension-ordered chain if and only if all the elements are distinct and the sequence is dimension-ordered, that is, if $d_i <_d d_j$ for all i, j, such that $1 \leq i \leq j \leq m$.*

Definition 3 *A sequence $\{d_1, d_2, d_3, \ldots, d_m)$ is called a d_0-relative dimension-ordered chain if and only if $\{d_0 \oplus d_1, d_0 \oplus d_2, \ldots, d_0 \oplus d_m\}$, is a dimension-ordered chain.*

The U-cube Tree Algorithm is shown in Figure 2.3. An example based on (11010)-relative dimension-ordered sequence {01110, 01000, 11100, 11011, 00001, 01101} is given in Figure 2.4. In the reduction phase, the algorithm just use a reverse U-cube Tree. This algorithm is classified as SBIT. It is static-binding because it needs to know the joining PEs to arrange the dimension-ordered sequence before it starts. It is idle-tasking since all PEs have to wait for the barrier processor for new messages.

Algorithm: The U-cube Tree Algorithm

Input: d_0-relative cube ordered address $\{d_{left}, d_{left+1}, \ldots, d_{right}\}$,
where d_{left} is the local address.
Output: Send $\lceil \lg(right - left + 1) \rceil$ messages
Procedure:
begin
$p = \lceil \lg(right - left + 1) \rceil$ messages
while$\{p > 0\}$ do
$center = left + \lceil \frac{right - left + 1}{2} \rceil$;
$D = \{d_{center}, d_{center+1}, \ldots, d_{right}\}$;
Send a message to node d_{center} with the address field D;
$right = center - 1$;
$p = p - 1$;
endwhile
end;

Figure 2.3: U-cube Tree Algorithm [23]

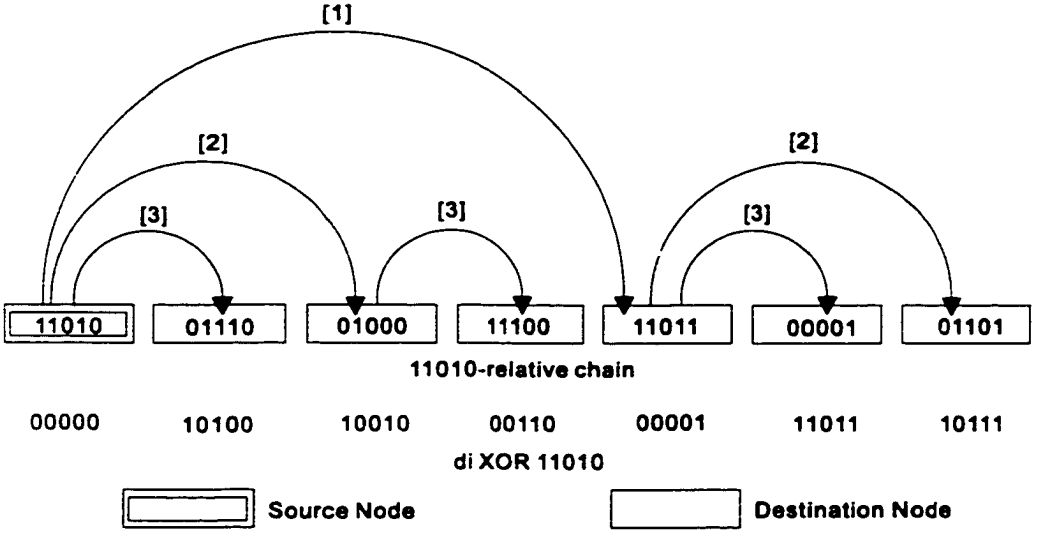


Figure 2.4: Example for U-cube Tree Algorithm

```
Procedure start;
(* performed by the root node when it decides to
detect termination of the underlying computation *)

begin
        mystate ← DT;

        for each outgoing network link ln do
            begin
                color ln;
                Send a warning message on ln
            end
end;

Procedure receive_warning;
(* performed when a node p, receives a warning message from its neighbor q *)

begin
       color the incoming link (q, p);
       if (mystate <> DT) then
            begin
                mystate ← DT;
                for each outgoing link ln do
                    begin
                        color ln;
                        Send a warning message on ln
                    end
            end
end;
```

Figure 2.5: Procedures used in CV Algorithm [10]

Procedure send_message(q : neighbor);
(* performed when a node p wants to send a message to its neighbor q *)

begin
Push $TO(q)$ on the stack;
send the actual message to node q
end;

Procedure receive_message(y : neighbor);
(* performed when a node x receives a message
from its neighbor neighbor y on the link (y, x) that was colored by x*)

begin
receive message from y on the link (y, x)
if (link (y, x) has been colored by x) **then**
push $FROM(y)$ on the stack
end;

Figure 2.6: Procedures used in CV Algorithm [10]

2.3 Static-Binding Same-Tasking Capable Techniques

2.3.1 CV Algorithm

The procedures used in the CV algorithm [10] are given in Figures 2.5 to 2.7. The CV algorithm first organizes all the participating processors as a logical spanning tree of PEs. It can start after the underlying computation starts. When the CV algorithm starts, it first flushes every links in the spanning tree to take care of messages sent before the algorithm starts; then the root node changes its state to DT (detecting termination) and sends a *warning* message to each of its children and color the link at the same time. In turn, the warning messages are passed through links connected to all its child nodes until all the participating nodes are notified of the detecting termination decision. The CV algorithm maintains a stack in each PE to keep track of sending and received activities on each PE. When a node becomes idle, it examines

```
Procedure stack_cleanup;

begin
        while (top entry on stack is not of the form "TO()") do
                begin
                        pop the entry on the top of stack;
                        let the entry be FROM(q);
                        send a remove_entry message to q;
                end
end;

Procedure idle;
(* performed as soon as the node becomes idle *)

begin
        stack_cleanup
end;

Procedure receive_remove_entry(y : neighbor);
(* performed when a node x receives a remove_entry message from its neighbor y *)

begin
        scan the stack and delete the first entry of the form TO(y);
        if idle then
          stack_cleanup
end;
```

Figure 2.7: Procedures used in CV Algorithm [10]

its stack from the top. For every received entry, it sends the *remove_entry* message to the sender and erase the entry from its stack. It repeats this until it encounters a sending entry. This procedure is defined as *stack_cleanup*. When a node receives a remove_entry message, it scans its stack and deletes the sending entry related to this message and repeats the stack_cleanup procedure as previously described if it is in idle status. A node sends a *terminate* message to its parent when it is idle, its stack is empty, each of its incoming links is colored, and it has received the terminate message from each of its children. When the root node meets the requirements to report terminate message, it declares the termination.

The CV algorithm is classified as SBST. It is treated as static-binding because it is performed on a fixed spanning tree of PEs formed before its execution. The CV algorithm, as originally defined, supports only processor reactivation for the same barrier; hence it is classified as same-tasking.

2.3.2 LTD Algorithm

The LTD algorithm [12] [13] is an improvement over the CV algorithm. Its algorithm for all PEs is given in Figure 2.8. The two procedures used in the algorithm are given in Figure 2.9. Like the CV algorithm, it organizes participating processors as a spanning tree of PEs first and it can start after the underlying computation starts. When the root decides to start the algorithm, it changes to DT (detecting termination) status and sends a *start* message to each of its children. In turn, its children send start messages to their own children until all participating processors are notified of the root's decision. Each PE maintains four variables, namely $in_i, out_i, mode_i$,and $parent_i$, to apply *message counting* to decide whether all the messages sent by it have been finished. The integer array, $in_i[1..n]$, is used to keep track of the messages which are received from PE 1 to n and have not been finished. The number of messages

A1:(Upon sending a basic message to p_j)
$out_i := out_i + 1$;

A2:(Upon receiving a basic message from p_j)
$in_i[j] := in_i[j] + 1$;
if ($parent_i$ = NULL) $\wedge$ ($i \neq 1$) **then** $parent_i := j$;

A3:(Upon deciding to switch to DT mode) /* for p_1 */
or (Upon receiving a **START** message) /* for $p_i, 2 \leq i \leq n$ */
$mode_i$:= DT;
for each child p_j of p_i **do**
send a **START** message to p_j;
end for
if (p_i is idle) **then**
call *respond_minor*(i);
call *respond_major*(i);
end for

A4:(Upon receiving a **FINISHED(k)** from p_j
$out_i := out_i - k$;
if ($mode_i$ = DT) $\wedge$ (p_i is idle) **then call** *respond_major*(i);

A5:(Upon turning idle)
if ($mode_i$ = DT) **then**
call *respond_minor*(i);
call *respond_major*(i);
end if

Figure 2.8: Algorithm for $p_i, 1 \leq i \leq n$, in LTD Algorithm [12]

```
Procedure respond_minor(i : integer)
begin
      for each j ≠ parent_i with in_i[j] ≠ 0 do
          send a FINISHED(in_i[j]) to p_j;
          in_i[j] := 0;
      end for;
end;

Procedure respond_major(i : integer)
begin
      if (out_i = 0) then
          if (i = 1) then report termination
          else
              send a FINISHED(in_i[parent_i]) to parent_i
              in_i[parent_i] := 0;
              parent_i := NULL;
          end if;
      end if;
end;
```

Figure 2.9: Procedures used in LTD Algorithm [12]

sent by j to i is stored in $in_i[j]$. The integer, out_i, records the number of unfinished messages sent by PE i itself. The Boolean variable, $mode_i$, shows the status of the processor (DT or NDT). The pointer which indicates where the most recent major message came from is stored in $parent_i$. A major message is the message which is received when the processor is idle and has finished all the messages which it sent to other processors, otherwise the received message is defined as a minor message. Whenever a node turns idle, it calls procedures *respond_minor* and *respond_major* to detect termination. What procedure *respond_minor* does is to send one $FINISH(k)$ message to each non-parent node which has sent it messages to inform them of the number of messages which it has finished for them, where k means the total number of messages which it has finished for a specific node before turning idle. This is the largest improvement over the CV algorithm. It uses one FINISH(k) message instead

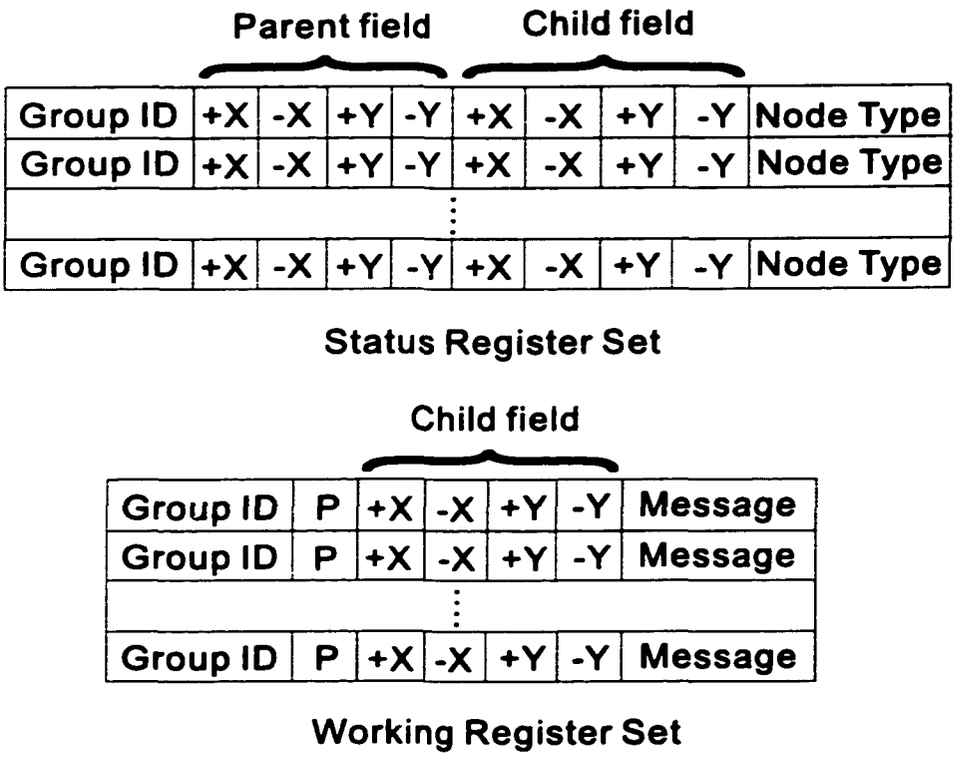


Figure 2.10: Status and Working Registers

of k remove_entry messages as in the CV algorithm to save $(k-1)$ messages. As for the procedure *respond_major* . it checks if all the sent messages have been finished and sends one $FINISH(k)$ message to parent node if all the messages sent by it are finished. After the root turns idle and finds out that all the messages it sent out are finished, it concludes the termination. It is classified as SBST with the same rationale for the CV algorithm.

2.4 Static-Binding Different-Tasking Capable Techniques

2.4.1 Collective Synchronization Tree

The Collective Synchronization (CS) Tree algorithm [29] implements on 2D mesh networks. A CS tree is built on joining PEs before the algorithm begins. The CS

tree is a logic tree which is rooted at the central node of the joining PEs and links all member nodes together. In short, a CS tree is built by dividing the joining PEs into four quadrants according to their positions in the 2D mesh network and finding the central node as the root. Then it initialize the routers to set up the CS tree in hardware. To record the parent-child relationship in the CS tree, the routers use two sets of centralized registers, namely *status* and em working registers, which are shown in Figure 2.10 [29]. each status register contains two fields, *parent* and *child*, and each has four bits (+X, -X, +Y, -Y). A "1" in any bit in a field indicates that the parent or child node can be reached through the corresponding port. The *node type* indicates the role of the node in the CS tree, which can be the central node, a leaf, an internal node, or an intermediate node. The working status is used to record whether the message from the local processor (P field) or child nodes (child field) has arrived. Both registers for the same barrier are identified by the *Group ID*, hence the CS Tree algorithm can implement on different barriers simultaneously by applying different group ID. The operations of all nodes in the algorithm are summarized in Table 2.1. In general, the leaf nodes reports to their parents after they finish their tasks; the internal and intermediate nodes wait for messages from all their children and the local processor before they report to their parents; the central node declares the completion of the barrier after it receives messages from all its children. The CS Tree algorithm is classified as SBDT. It is static-binding because it must know the joining PEs a-priori to build the CS Tree. It employs different register sets for different barriers hence it has different-tasking ability.

2.4.2 Fetch-and-Add

The *Fetch-and-Add* (F & A) primitive [19] [45] is a hardware feature which allows a PE to indivisibly read and increment a counting variable stored in shared memory.

Node Type	Operations
Leaf	**RA1:**Receive a Rmsg from local processor and forward it through the port specified in SReg[PF].
Internal	**RB1:**Receive a Rmsg (perhaps from local processor) and set the bit corresponding to its input port in WReg. If SReg[CF]≠ WReg[CF] or WReg[P] is not set. then discard the message and go to RB1. **RB2:**Forward the message through the port specified in SReg[PF]. **RB3:**Reset WReg.
Central	**RC1:**Same as RB1. **RC2:**Reset WReg and notify local processor.
Intermediate	**RD1:**Receive a Rmsg and set the bit corresponding to its input port in WReg[CF]. If SReg[CF]≠ WReg[CF], then discard the message and go to RD1. **RD2:**Same as RB2. **RD3:**Reset WReg.

Table 2.1: Operations in the Router for CS Tree [29]

$num\_at\_barrier = F\&A(counter, 1)$;
if ($num\_at\_barrier < num\_expected$)
while($exit\_flag == 0$);
else
{sequential code};
$exit\_flag = 1$;
endif;

Figure 2.11: Fetch and Add Barrier Code

If the number of processes converging on a barrier is known a-priori, a counting variable can be used to detect the barrier . An example of the code executing on the converging processes is shown in Figure 2.11. As each processor reaches the point in its operation where synchronization is required, it increments and tests the counting variable using an F & A instruction. When a process detects that the counting variable has reached the expected final value, the barrier has been reached. Both *counter* and *exit_flag* are initialized to zero prior to executing the barrier code. Note that the process which detects the barrier (the last converging process to reach the barrier) can execute sequential code, since the *else* portion of the *if* statement can be executed by a process only if all other converging processes are in a busy-wait condition testing the exit_flag. The use of this primitive for barrier synchronization can result in significant hot spots due to contention for both the counting variable and the exit_flag. The detection latency encountered when using the F & A primitive is determined by the access time of the counting variable and the test for the terminal count by the last PE reaching the barrier. This synchronization method is SBIT per se because joining PEs must be known in advance and all but the last PE reaching the barrier code shown in Figure 2.11 must wait for the exit_flag to be set. However concurrent barriers can be accommodated by duplicating the barrier code on multiple sets of PEs, each using a different counting variable. Thus a PE can be reactivated for a different barrier. Therefore we classified it as SBDT to show that most SBIT techniques can be easily upgraded to SBDT techniques.

2.5 Static-Binding Any-Tasking Capable Techniques

So far we have not discovered any SBAT mechanism in literature. However, some mechanisms in other category can easily be adapted to SBAT capability. For example, both CV and LTD algorithms can be extended to be SBAT capable by attaching

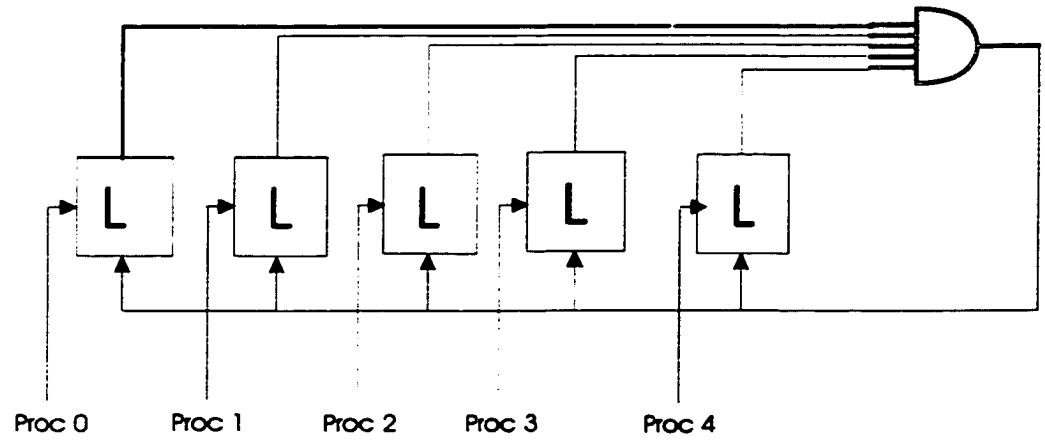


Figure 2.12: Simple AND Gate Barrier

barrier ID to each message; then multiple barriers can be executed simultaneously without ambiguity.

2.6 Dynamic-Binding Idle-Tasking Capable Techniques

2.6.1 AND Gate Barrier

Ghose and Cheng propose a simple AND gate hardware barrier [24] as shown in Figure 2.12. Each processor notifies of its arrival at the barrier by setting a local latch. This figure shows a 5 processor synchronization circuit. The block containing the symbol is a latch set by the processor when the barrier has been reached by the processor. The output of all of the latches are AND-ed together, generating a global reset signal to all latches. It is classified as DBIT. It is dynamic-binding because it does not need to know the participating PEs in advance. It is idle-tasking because every joining PE has to wait for the global reset signal after it has reached the barrier.

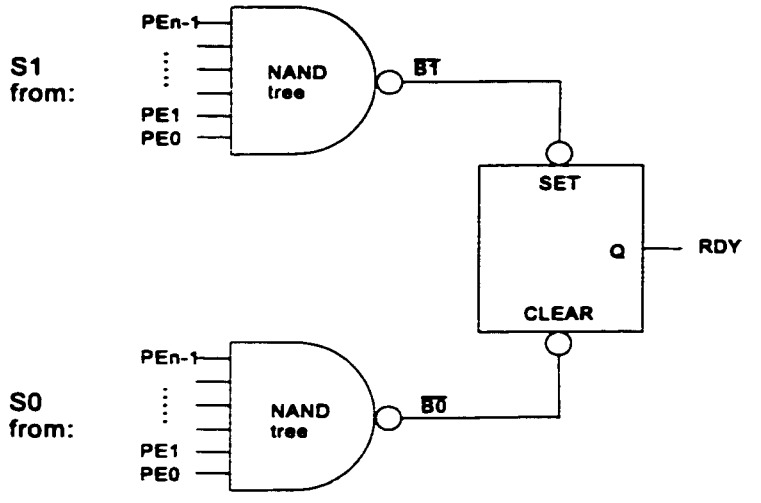


Figure 2.13: NAND Tree in TTL_PAPERS

2.6.2 TTL_PAPERS

The TTL_PAPERS [28] [40] [41] is a simple TTL hardware implementation of PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) [30]. It is plugged into the parallel ports of all personal computers in the cluster. Conceptually the TTL_PAPERS employs a AND tree to detect whether every PE has reached the barrier. However there are two serious problems with AND tree in asynchronous barrier. 1. A PE with a small task for the first barrier may reset its signal before all other PEs signal for completion of the first barrier. They end with waiting for a signal which is gone. 2. A PE which finishes the first barrier faster may set the signal high again before other PEs clear their signals for the first barrier. The TTL_PAPERS adopts the two NAND trees and a one-bit register design as in Figure 2.13 to handle the two problems. To solve the first one, it adds the one-bit register. When all PEs signal completion of a barrier, the output of the first NAND tree set the register to one; then every PE can test RDY to know whether the barrier has been reached. To solve the second problem, it adds a second NAND tree. Any PE sets signal $S0$ after

Signaling Condition	Value Submitted
The processor does not want to delete from nor insert into the shared queue.	0
The processor inserts an element into the shared queue.	-1
The processor deletes an element from the shared queue.	0
The interface processor fails to delete an element from the shared queue.	+1
PE is idle.	+1

Table 2.2: SAV Value Returned by PEs

it clears the signal $S1$. When all the PEs set its S0, the output of the second NAND tree reset the register. Any PE can enter the next barrier after it senses the RDY is zero. The TTL_PAPERS is classified as DBIT. The fact that it does not need to know the joining PEs makes it dynamic-binding. However every PE has to be committed to the barrier makes it idle-tasking capable.

2.7 Dynamic-Binding Same-Tasking Capable Techniques

2.7.1 Simultaneous Access Variable

The *Simultaneous Access Variable (SAV)* [22] technique is a simultaneous access design which provides idle processor reactivation and termination detection capabilities for the shared-memory architecture. The basic idea is: The SAV algorithm organizes PEs as a binary tree and uses a shared queue to store spawned tasks. The values sent to the SAV by every PE under different situations are tabulated in Table 2.2. Every PE keeps track the accumulated SAV of the subtree which is rooted at itself and reports it to its parent.The SAV acts like a counter which counts the difference of the idle PEs and the tasks inserted into the shared job queue. If the value of SAV is greater than zero, there are more idle PEs than the tasks inserted to the queue; otherwise there are more spawned tasks than available idle PEs. The

$U_l \leftarrow U_r \leftarrow 0$;
$R \leftarrow R_l + R_r$;
case
$R_l > 0$ and $R_r > 0$:
$U \leftarrow min(U_p, R)$;
$U_l \leftarrow |\frac{R_l}{R} \times U|$;
$U_r \leftarrow |\frac{R_r}{R} \times U|$;

$R_l > 0$ and $R_r \leq 0$:
$U_l \leftarrow min(U_p - R_r, R_l)$;

$R_l \leq 0$ and $R_r > 0$:
$U_r \leftarrow min(U_p - R_l, R_r)$;
endcase
$R \leftarrow R - U_p$;
send R to parent;
send U_l to left child;
send U_r to right child;

Figure 2.14: SAV Algorithm [22]

termination is reached when the accumulated SAV at the root equals the number of the PEs in the system. The PEs at odd and even levels execute alternately and the algorithm for each active PE is listed in Figure 2.14. Each PE maintains four sets of registers, each consisting of R and U registers. Three sets of registers are for information received from or intended for the left child, the right child, and parent respectively and they are denoted by the subscripts of l, r, and p respectively. The remaining set is for its own use. When a PE which is a left child sends R to its parent, the parent stores it in R_l; otherwise it is stored in R_r. On the other hand, the value received from the parent node is stored in U_p. The value of U_p is the number of tasks sent by its parent which may be consumed in this subtree and $U_p \geq 0$. Every PE processes according to cases based on the values of R_l and R_r. When $R_l > 0$ and $R_r > 0$, which means both child subtrees have more idle PEs than spawned tasks, tasks from its parent are shared among two children proportionally. If $R < U_p$, only R of U_p are shared; the rest are used by its parent. If $R_l > 0$ and $R_r \leq 0$, which means left child subtree has more idle PEs and right child subtree has more tasks to be consumed, the excess tasks from the right child subtree and the parent can be dispatched to the left child subtree. The case $R_l \leq 0$ and $R_r > 0$ is symmetric to the previous case. When both child subtrees have more tasks than idle PEs, i.e. when $R_l < 0$ and $R_r < 0$, the excess tasks from both child subtrees together with the tasks from the parent are dispatched to its parent. When $R \leq 0$ at the root, there are still tasks to be consumed. When $R > 0$ at the root, there are more idle PEs than the tasks to be consumed. When R at the root equals the number of PEs in the system, which means all PEs are idle, the termination has been reached. The SAV algorithm is classified as DBST. It is dynamic-binding because the joining PEs do not have to be known in advance. It is same-tasking because PEs can only be reactivated for the same barrier.

2.7.2 The Counting Algorithm

The *Counting Algorithm* [31] is a two-phase distributed termination detection algorithm. The pseudo codes for both phases are listed in Figure 2.15. A copy of the algorithm runs on every PE. All participating PEs are organized as a spanning tree. Every PE keeps track of the created and processed tasks locally with the variables n_c and n_p respectively and maintains the accumulated counts of the created and processed tasks of the subtree rooted at itself with the variables N_c and N_p respectively. In phase 1, each leaf PE sends the idle message with N_c and N_p initialized to n_c and $n_p$respectively after it turns idle. The idle message signifies that each PE in the subtree below has been idle at least once since the last idle message; in contrary to the activity message in phase 2, which is merely a report of creation and processing activities. As for the other PEs, they update the local N_c and N_p by adding the N_c and N_p sent from their children with the idle message. After receiving idle messages from all its children, a PE sends an idle message with N_c and N_p, updated with n_c and n_p respectively, to its parent when it turns idle. When the root node has received idle messages from all children and turns idle, it compares N_c and N_p. If they match, enters phase 2 because there is a very good chance that the termination has been reached. If not, restart phase 1. In phase 2, every PE sends up an activity message containing new values of N_c and N_p. These activity messages are assembled in the same way as in the phase 1. When the root has received activity messages from all children, it compares the old and new values of N_c and N_p. If they are the same, it means that there has been no new activities. The root declares termination of the barrier; otherwise restarts phase 1. The Counting Algorithm is classified as DBST. It is dynamic-binding because the joining PEs do not have to be known a-priori. It is same-tasking capable because PEs can only be reactivated to the same barrier.

Phase 1()

```
{
    N_c = 0; N_p = 0;
    wait until (RecdMsgsFromChildren());
    add to local N_c and N_p the values received from children.
    wait until (Idle());/* wait until this PE has no activation messages */
    N_c = N_c + n_c; N_p = N_p + n_p;
    if (RootSpanTree())
        if (N_c ≠ N_p)
            Broadcast message to begin Phase 1
        else
            N_c^old = N_c; N_p^old = N_p
            Broadcast message to begin Phase 2
    else
        Send message with N_c and N_p to Parent in Spanning Tree
}
```

Phase 2()

```
{
    N_c = 0; N_p = 0;
    wait until (RecdMsgsFromChildren());
    add to local N_c and N_p the values received from children.
    wait until (Idle());
    N_c = N_c + n_c; N_p = N_p + n_p;
    if (RootSpanTree())
        if (N_c^old == N_c AND N_p^old == N_p)
            Report Quiescence
        else
            Broadcast message to begin Phase 1
    else
        Send message with N_c and N_p to Parent in Spanning Tree
}
CreateMessage(){n_c++}
ProcessMessage(){n_p++}
```

Figure 2.15: Counting Algorithm [31]

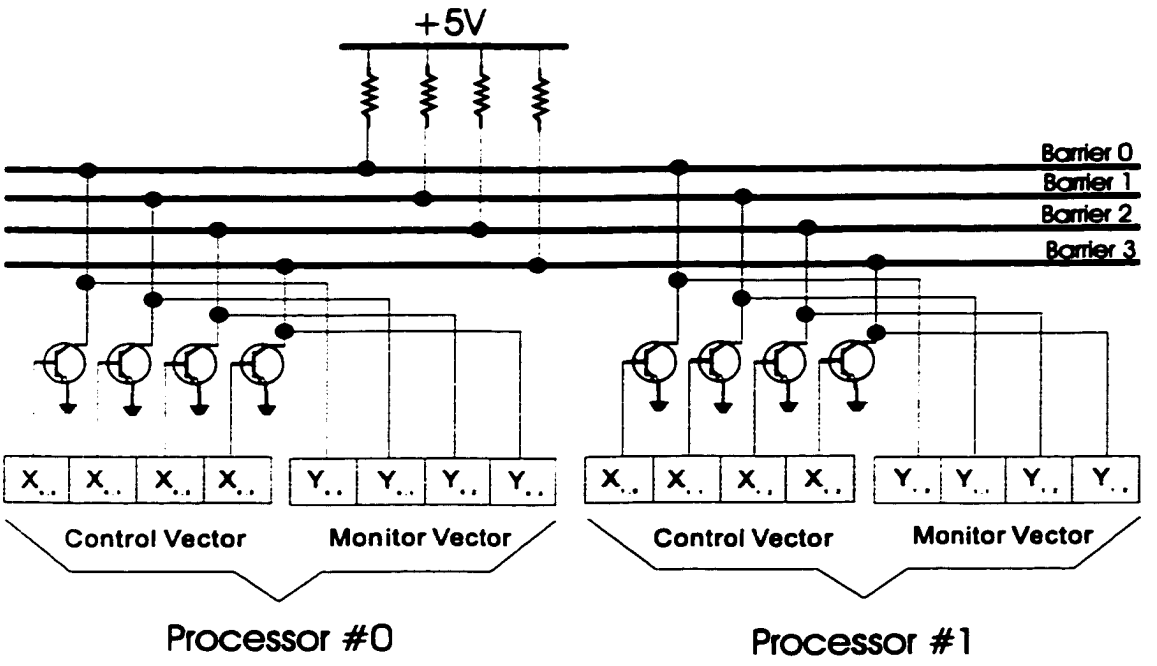


Figure 2.16: Wired NOR Barrier

2.8 Dynamic-Binding Different-Tasking Capable Techniques

2.8.1 Wired-NOR Barrier

The Wired-NOR Barrier is a distributed and hardwired barrier architecture which supports both intracluster and intercluster synchronization [26] [25]. An example supporting 4 barriers with 2 PEs is shown in Figure 2.16. The description for general case follows. There are m barrier wires and each supports an independent barrier at the same time. Physically every barrier wire is connected to n PEs, where n is the size of the system or cluster. Each PE i, where $1 \leq i \leq n$, uses a *control vector* $X_i = (X_{i,1}, X_{i,2}, \cdots, X_{i,m})$ and a *monitor vector* $Y_i = (Y_{i,1}, Y_{i,2}, \cdots, Y_{i,m})$ for synchronization control. These vectors are mapped into the shared memory or distributed to special registers in each processor board. Thus, they are program accessible from each PE. Each barrier wire, labeled as j for $1 \leq j \leq m$, is connected to n NPN bipolar

transistors [50], associated with n PEs separately. Changing a view point, every PE contains m transistors tied to m barrier wires. At each PE i, the base of each transistor is connected to a control bit $X_{i,j}$; the collector of the same transistor is monitored by a monitor bit $Y_{i,j}$. When a barrier exists, the corresponding barrier wire is pulled up to the high voltage. Any PE sets its corresponding control bit $X_{i,m}$ when it enters the barrier. That makes the associated transistor closed and pulls down the voltage the barrier wire. A PE resets its corresponding control bit when it finishes its job for the barrier. A barrier line will be pulled high again only when all transistors connected to it are reset low, which will be sensed by the monitor bit. That performs the wired-NOR logic and also means the barrier is terminated. The Wired-NOR barrier architecture is classified as DBDT. It is dynamic-binding because the participating PEs do not have to be known in advance. It is different-tasking because PEs can only be reactivated to different tasks.

2.8.2 Barrier Synchronization Register Hardware

A hardware for supporting barrier synchronization in parallel loops [27] is proposed by Beckmann and Polychonopoulus. The single barrier version supporting N PEs is shown in Figure 2.17. The R register contains a bit for each PE. As a PE completes the loops required to reach the barrier it clears its bit in the R register. The *zero detect logic*, which is a N-input NOR gate, determines when all bits in the R register are clear. The *BR*(Barrier Register) is used as a single flag to inform all the PEs of the termination of the barrier. The *BREN*(Barrier Clear Enable Register), which is ANDed together with the output of the zero detect logic, enables the automatic clear of the BR when all the R bits are 0. The mechanism of this design works as: Initially, BR=1 and R[1..N] and BREN are all 0. Every PE sets its R bit to 1 when it enters the barrier and resets its R bit after it finishes its loops for the barrier.

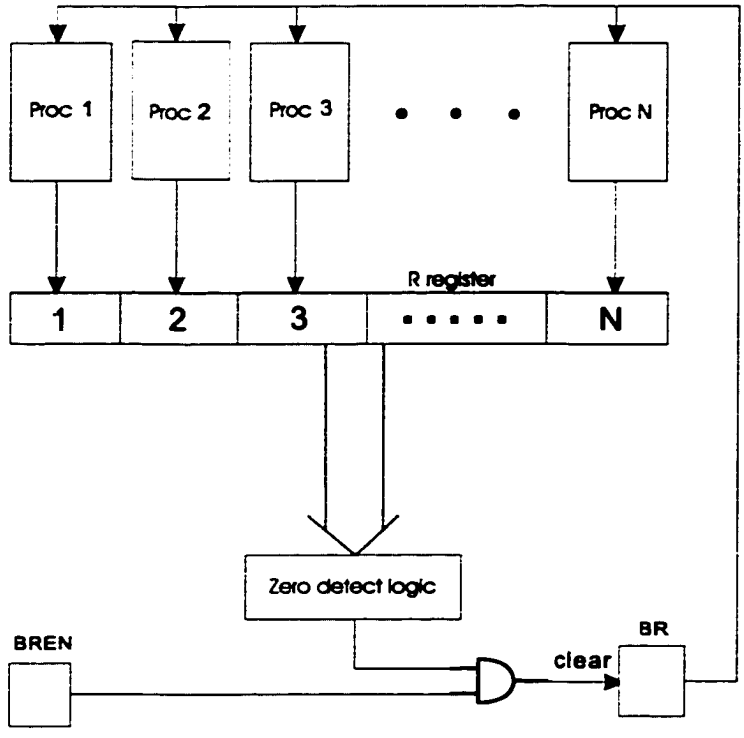


Figure 2.17: Single Barrier Register Hardware [27]

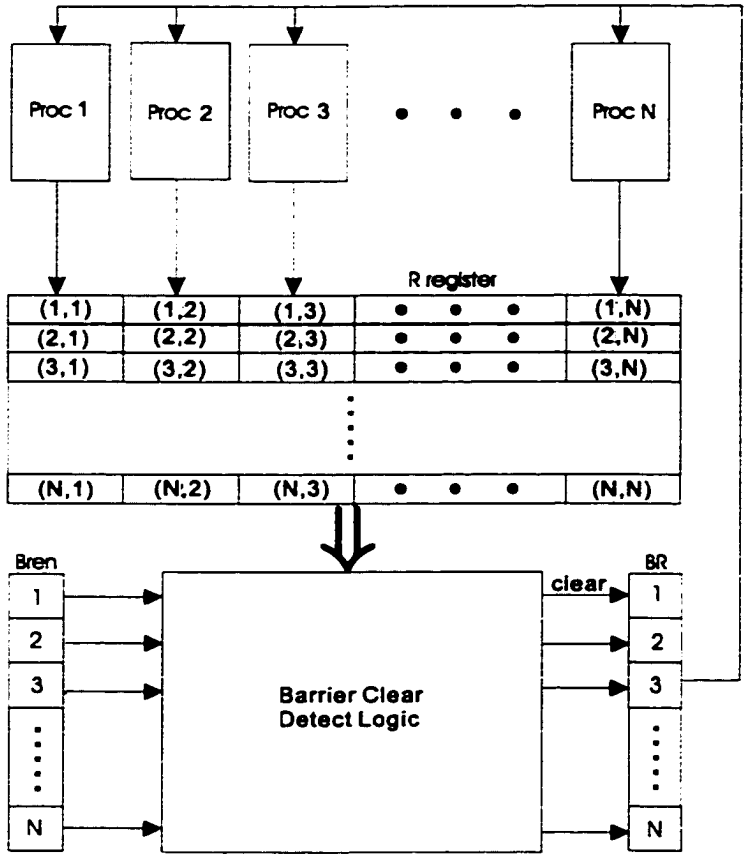


Figure 2.18: Multiple Barrier Register Hardware [27]

The PE dispatched the last iteration sets the BREN to 1 to enable the detecting of termination before executing its loops. After all PEs finish their jobs and reset their corresponding R bits, the BREN and the output of the zero detect logic triggers the clear of the BR, which in turn clears all PEs for next barrier. The multiple barrier version just duplicates the single barrier version and is shown in Figure 2.18. The Barrier Synchronization Register Hardware is classified as DBDT. It is dynamic-binding because it support arbitrary nested loops [27]. It is different-tasking capable because each PE can be reactivated for different tasks.

2.9 Dynamic-Binding Any-Tasking Capable Techniques

2.9.1 Credit Algorithm

The Credit Algorithm [9] is a global quiescence detection algorithm based on a very simple principle. There are some variants which are not necessarily better than the original design, hence only the original design is described. When the underlying computation begins, the controller which can be on either a dedicated PE or any PE distributes a credit of total value 1 to all processes. These processes either distribute part of their credit share to the new processes spawned by them or return the credit share to the controller when they finish or become *passive* as described in the original paper. The controller declares termination when it regains all the credit. To ensure the credit distribution, the algorithm follows the rules:

1. When a process becomes passive it transmits its credit share to the controller.
2. When an activating message with credit share C arrives at an *active* process, C is transmitted to the controller.

3. When an activation message with credit share C arrives at a passive process, C is transferred to the activated process.

4. When an active process with credit share C sends an activation message, the process keeps $\frac{1}{2}C$ and the message gets the other half.

Although not mentioned in the original paper, the Credit Algorithm can easily support multiple barriers by attaching barrier IDs to each credit share. Hence it is classified as DBAT. By the simple principle of credit distribution and the fact that no restrictions exist, obviously the Credit Algorithm is dynamic-binding and any-tasking capable.

2.10 Summary

Thirteen major termination detection algorithms are examined in this Chapter. They are classified by the capability category proposed in the previous Chapter. Their individual capability level can be recognized by matching their classification to the hierarchy of termination detection capability class in Figure 1.3. Therefore less capable algorithms can be compared with the more capable algorithms and differences among them can be clearly identified to make substantial improvements.

CHAPTER 3

OPTIMALITY ANALYSIS OF TERMINATION DETECTION TECHNIQUES

When it comes to find the optimality for termination detection algorithm or analyze their performance, traditionally researchers focus on message complexity. It is because determining the transit time of messages across the network is usually not as practical as the theoretical value. Moreover, messages may not arrive in the order as they were sent out. The concept that increased message traffic causes more performance degradation is correct in this aspect. However, pursuing the least message complexity only does not necessarily ensure the optimal performance as will be shown in the following research. The message delivery architecture and mechanism should also be taken into consideration to provide the optimal overall performance. Especially the messages travel inside local PEs and through the network should be clearly identified to accomplish the optimal performance because there is significant difference in the overhead to transmit both kinds of messages. The notation which will be used in the performance analysis is tabulated in Table 3.1.

Notation	Meaning
Epoch	duration of processing which occurs between barriers
N	total number of physical processing elements (PEs) in the parallel machine
N'	number of distinct PEs actively processing tasks during an epoch, where $0 \leq N' \leq N$
E	total number of events which happen in an epoch
M_i	number of internal notifications incurred in event i
T	total number of logical tasks created during an epoch
D	maximum depth of task nesting levels during an epoch
L	links between physical processing elements
t_{send}	message transit time
$t_{checkup}^{Protocol}$	time required for termination criterion checkup of specific protocol

Table 3.1: Notation used in Performance Analysis

3.1 Basis

Chandy and Misra established a lower bound of message complexity of T for termination detection algorithms[11]. They built a distributed environment model and used induction to prove that any termination detection algorithm needs to send out at least as many messages as the underlying computation messages to detect the completed barrier. This matches the intuition that every process involved in a barrier in a parallel program needs to send out at least one message to let other processes know its status. Hence T processes initiated by T underlying computation messages need at least T control messages to make other processes understand its status. Theoretically, the optimal value of messages required to detect termination of a barrier in a parallel program with T processes or messages involved is T. That can only happen in a unique case for the dynamic-binding termination detection algorithms as shown in Figure 3.1. In particular, all processes send out one message directly to the centralized control process. However, the time required for detection

is expected to be very lengthy and much longer than the theoretical value because of the network traffic caused by so many messages. In practice, it is difficult to achieve the optimal performance bound. Although the same case and an additional case as shown in Figure 3.2 apply to the static-binding termination detection algorithms, the latter will be proved inferior to other case later. Therefore optimality for overall performance will be further explored in the following sections.

3.2 Preliminary Analysis

The analysis of Chandy and Misra is based on the individual processes. Therefore the overhead of transmitting a message in their model is uniform. However in real-world parallel and distributed systems, the overhead for delivering an message inside a local node, which is designated as *internal message*, is much less than that of delivering an message out of the local node and across the network, which is designated as *external message*. Hence internal control messages and external control messages should be clearly identified to determine optimal overall performance. As a rule of thumb, opting for as many internal messages, rather than external messages, as possible will achieve more performance gain.

Assume the time to deliver control messages in our models is uniform. Consider first the case with the least control messages required; i.e., T control messages sent for T underlying computation messages. There are T processes generated by the T underlying computation messages. Every process sends out one message after it finishes its job. We hereby define the *decision process* as the process in charge of determination of termination. For a static-binding algorithm, the most fundamental form which can be achieved is: every process sends out one external control message to the centralized decision process. This case is designated as case A and shown in Figure 3.1. For simplicity of analysis, the decision process is assumed to be allocated to an

dedicated PE other than the working PEs in all cases. Apparently, the performance will suffer because all messages are external messages with much higher overhead. An intuitive thought to solve the shortcoming is to send out all control messages as internal instead of external. However, it is implausible since the information for all processes dispatched to the same PE cannot be transmitted out of the PE. There is no way that the decision process knows the status of all the other processes. Because of the characteristic of static-binding, the distributed locations of all processes are known a-priori. Taking advantage of this fact, one compromising solution can be offered. All processes reside at the same PE except one still send out one internal control message to other processes after they complete, in linear order. The last process in the chain sends out an external message to the decision process after it terminates. This case is designated as case B and shown in Figure 3.2. Any design which lies in between case A and case B has performance in between those extremes. This is because it always has less external messages than case A while having more external control messages than case B. We conclude that case B corresponds to the optimal case for the static-binding termination detection algorithms under these conditions. It will be compared with other case later.

For the dynamic-binding algorithms, the first case considered is the same as case A in the static-binding category. Due to the characteristic of dynamic-binding, there is no way to know how many processes will be created at each PE in advance. Therefore, the predecessor chain as in case B cannot be built because no process can be identified as the last one. If no process can be assigned the task to send the external control message to the decision process, the status of the processes at the local PE would not be learned by the decision process. Case B cannot occur in the dynamic-binding termination detection algorithms; neither does any situation in between case A and

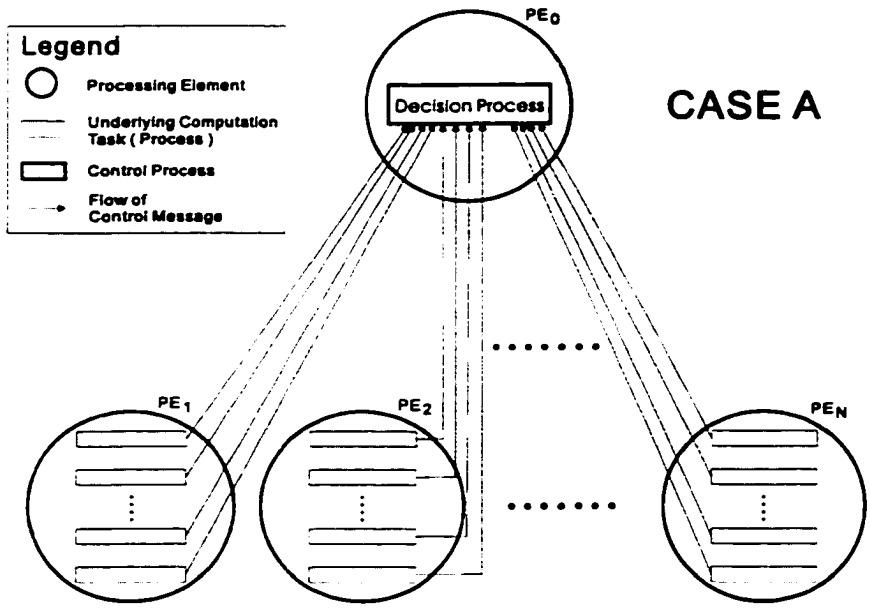


Figure 3.1: Case A for Optimality Analysis

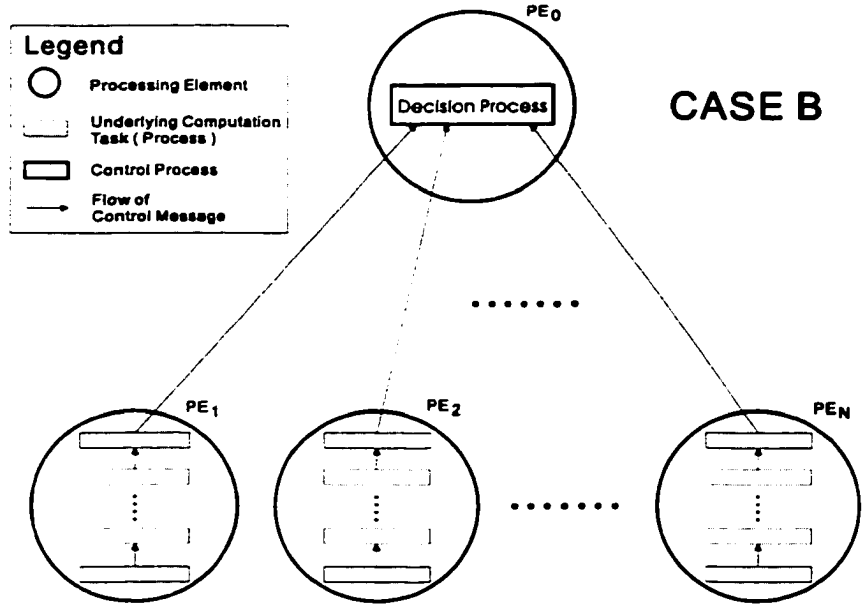


Figure 3.2: Case B for Optimality Analysis

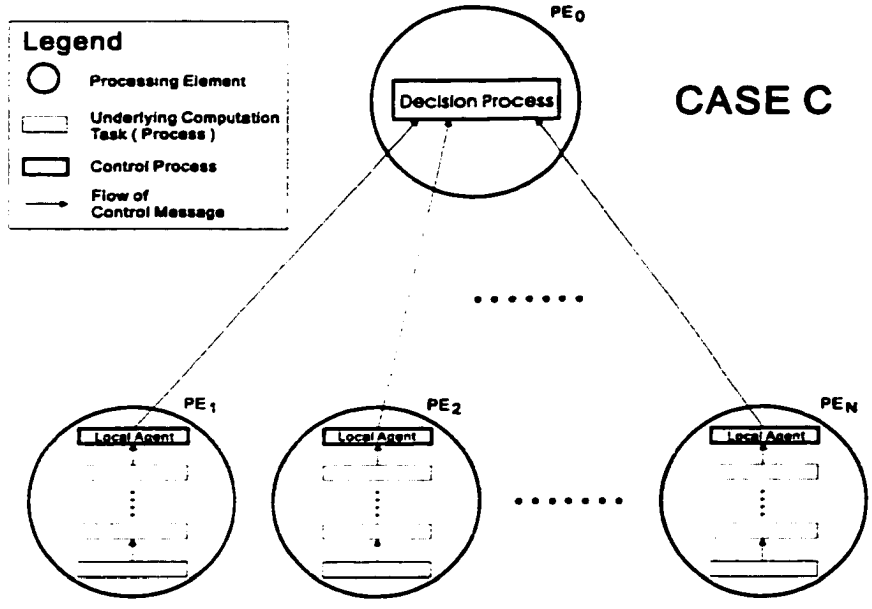


Figure 3.3: Case C for Optimality Analysis

case B because of the same rationale. Only case A is plausible. However, the performance limitations of case A are expected as previously stated.

3.3 Analysis of Optimality Cases

Since the only case for the dynamic-binding termination detection algorithms using only T control messages precipitates performance limitations in practice, let us try to adapt the control messages to improve the overall performance. Returning to the general rule of applying as many internal control messages as possible can provide an approach. In particular, let every process send out an internal control message to let others know its status. However, some *local agent processes* need to be adopted to take the responsibility of counting processes dispatched to its domain by collecting status information from all processes in its domain and sending one external control message to report the status summary to the decision process. The performance depends on how many such agent processes are necessary. Because every process can

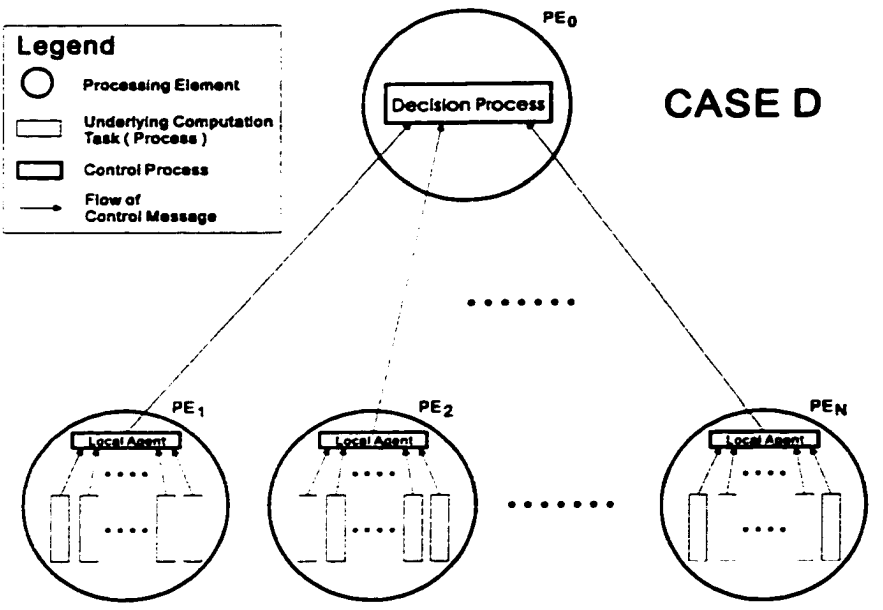


Figure 3.4: Case D for Optimality Analysis

transmit internal control message inside the boundary of a local PE, one local agent process for each PE is necessary and sufficient. Namely, the new better scheme needs $(T + N')$ control messages, where N' is the number of participating PEs in the system other than the dedicated PE where the decision process resides. Basically there are two options to implement this case. One is similar to case B and is shown in Figure 3.3. Every process at a node sends an internal control message to another process to form an information chain with the last process in the chain sending one message to the local agent process. The local agent process sends an external control message to the decision process after it receives the message from the last process. This case is designated as case C. Another option is illustrated in Figure 3.4. Every process at the same PE sends out one internal control message directly to its local agent process after it finishes its job. Since the local agent process keeps account of processes dispatched to the node, the local agent process will send an external control message to the decision process after it senses that all processes at the node are completed. This

case is designated as case D. Considering the detection delay, the time required to detect termination after the last processes in the barrier ends, case D is more efficient than case C. Because every process in case D reports directly to the local agent process, the local agent process can report to the decision process immediately after it receives the control message from the last finished process at its node. However, in case C the local agent process can report to the decision process only after it receives the control message from the last process in the information chain. If the last finished process happens not to be the last process in the information chain, the local agent process has to wait longer for the information to pass through the information chain. Apparently case D incurs less detection delay than case C under many scenarios. Thus, a lower bound for a practical implementation of tradeoffs yields message complexity for dynamic-binding termination is $(T + N')$ under these conditions. More details will be investigated later.

3.4 Optimality for Static-Binding Category

Now it is possible to re-examine whether case D is better than case B for static-binding termination detection algorithms. In case D, no matter where the last finished process is located, it sends out an internal control message to its local agent process and the local agent process in turn transmits an external message to the decision process to let it determine termination. That is to say, the detection delay after the last process ends always takes the overhead of delivering one internal and one external control messages. Yet in case B, although it requires same number of control messages, its detection delay varies with the locations of the few last finished processes. The best and only situation of case B which can outperform case D is when the last completed process is located at the end of one predecessor chain and external control messages from all PEs except the one which the last finished processes resides have

been sent out before the last process is done. This unlikely situation beats case D only by the overhead of delivering one internal message. The only case which matches the performance of case D is when the external control messages from all PEs except the one which the last finished processes resides have been sent out before the last process is done. Meanwhile, the last completed process is located at the second from the end of one information chain. This is also less likely to occur than other scenarios in general. Even the fact that the last finished process is located at the end of one information chain cannot guarantee that the detection delay is only the overhead of one external control message delivery. Because the process completed just before the last process could be at the first position of a long information chain and the traversing of the information chain might end later than the last process finishes. All the other cases are worse than those of case D. All the performance improvements are gained by adding N' additional messages. Recognizing the fact that typically $T \gg N'$ and the extra messages are all internal messages with less overhead, the little increased message complexity is worthwhile. We can conclude that the optimal case for static-binding termination detection algorithms is depicted by case D. Theorem 1 follows directly from these findings.

Thoerem 1 *When taking into account tradeoffs between internal and external messages, the lower bound for message complexity of static-binding termination detection algorithms is given by* $\min(T, N')$.

3.5 Optimality for Dynamic-Binding Category

The analysis for dynamic-binding termination detection algorithms can be similarly refined. The previously established lower bound of $(T + N')$ is based on the foundation that each PE requires one local agent process to report its status. To be

the most efficient, a dynamic-binding termination detection algorithm should support processor re-activation to fully utilize the available PEs and improve overall performance. The local agent process must report to the decision process once the local PE becomes idle because there is no way to know whether more processes will be dispatched to the local PE without incurring more notification messages. These extra messages will not only increase message complexity, but also deteriorate the overall performance since they are all external control messages. If any PE is re-activated, then its local agent process needs to report one additional message for this re-activation event after the local PE turns idle again. Therefore, the overall number of control messages for all the local agent processes to report can be tightened to E, the number of events, rather than N', the number of participating PEs in the system. Theorem 2 follows directly from these results.

Thoerem 2 *When taking into account tradeoffs between internal and external messages, the lower bound for message complexity of dynamic-binding termination detection algorithms is given by* $\min(T, E)$.

CHAPTER 4

TIERED DETECTION ALGORITHM

In this chapter we propose a software-based distributed termination detection algorithm, the Tiered Detection Algorithm [32] [16], which solves the spawn-in-transit problem by applying a global invariant without introducing significant overhead during termination detection.

4.1 Overview

The basic idea of the Tiered Algorithm is described below. A node is designated as the centralized *controller* to keep track of the global status. Every participating PE reports the amounts of the locally consumed and locally produced tasks at each level of process nesting to the controller whenever it becomes idle. The controller updates the records it keeps accordingly. After all PE's have turned idle and reported, the controller determines whether the global consumption count and production count at each level of nesting match. If they do, then the controller announces global termination. Otherwise, it waits for the next round of checking. Obviously, the reporting action is processor-centered, contrary to process-centered, to reduce message traffic as suggested in Chapter 3. The controller can maintain the difference of the amounts of the consumed tasks and the produced tasks. No difference between these

quantities means that all spawned tasks are consumed, hence no tasks are in transit and the global termination is reached. However, at least one scenario can induce false termination detection if level of thread nesting is not considered. It is described below.

Assume a two processor system for simplicity. After the parallel program has executed for a while, the current status sensed by the controller is: PE_1 is busy with the only task spawned to it while PE_2 is idle. The difference of the amounts of the consumed and produced tasks known by the controller is one which means one spawned task is unfinished. Then the task at PE_1 spawns a task to PE_2. PE_2 processes the task which in turn spawns a task back to PE_1. Before PE_2 finishes its job and reports to the controller, PE_1 completes the tasks and report "*consumed = 2, produced = 1*" to the controller. The controller will erroneously declare false global termination based on the fact that all PE's are idle and the consumed and produced tasks match in amount while PE_2 is still processing.

To remedy the false detection issue, task hierarchy information is required along with the consumption count and production count. For example in the false termination detection scenario, if the two consumed tasks are distinguished as in the k_{th} level and $(k+2)_{th}$ level respectively and the produced task as in the $(k+1)_{th}$ level, they will not be accounted together as to precipitate erroneous detection. The process creation message of the underlying computation will carry the level number information. Every PE needs to keep track of the consumption count and production count for each level of the logical tree of the process. The PE reports these to the controller whenever it turns idle. When the controller checks for termination, if the amount of the consumed tasks for each level equals the amount of tasks spawned by its parent level, then the global termination has been achieved. The detailed algorithm is described in the following sections.

Procedure: *Receive_TaskSpawn_Message*(l : level number)
begin
    Update local activity table accordingly;
end

Procedure *Finish_A_Task*(l : level number)
begin
    Update local activity table accordingly;
end

Procedure *Upon_Idle*
begin
    Report non-zero difference in local activity table to controller;
end

Figure 4.1: Operation of the Processing Element in Tiered Detection Algorithm

4.2 Operation of the Processing Element

The algorithm for the local PE is given in Figure 4.1. Every PE needs to maintain an *activity table* as shown in Figure 4.2a, which records the local consumption count and production count for each level. The consumption count stands for the number of tasks which are consumed for any specific level at this local PE. Likewise, the production count means the number of tasks spawned by any specific level at the local PE. One relationship employed with the production count is that the tasks dispatched by the k_{th} level are also the tasks created on the $(k+1)_{th}$ level. Since we are really interested in whether the amount of tasks spawned to a specific level matches the amount of tasks consumed at the same level, it is sufficient to maintain the difference of the two amounts for each level. Furthermore, the number of quantities communicated are reduced. Hence we will maintain a one-dimension array as shown in Figure 4.2b for the difference of the local consumption count and production count for each level, which is still referred to as the activity table to reflect its function. Whenever a

Level	Consumption Count	Production Count
0	0	4
1	4	6
2	6	8
. . . . .		
(D-1)	5	7
D	7	0

(a)Theoretical

DIFF(1)
DIFF(2)
. . . . .
DIFF(D-1)
DIFF(D)

(b) Implementation

Figure 4.2: Activity Table

creation message is received by a PE, the procedure *Receive_TaskSpawn_Message* is called to update the local activity table in accordance to the level number accompanying the creation message, i.e. increment the number in the corresponding cell by one. Likewise, the procedure *Finish_A_Task* is called whenever a task is completed at a PE to refresh the local activity table according to the level number which the finished task belongs, i.e. decrement the number in the corresponding cell by one. The consumption count and production count are kept current by the execution of the procedures Receive_TaskSpawn_Message and the Finish_A_Task. After the PE finishes all the tasks in its execution queue and turns idle, the procedure *Upon_Idle* is activated to report the difference of the amounts of the consumed tasks and produced tasks for each level to the controller. To reduce network traffic, only levels with nonzero value are reported. After reporting, any PE can be reactivated by the creation message of the underlying computation.

4.3 Operation of the Controller

The algorithm for the controller is given in Figure 4.3. The controller maintains a *ledger table* to keep track of the global consumption count and production count by the information reported by all PE's. For the same rationale as for the activity table for a PE, a one-dimension array serves as a ledger table well, i.e. we maintain only the difference of the consumption count and production count for each level. Whenever a PE reports to the controller, the controller calls the procedure *Receive_Report* to respond. It updates the ledger table according to the information sent in by the reporting PE. That is, increase or decrease the number in the corresponding level cell of the ledger table by the amount reported. Then it checks the ledger table. If values of the differences in all cells of the ledger table are equal to zeroes, which means all tasks spawned to all levels are consumed; then the global termination has been

```
Procedure Receive_Report(r : report)
begin
      Update ledger and idle table accordingly;
      If(Check_Ledger)
      Then
            declare global termination
      End if;
end


Procedure Check_Ledger
begin
      Check ledger table to determine if consumption
      and production counts of every level match;
      If yes, report TRUE
      else report FALSE
      end if;
end
```

Figure 4.3: Operation of the Controller in Tiered Detection Algorithm

reached. If the value of any cell in the ledger table is not zero, which means that there are messages still in transit, then the controller exits the procedure and waits for the next report.

4.4 Performance Analysis and Comparison

In this chapter, the performance of the Tiered Detection Algorithm is analyzed and compared against the other three of the more effective termination detection algorithms which can be found in literature, namely *Credit Algorithm*, *CV Algorithm*, and *LTD Algorithm*. The performance of the Tiered Detection Algorithm and the LTD Algorithm depend greatly on the mapping of tasks in an epoch.

However, determining scenarios used for mapping does not produce deterministic results [14][17][15][48][4]. For example, when same batch of tasks and same algorithm

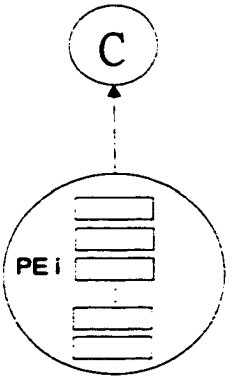
are applied to different distributed systems with different processing speed component nodes, mapping outcomes differ. The fact makes it difficult to do absolute quantitative analysis. Hence we will apply different approaches where appropriate to do relative quantitative analysis. Four aspects of performance will be analyzed; namely message complexity, bit complexity, detection delay, and space complexity [43] [5].

4.4.1 Notation and Assumptions

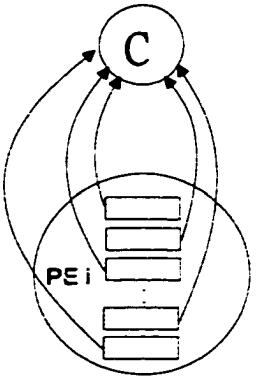
The notation used in the analysis of these distributed termination detection algorithms is the same as tabulated in Table 3.1.

The following assumptions are extensively used in the analysis of these algorithms. We state them here to prevent ambiguity and redundant explanation in the future. The mapping of a distributed application plays a decisive role for the performance of some termination detection algorithms. For the comparisons to be made in the following sections, all the mapping cases for the four algorithms in the same category comparison are the same. That is, the mappings of tasks to physical PEs are the same for all algorithms no matter how they are mapped logically in each algorithm.

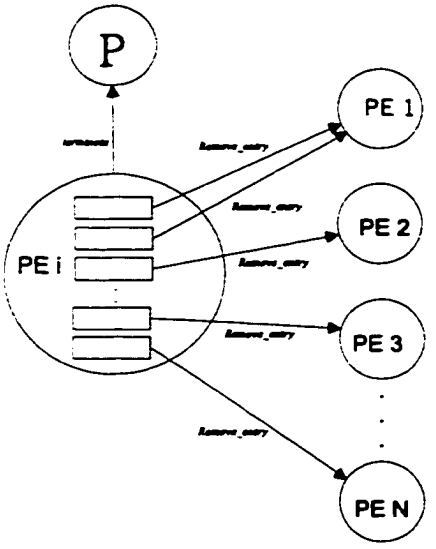
In the four algorithms to be analyzed, each of them needs to attach some information to the initializing messages of the underlying computation. The Tiered Algorithm attaches level number, the Credit Algorithm attaches credit, the CV Algorithm attaches PE ID, and the LTD Algorithm attaches PE ID. Since the information are appended to the existing messages and do not incur new messages, those messages will not be accounted for. We count only the new messages generated by the termination detection algorithms.



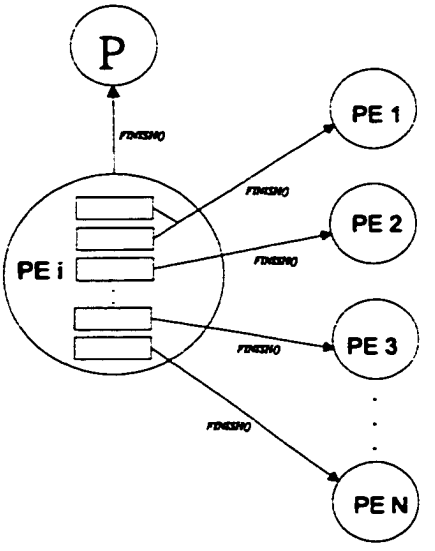
(a) Tiered Algorithm



(b) Credit Algorithm



(c) CV Algorithm



(d) LTD Algorithm

Figure 4.4: Messages Sent After the PE turns Idle

4.4.2 Message Complexity

Message complexity accounts for the number of messages required to detect termination. As stated in Chapter 3, the overhead of delivering external messages is much larger than that of internal notifications. Therefore, we should focus on the total number of external messages required for an algorithm instead of overall messages required because it is the number of external messages that dominates the performance of a termination detection algorithm. As mentioned in the beginning part of this section, the mapping of tasks in a distributed application is nondeterministic; the combination of mappings is large. Therefore we need to use an event-based approach to make a relative comparison. *Event* is defined here as the process that tasks are allocated to a specific PE, all tasks are executed by the PE, and the PE turns idle. A distributed application is achieved by execution of its component events. In the Tiered Algorithm, when an event on a specific PE ends, the PE needs to send one external message to the controller as seen in Figure 4.4. However before that happens, every task existing in this single event must send one internal notification to notify the local agent process of its completion. Let M_i denote the number of internal notifications in event i. Eventually $\sum_{i=1}^{E} M_i = T$ internal notifications are required for T tasks in the epoch. Since there are E events in the epoch, E external messages are required. Overall, $(T + E)$ messages are required by the Tiered Detection Algorithm. In the Credit Algorithm, each task sends one external message to the controller after it is finished. In any event, the number of external messages sent to the controller is as many as the number of tasks in the event as shown in Figure 4.4b. Since these external messages are sent directly to the centralized controller process, there is no need to adopt local agent processes nor are internal notifications necessary. Hence the overall number of messages required by the Credit Algorithm is still T. However its performance would not be optimal because of the heavy network traffic caused by

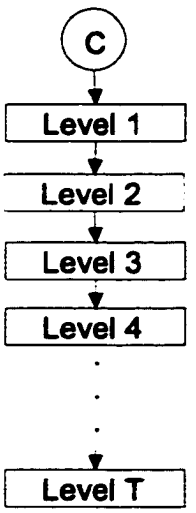
the external messages. In the CV Algorithm as shown in Figure 4.4c, every task in an event needs to send a external *remove_entry* message to its sender; the total is M_i for event i. In whole, $\sum_{i=1}^{E} M_i = T$ external *remove_entry* messages are generated. The PE where the event resides needs to send a *terminate* message to its logical parent, $(N-1)$ external messages are required for $(N-1)$ child PEs. However, $(N-1)$ messages instead of $(E-1)$ messages are needed here not because it requires less messages than other algorithms, but that the CV Algorithm does not support PE reactivation. Combined with $2L$ external messages to build the logical tree of PE, $(2L+T+N-1)$ external messages are needed for the CV Algorithm. Because a child PE is required to send a *terminate* message to its parent PE after it becomes idle, apparently every task in an event needs to send one internal notification to let the local agent process know of its completion so that the local agent process knows when to send out the *terminate* message. The number of these internal notifications amounts to T for T tasks in the epoch. Since the LTD Algorithm is an improvement over the CV Algorithm, it usually needs less external messages than the CV Algorithm does. The number of messages required depends on the mapping of the tasks. As shown in Figure 4.4d, some tasks are spawned by the same PE, the event needs to report to the spawning PE with only one $FINISH$ message instead of several messages as in the CV Algorithm. This is also the largest improvement over the CV Algorithm. In the worst case, every task in any event is spawned by different PE; the performance is degraded to the level of the CV Algorithm, i.e., $\sum_{i=1}^{E} M_i - 1 = (T-1)$ external *FINISH* messages are generated. In the best case, the performance matches that of the Tiered Algorithm; i.e., every task in an event are spawned by the same PE, therefore only one external message is reported by the event to the spawning PE. Finally $(E-1)$ *FINISH* messages are required for E events except for the event happening on the root node of the tree of PE. Counting the $(N-1)$ external messages required

Algorithm	External Messages	Internal Notifications	Total Messages
Tiered Algorithm	E	T	$E+T$
Credit Algorithm	T	0	T
CV Algorithm	$(2L+T+N-1)$	T	$(2L+2T+N-1)$
LTD Algorithm	from $(N+E-2)$ to $(N+T-2)$	T	from $(N+T+E-2)$ to $(N+2T-2)$

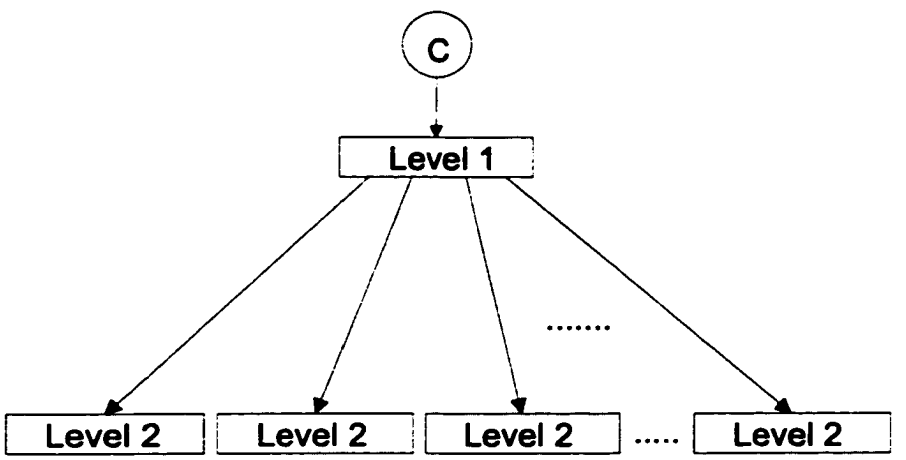
Table 4.1: Comparison of Message Complexity

to inform the DT status, the best case generates $(N+E-2)$ external messages while the worst case generates $(N+T-2)$ external messages. As for the internal notification, every task needs to inform its local agent process of its completion so that the local agent process senses that the local PE is idle and sends out *FINISH* messages to its parent node and/or other PEs. The required number of internal notifications amounts to T no matter in what case. The overall number of messages required by the LTD Algorithm ranges from $(T+N+E-2)$ to $(2T+N-2)$. Those results are summarized in Table 4.1.

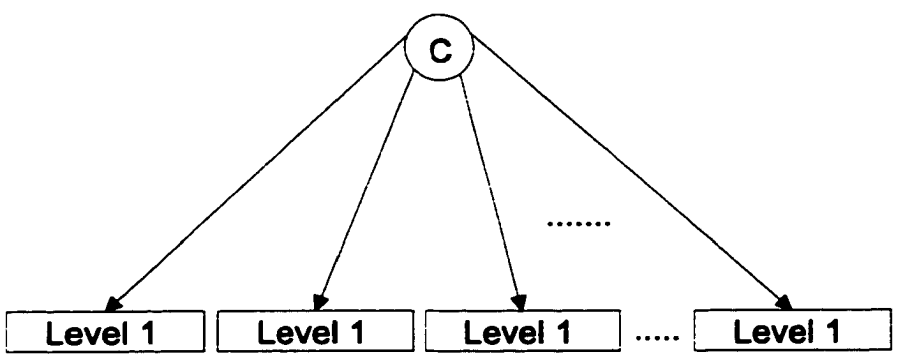
The Tiered Detection Algorithm performs best with the same number of messages as the number of events in the epoch. The Credit Algorithm needs the number of messages as many of number of tasks. The CV Algorithm needs more than the number of tasks. The LTD Algorithm's performance lies in between.



(a) Extreme Dispatching Case 1



(b) Extreme Dispatching Case 2



(c) Extreme Dispatching Case 3

Figure 4.5: Extreme Dispatching Patterns for Tiered Detection Algorithm

4.4.3 Bit Complexity

Bit complexity accounts for the number of bits of the messages required to perform termination detection. In the Tiered Algorithm, every report consists of two fields, namely level number and difference of the production count and consumption count in the matching level. The maximum level number of an epoch with T tasks is T when all tasks are dispatched to different levels; as shown in Figure 4.5a. Hence $\lceil \lg T \rceil$ bits are required. The maximum difference which can be contained in one level of an epoch with T tasks is $(T-1)$. That happens when the only task spawned by the controller spawns all the remaining $(T-1)$ tasks to the second level as shown in Figure 4.5b. When only the first level task is dispatched to a PE as an event, the PE needs to report the difference of $(T-1)$ for level 2. We still count that $\lceil \lg T \rceil$ bits are required for the difference field for simplicity. Therefore a basic report unit requires $2\lceil \lg T \rceil$ bits. The worst case happens when all tasks are dispatched to different levels of the logical tree and are physically allocated to the PE's in a way that no two tasks in adjacent levels are dispatched to the same PE, same as the case shown in Figure 4.5a. In that case, the PE needs to report "one consumed and one spawned" for every finished task because no two tasks from adjacent levels are dispatched to a same PE; eventually $2T$ basic report units are required to cover the T finished tasks. The worst case takes $4T\lceil \lg T \rceil$ bits. The best case happens when all tasks are dispatched to the first level, as shown in Figure 4.5c. Since all tasks are in the first level, all tasks dispatched to the same event take only one basic report unit to report the amount consumed. Finally, E basic report units are required to cover all consumed tasks dispatched to the E events. The best case takes $2E\lceil \lg T \rceil$ bits.

In the Credit Algorithm, the capacity of the message needs to accommodate the extreme case when all tasks are dispatched to different levels as shown in Figure 4.6. As the algorithm uses a local integer variable CREDIT to stand for the value of

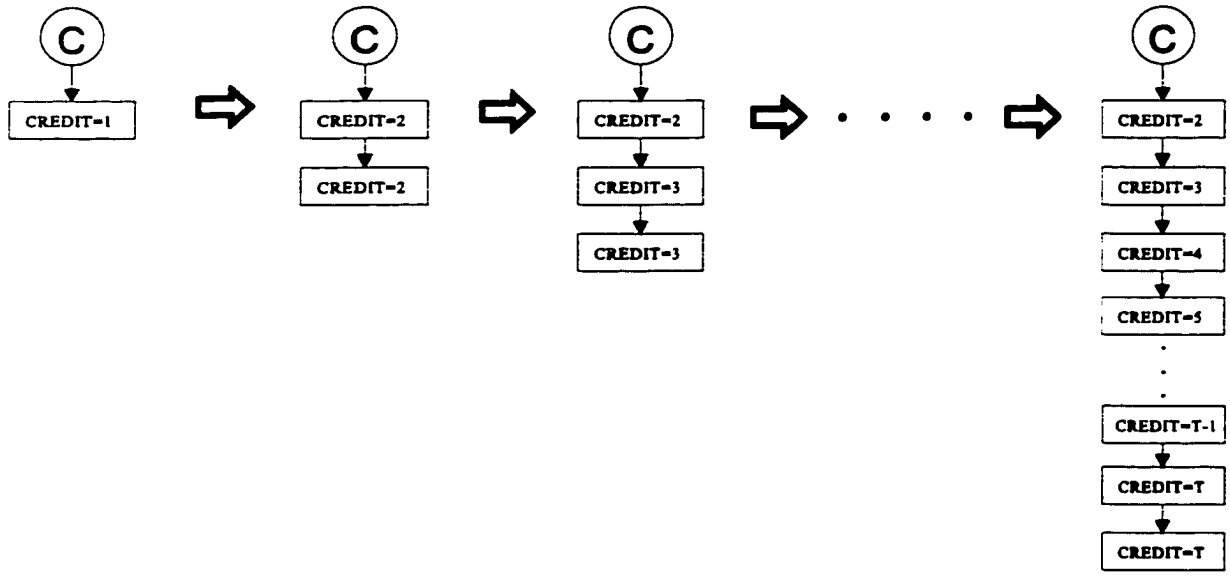


Figure 4.6: Extreme Dispatching for Credit Algorithm

$2^{-CREDIT}$, $\lceil \lg T \rceil$ bits are sufficient to represent the smallest credit share. Since the messages required are T as stated in previous section, $T\lceil \lg T \rceil$ bits are required in an epoch. In the CV Algorithm, the message needs to identify the identification of the PE which it comes from and what kind of message it is; hence we assume the message consists of two fields: PE ID and message ID. There are N PEs joining the operation; $\lceil \lg N \rceil$ bits are sufficient to represent all the PEs. There are only three kinds of messages in the algorithm; two bits are sufficient to identify them. The CV Algorithm needs $(2L+T+N-1)$ messages in any case; hence $(2L+T+N-1)(\lceil \lg N \rceil + 2)$ bits are necessary. As for the LTD Algorithm, two fields are sufficient: message ID and amount. There are only two kinds of messages in the algorithm; one bit is enough. The amount field which represent the messages being reported by the $FINISH(n)$, needs $\lceil \lg T \rceil$ bits because the greatest possible number of messages being reported is T. Hence the bits required by the LTD Algorithm ranges from $(E+N-1)(\lceil \lg T \rceil + 1)$ to $(T+N-1)(\lceil \lg T \rceil + 1)$. The results are summarized in Table 4.2.

Algorithm	Best Case	Worst Case	Complexity
Tiered Algorithm	$2E\lceil \lg T \rceil$	$4T\lceil \lg T \rceil$	$O(T \lg T)$
Credit Algorithm	$T\lceil \lg T \rceil$	$T\lceil \lg T \rceil$	$\Theta(T \lg T)$
CV Algorithm	$(2L + T + N - 1)\times$ $(\lceil \lg N \rceil + 2)$	$(2L + T + N - 1)\times$ $(\lceil \lg N \rceil + 2)$	$\Theta(T \lg N)$
LTD Algorithm	$(E + N - 1)\times$ $(\lceil \lg T \rceil + 1)$	$(T + N - 1)\times$ $(\lceil \lg T \rceil + 1)$	$O(T \lg T)$

Table 4.2: Comparison of Message Bit Complexity

Looking at the complexity of bit in Table 4.2, we can find that the Credit Algorithm performs the worst with a complexity of $\Theta(T \lg T)$. This indicates that it always needs $(T \lg T)$ bits. The CV Algorithm is slightly better than the Credit Algorithm with a complexity of $(T \lg N)$, however it still always needs $(T \lg N)$ bits. The Tiered and LTD Algorithms are better than the other two with a complexity of $O(T \lg T)$, which means that chances are that they need less than $(T \lg T)$ bits. Comparing the best and worst cases of the Tiered and the LTD Algorithms, we can further find that the LTD Algorithm usually needs less bits than the Tiered Algorithm.

4.4.4 Detection Delay

Detection delay accounts for the interval between when the last task ends and the controller or the root node concludes global termination. Two new kinds of notation are introduced for these quantities. Although the time necessary to send a message across the network depends on the state of the network and is usually variable, we

Figure 4.7: Dispatching for the Worst cases of CV and LTD Algorithms

always designate it as t_{send}. The procedure which every algorithm uses to check global termination is different and takes different amounts of time; we designate $t_{checkup}^{protocol}$ to represent the time taken by the execution of each checkup procedure for a given protocol. In the Tiered Algorithm, in all cases, after the last task ends the resided PE sends a report to the controller; the controller checks up the status and concludes global termination. The detection delay is $(t_{send} + t_{checkup}^{Tiered})$. The quantity $t_{checkup}^{Tiered}$ is bounded as follows. $t_{add} + t_{compare} \leq t_{checkup}^{Tiered} \leq T \cdot t_{add} + D \cdot t_{compare}$. In the Credit Algorithm, same as the Tiered Algorithm, the resided PE sends a report to the controller after the last task ends; the controller executes checkup procedure and concludes global termination. The detection delay is $(t_{send} + t_{checkup}^{Credit})$. For $t_{checkup}^{Credit}$, the credits are kept in a set [9].

As for the CV Algorithm, the detection delay is variable and depends on the location of the last task in the physical tree of PE. The worst case happens when tasks are dispatched as shown in Figure 4.7, i.e. only one task to each of the first $(N-1)$ PEs and the rest to the last PE in the tree of PEs, and the last ending task

resides in the last PE. After the last task ends, the last PE needs to send $(T-N+1)$ *remove_entry* messages serially first, which takes $(T-N+1)t_{send}$; then it checks its status up and sends *terminate* to its parent. Its parent also checks up its status and sends *terminate* to its own parent. This process goes on on every PE except the root PE of the physical tree, which takes $(N-1)(t^{CV}_{checkup}+t_{send})$. Receiving the *terminate* message from its child, the root PE checks up the status and concludes global termination, which takes $t^{CV}_{checkup}$. Summing up, the detection delay for the worst case is $(Tt_{send}+Nt^{CV}_{checkup})$. The best case happens when the last ending task residing in the root PE. The root PE checks up the status and concludes global termination. The detection delay is $t^{CV}_{checkup}$. In the LTD Algorithm, the situation is very similar to that of the CV Algorithm and also depends on where the last ending task is located. The worst case happens when the tasks are dispatched as in Figure 4.7 and the last ending task resides in the last PE. The scenario is slightly different from that of the worst case of the CV Algorithm: Same as the CV Algorithm, only one task is dispatched to each of the first $(N-1)$ PEs, the rest to the last PE in the tree of PEs. However every PE's major message comes from its parent in the physical tree of PEs. After the last task ends, the last PE in the tree sends one $FINISH()$ message to each of the $(N-1)$ PEs above it, which takes $(N-1)t_{send}$. Since the PE above the last PE sent the major message to the last PE, the last $FINISH()$ message from the last PE is sent to its physical parent as in Figure 4.7. After receiving the message, the second PE from the last checks up its status and sends a $FINISH()$ message to its own parent, which takes $(t^{LTD}_{checkup}+t_{send})$. All the PEs above the last PE in the hierarchy except the root PE take the same action. That takes $(N-2)(t^{LTD}_{checkup}+t_{send})$. The root PE only checks up its status and declares global termination, which takes only $t^{LTD}_{checkup}$. Summing up, the detection delay for the worst case of the LTD Algorithm is $(2N-3)t_{send}+(N-1)t^{LTD}_{checkup}$. The best case happens when the last ending task

Algorithm	Best Case	Worst Case	Complexity
Tiered Algorithm	$(t_{send} + t_{checkup}^{Tiered})$	$(t_{send} + t_{checkup}^{Tiered})$	$\Theta(1)$
Credit Algorithm	$(t_{send} + t_{checkup}^{Credit})$	$(t_{send} + t_{checkup}^{Credit})$	$\Theta(1)$
CV Algorithm	$t_{checkup}^{CV}$	$(Tt_{send} + Nt_{checkup}^{CV})$	$O(T)$
LTD Algorithm	$t_{checkup}^{LTD}$	$(2N-3)t_{send}$ $+(N-1)t_{checkup}^{LTD}$	$O(N)$

Table 4.3: Comparison of Detection Delay Complexity

resides in the root PE. After the last task ends, the root PE checks up its status and concludes global termination, which takes only $t_{checkup}^{LTD}$. The results are summarized in Table 4.3.

Apparently both the Tiered and the Credit Algorithms performs the best with a complexity of $\Theta(1)$. The CV Algorithm has the worst performance. The LTD Algorithm lies in between the other three algorithms.

4.4.5 Space Complexity

The space complexity accounts for the memory space required by the mechanism of each algorithm. We assume that all four algorithms use fixed memory allocation instead of dynamic memory allocation to save the execution overhead. In the Tiered Algorithm, the controller needs to maintain the ledger table while every PE needs to maintain an activity table. For the ledger table, we reserve T record space in the table for possible T levels in the worst case. Because the index of the record in the table can serve as the level number implicitly, there is no need to set a field for the level

number; the difference field is enough. The largest possible number for the difference is $(T-1)$, hence $\lceil \lg T \rceil$ bits are sufficient for one record. Eventually $T\lceil \lg T \rceil$ bits are required for the ledger table. As the ledger table and the activity table are the same thing, $NT\lceil \lg T \rceil$ bits are needed for N tables at N PEs. Finally $(N+1)T\lceil \lg T \rceil$ bits are required for the Tiered Algorithm. In the Credit Algorithm, in order to avoid underflow problems and process exponents practically, the Credit algorithm proposes a debts bookkeeping technique. It lets $K = CR$ for each task and maintains a $DEBTS$ set, which contains K for every active task. Whenever a task becomes passive and returns its credit share, the controller deducts it from the $DEBTS$ set. When $DEBTS$ becomes empty, termination is concluded. The controller needs space to maintain the set. The worst case is the same as Figure 4.6; when all T tasks are active and the largest $K = T$. Therefore it needs $T\lceil \lg T \rceil$ bits to accommodate the worst case. As for the CV Algorithm, every processor maintains a stack to record sending and receiving activities. The stack must be big enough to accommodate $(T-N+1)$ records which are $\lceil \lg N \rceil$ bits wide each in the worst case that all the other $(N-1)$ processors send the remaining messages in the epoch aside from the messages spawning them to the same processor. The space required is $N(T-N+1)\lceil \lg N \rceil$ bits in total fro N PEs. Hence the space complexity is $O(NTlogN)$. In the LTD Algorithm, every node has to maintain four variables. The first, in_i, needs $(N-1)\lceil \lg T \rceil$ bits. The second, out_i, needs $\lceil \lg T \rceil$ bits. The third, $mode_i$, needs 1 bit. The last, $parent_i$, needs $\lceil \lg N \rceil$ bits. The total is $(N\lceil \lg T \rceil + \lceil \lg N \rceil + 1)$ bits for each PE. Hence N joining PEs need $N(N\lceil \lg T \rceil + \lceil \lg N \rceil + 1)$ bits. The results are summarized in Table 4.4.

The Tiered Algorithm requires the most space. The CV Algorithm needs slightly less space than it. The Credit and the LTD Algorithms requires much less space than the other two. The Credit Algorithm needs only $\frac{1}{N}$ of the bits required by the Tiered

Algorithm	Space Required	Complexity
Tiered Algorithm	$(N+1)T\lceil \lg T \rceil$	$\Theta(NT \lg T)$
Credit Algorithm	$T\lceil \lg T \rceil$	$\Theta(T \lg T)$
CV Algorithm	$N(T-N+1)\lceil \lg N \rceil$	$\Theta(NT \lg N)$
LTD Algorithm	$N(N\lceil \lg T \rceil + \lceil \lg N \rceil + 1)$	$\Theta(N \lg T)$

Table 4.4: Comparison of Aggregate Space Complexity

Algorithm. The LTD performs the best with only $\frac{1}{T}$ of the bits needed by the Tiered Algorithm.

4.5 Software Design Optimizations

The Tiered Detection Algorithms is more efficient than the other three algorithms in message complexity, bit complexity, and detection latency respects. Its advantages are gained by some optimizations in software design which cannot be recognized by the performance analysis alone. First, the scheme adopts to report the global invariance of equal production count and consumption count, which eliminates the necessity to understand other PEs' status. That saves either the inquiry messages to other nodes or the informing messages from other nodes. Attaching level number to each task provides the controller a way to uniquely recognize production count and consumption count without false detection. The two factors makes it possible that any local node needs only to report to the controller without communicating with other nodes, which greatly reduces the costly external messages. The choice of processor-centered

reporting activities further cuts the number of required reporting messages than that of process-centered reporting. All those makes the Tiered Detection Algorithm conform to the practically optimal message complexity as predicted in the optimality analysis. The adoption of production count and consumption count with attached level numbers also helps in minimizing detection latency. Since that makes the last finished task able to report directly to controller without traveling through the logical tree structure as in the CV Algorithm and LTD Algorithm. The practice of applying the difference of production count and consumption count instead of individual production count and consumption count cuts the bit complexity almost in half. That effectively reduces the bit requirement.

4.6 Summary

Judging from the previous performance analysis, the Tiered Detection Algorithm outperforms the other three algorithms in message complexity, bit complexity, and detection latency by a tradeoff in space complexity. The former three factors dominate the performance of a termination detection algorithm while the latter factor is merely a cost factor. The cost difference is negligible with the fact that RAMs are very cheap and affordable. Therefore, the Tiered Detection Algorithm proves to be a high performance termination detection algorithm in terms of software-based approaches.

CHAPTER 5

DISTRIBUTED-SUM BIT-COMPARISON LOGIC

5.1 Overview

The *Distributed Sum Bit Comparison (DSBC) logic* configuration for a system with n PEs is shown in Figure 5.1. It consists one instance of the *Global Logic* which can reside at either an independent node or any one local node, and n instances of the *Local Logic*, one at each node. The Global Logic configuration consists of a *Responder Count Encoder* , a *Decision Module*, and *Global Control* signal source. Each Local Logic configuration is comprised of a *Summation Module*, a dual-port random-access memory (RAM), and a *Reporting and Recording Module*. The Local Logic at each node keeps a ledger of the task count produced or consumed by each thread on a single PE. The value is stored in the dual-port RAM. The Global Logic will demand the local task counts from the Local Logic units and performs a summation of local counts to evaluate if the present snapshot of all PEs in the system satisfies the barrier criterion for having an equal number of produced and consumed tasks.[32] The result will be sent back to each Local Logic configuration by a 1-bit signal. Each Local Logic configuration will respond depending on the result by either storing the completed

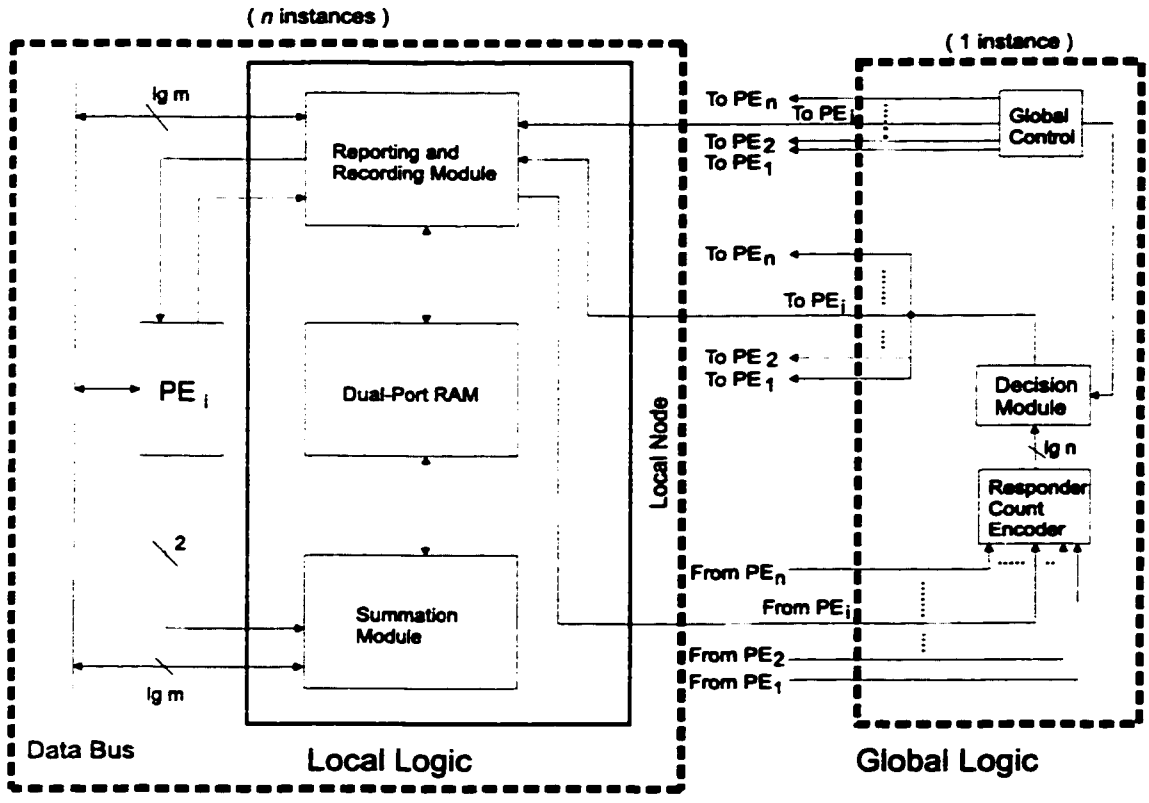


Figure 5.1: Basic Layout

barrier number into a First-In First-Out (FIFO) register or skipping to inspect the next barrier.

5.2 Operational Concept

Each PE notifies its own Local Logic whenever it produces or consumes a task. The Local Logic then makes adjustments to the local cumulative task count of the related barrier, which is stored in a dual-port RAM. Hence it is referred to as a "Distributed Summation" method. A value of 1 is added to the task count before each task is produced. while a value of 1 is deducted from the task count after each task is consumed. There are m words in the dual-port RAM to record task counts for m distinctive barriers to accommodate multithreading.

Since a dual-port RAM is adopted, the Global Logic can inspect the local task counts without interfering with the simultaneous adjustments to the local task counts by the PEs. The Global Logic broadcasts requests to each PE, demanding the local task count for a specified barrier. Each PE responds with its own local task count for the designated barrier. The Global Logic sums them up to obtain the global task count and determine if the barrier has been reached. If the global task count equals zero, indicating that the barrier has been reached, it signals each PE to record the finished barrier number in its own FIFO queue. If not zero, it signals each PE to load the task count for the next barrier to be examined. The Global Logic keeps inspecting the global task counts for consecutive barriers in a round-robin style, independent of other events. When it inspects each task count, the Global Logic sums up and examines the count bit-by-bit starting from the least significant bit. Hence, the design is referred to as a "Bit-Comparison" method. If any bit of the sum is one, which means that the task count is not zero; the task count word for the next barrier will immediately be fetched and checked. If the bit is not one, then the next bit of the current count will be fetched and checked. There are two reasons to do so:

1. The data lines between the global hardware and each local hardware can be reduced to be minimal instead of full width of the task count.

2. Considering the fact that it takes some time to process the spawned tasks, it is not necessarily inferior to the practice of comparing all bits simultaneously with getting negative results for most of the time.

Each PE decides when to examine the FIFO queue of finished barriers which enables the FIFO queue to output a value. The FIFO queue signals the PE only when it is empty, which means all barriers have been reached.

```
Procedure DSBC();

parbegin /*Summation Module*/
   adjust task count according to the request of local PEs;
   continue;
parend

parbegin /*Bit-Comparison Module*/
   case
     0≤ BCM_count1 ≤ (full_range-1): /*check the sum of all task counts bit by bit*/
            if decision == 0 /*the sum of this bit is zero*/
                  BCM_count1++;
                  fetch next bit;
            else /*the sum of this bit is not zero*/
                  reset BCM_counter1;
                  fetch next word;

     BCM_count1 == full_range: /*check the last bit of the sum of all task counts*/
            if decision == 0 /*all the bits of the sum are zeroes, the barrier is reached*/
                  reset BCM_counter1;
                  fetch next word;
                  write current barrier ID in FIFO;
            else /*the sum of the last bit is not zero*/
                  reset BCM_counter1;
                  fetch next word;
   endcase
   continue;
parend
```

Figure 5.2: DSBC Algorithm

5.3 Hardware Components

The functionality of the hardware components is specified in Figure 5.2.

5.3.1 Local Logic

Summation Module Algorithm

The operations of the Summation Module are subject to the inputs from the local PE. The four possible activities and the design of the summation module are shown in Figure 5.3. When the input is 00, no action is required. When the input is 01, which means that a task will be produced by the PE, it reads the current task count from the dual-port RAM, adds one to the count, and stores the result in the latch. When the input is 10, which means that a task is consumed by the PE, it reads the current task count from the dual-port RAM, deducts one from the count, and stores the result in the latch. The operation of deducting one is executed by adding a value of negative one as a two's complement number, for example, adding 1111 to a 4-bit number. The inputs of 01 and 10 must be followed by the input of 11, which enables writing the result in the latch back to the dual-port RAM.

Reporting and Recording Module Algorithm

The layout of the Reporting and Recording (R & R) module is schemed in Figure 5.4. The R & R module consists of two major components, namely *Parallel-In Serial Out* (PISO) register and FIFO register. The PISO register loads the task count in parallel and outputs the binary representation of the count serially while the DSBC logic is inspecting the task count. The FIFO queue is used to store the completed barrier numbers. It is driven by the result from the Global Logic.

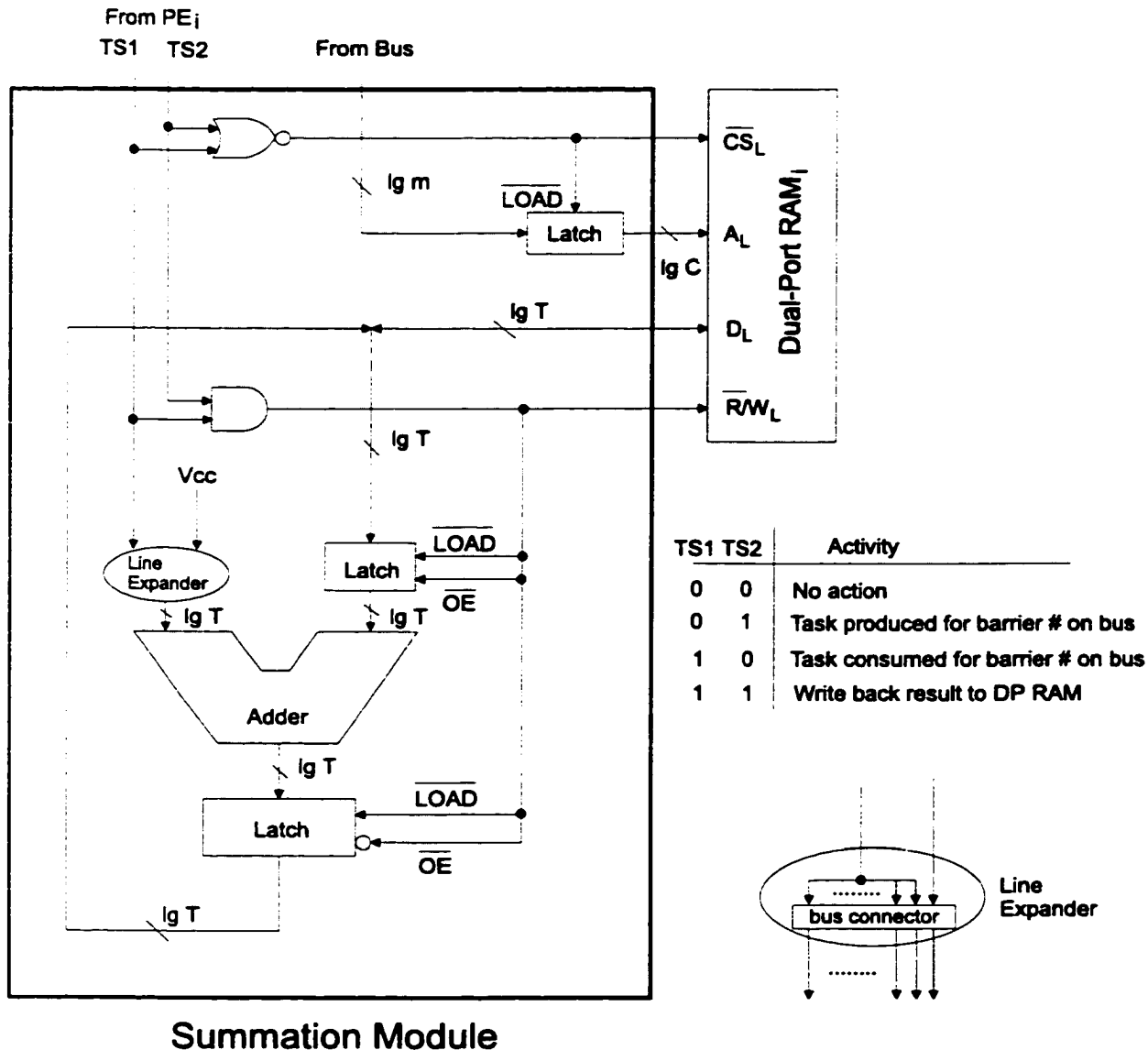


Figure 5.3: Summation Module

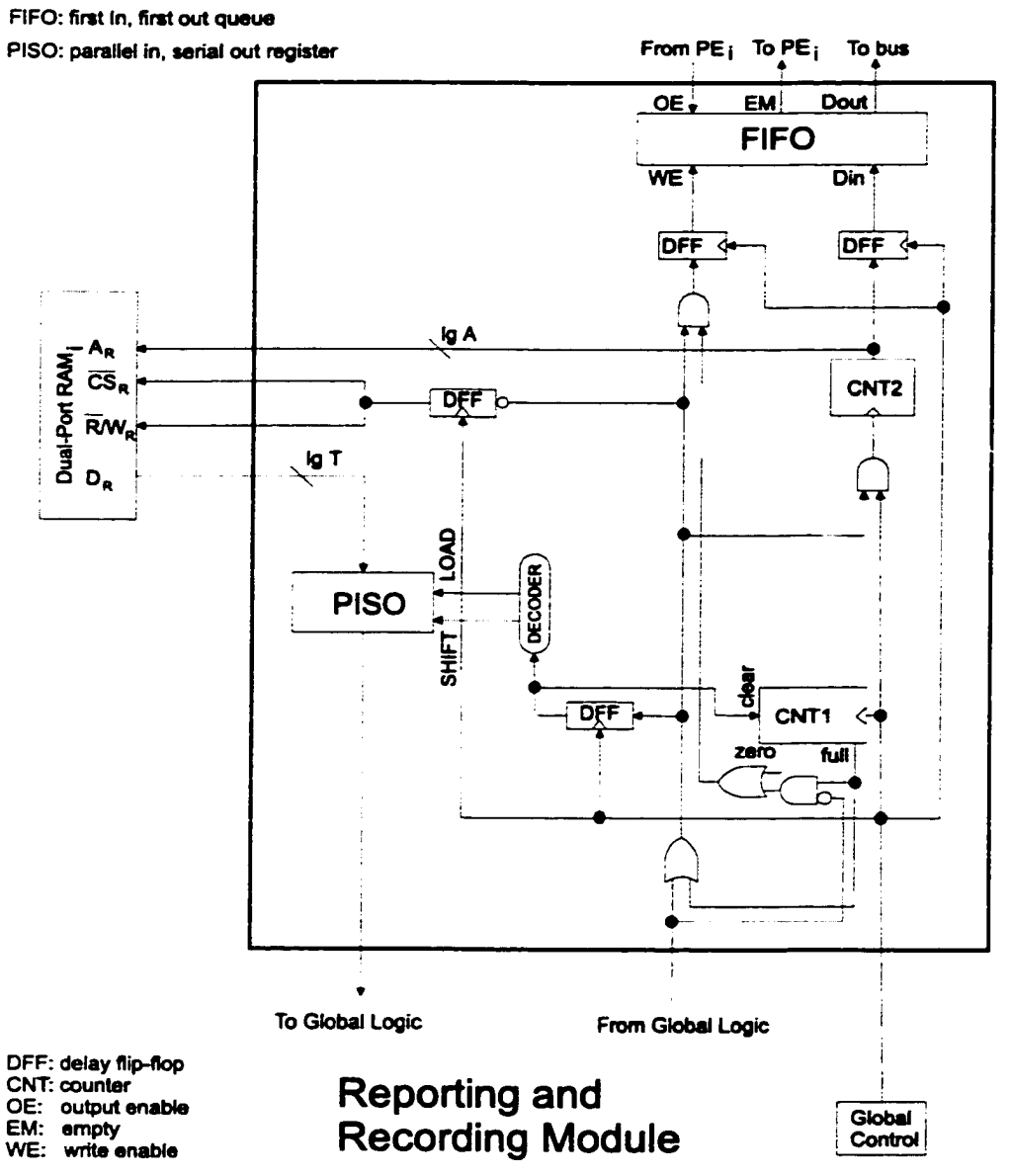


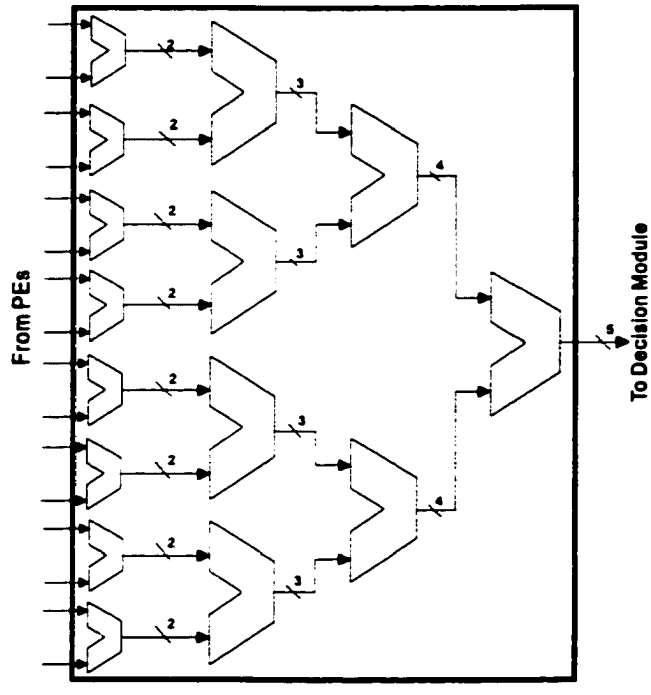
Figure 5.4: Reporting and Recording Module

The operation of the R & R module is driven and synchronized by the Global Control signal. The sequences are controlled by the counter CNT1 as shown in Figure 5.4. A second counter, labeled CNT2, maintains the barrier number to be inspected next and serves as the completed barrier number provider if the current barrier is found to be completed. When the cycle begins, it loads the task count of the barrier identified by CNT2 from the Dual-Port RAM into the PISO register. The PISO register outputs one bit at a time starting from the least significant bit to the global hardware. If the result from the decision module is zero. which means more bits need to be checked to determine termination then the next bit in the PISO register is fed to the Global Logic. If the result is non-zero then the barrier is not yet reached The Global Control will then start a new cycle with next task count word for the next barrier. If all of the resulting bits are zero for the entire word, then the sum of all the local task counts is zero which implies that the barrier has been finished. the control writes the current barrier ID into the FIFO queue and begins a new cycle for the next barrier.

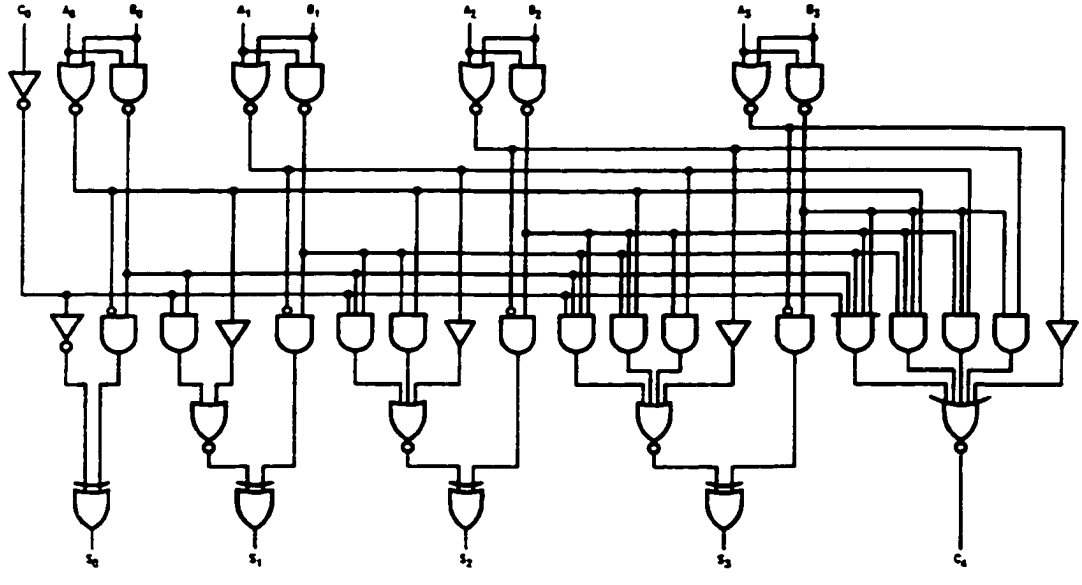
5.3.2 Global Logic

Responder Count Encoder

The Responder Count Encoder is used to sum up the single-bit indicator lines from all PEs and output the sum in a binary encoded format. An adder-tree as shown in Figure5.5(a) serves this purpose. The levels in the adder-tree depends on the number of the supported PEs. A 4-bit full adder with fast carry design, as shown in Figure5.5(b), is a direct way to extend this to a multiple-bit full adder without introducing significant gate delays. Therefore support for more PEs only slightly increases the propagation delays.



(a) Responder Count Encoder with 16 inputs



(b) TI 74F283 4-bit Full Adder with Fast Carry

Figure 5.5: Responder Count Encoder

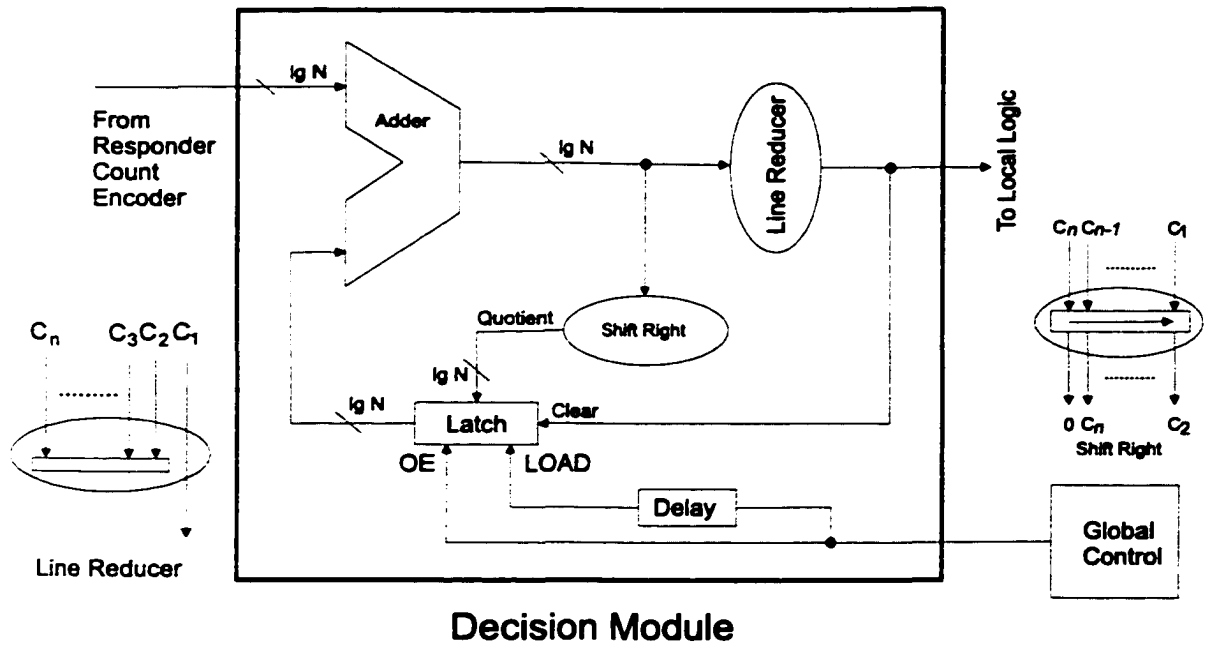


Figure 5.6: Decision Module

Decision Module

The Decision Module adds the sum from the Responder Count Encoder to the carry output from the previous bit, if any exists; then it extracts the least significant bit (LSB) to indicate the decision by using the Line Reducer. The reason for adopting the LSB to indicate the decision is that if the sum is an odd number as its LSB value is 1. Obviously the barrier is not reached under this scenario. The sum's LSB is zero if the count is either zero or an even number. Under this circumstance, we need to check further to the next bit to decide whether the barrier has been achieved. At the same time the Decision Module also directs the result from the adder to the *Shift Right* Function, which can be implemented by just relabeling each bit as the next less significant bit. It stores this quotient into the latch to be used as the carry for the next bit, however the carry will be cleared if the decision is one since the carry does not apply to the next barrier.

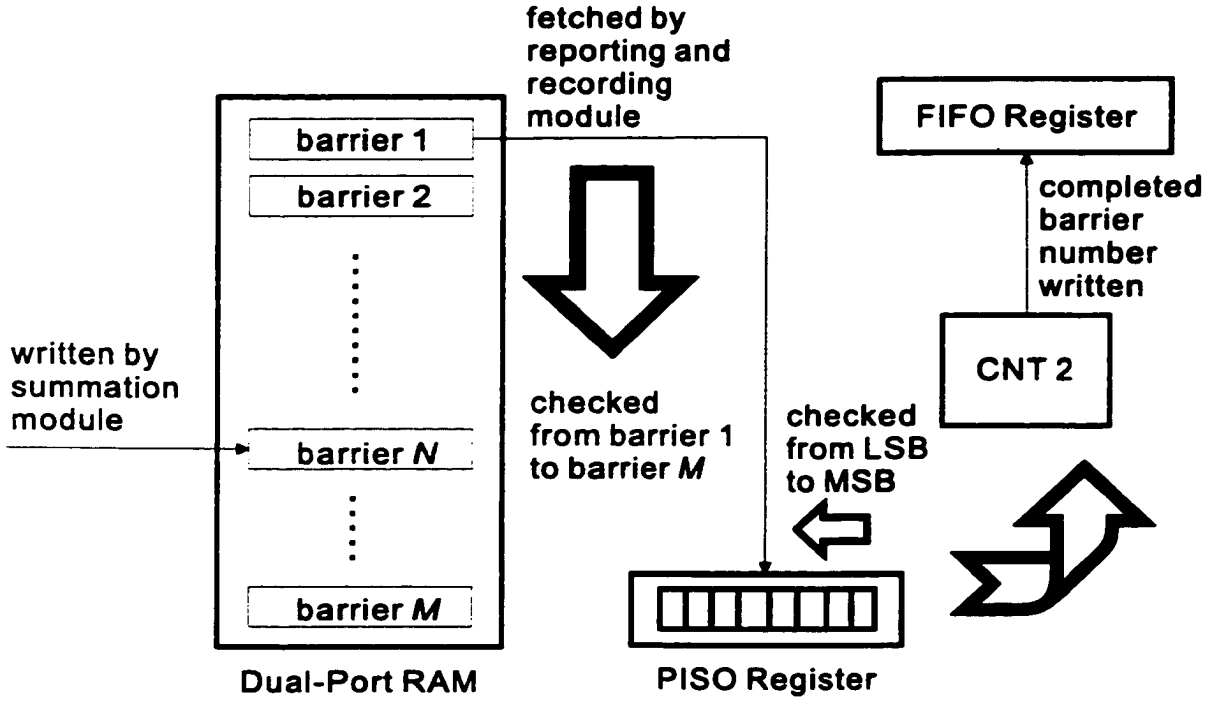


Figure 5.7: Procedure Applied by DSBC Logic to Detect Completed Barrier

5.4 Performance Analysis

In this section, we first analyze the time for termination detection with the DSBC logic, denoted T_{DSBC}. Then its performance is compared with a software-based Test-and-Set algorithm [34] and the Wired-NOR logic [26] in both performance and features.

5.4.1 Detection Time

The procedures for a completed barrier to be detected by the DSBC logic can be decomposed as shown in Figure 5.7. First, the local task counts for the current barrier are written into the dual-port RAM after the barrier has been reached. Then it must wait its turn to be checked, requiring time T_{wait}. Once it occurs, the barrier can be determined to be completed serially in time T_{word}. In particular, all bits of

the word instead of just partial bits are checked, and stored in the latch in front of the FIFO queue. The barrier number is immediately written into the FIFO register after the next cycle begins requiring time T_{FIFO}. Thus, this defines the DSBC logic timing as given by Eq.(5.1).

$$\begin{aligned} T_{DSBC} &= T_{RAM} + T_{wait} + T_{word} + T_{FIFO} \\ &= T_{RAM} + T_{wait} + q \cdot T_{bt\_chk} + T_{FIFO} \end{aligned} \tag{5.1}$$

where $q = t + w$, and 2^t is the maximum number of task counts supported by each PE and w denotes the additional bits generated by the carry operations.

For physically distributed computing system, we can estimate DSBC performance using discrete ICs and their datasheets collected from several semiconductor makers. First, the writing time to the dual-port RAM, T_{RAM}, is about 30 ns, since 2 cycles are required at 10 ns each to read the current count and then write back the incremented count, and 10 ns to perform the addition. The writing time to the FIFO queue, T_{FIFO}, is 12ns. The waiting time, T_{wait}, is variable and will be analyzed below. The time needed for one barrier-checkup cycle, T_{word}, ranges from 1 to q bit-checkup times, T_{bt_chk}, depending on the status of these bits. For a completed barrier, all bits of the global task counts are zero; hence every bits will be checked to decide whether the barrier is reached. Thus, $T_{word} = q \cdot T_{bt\_chk}$ as given in Eq. 5.1. The bit-checkup time can be estimated by summing up the propagation delays of the gates along the major datapath of the DSBC logic. However the depth of the adder tree in the Responder Count Encoder increases as the number of supported PEs grows. Therefore a new notation $T^n_{bt\_chk}$, where n is the number of PEs in the system, is introduced to identify

the different time delays. Using 7400-series components, the bit-checkup cycle time for DSBC logic supporting 256 PEs, $T^{256}_{bt\_chk}$, is 190 ns or less.

As the DSBC logic relies upon an instantaneous snapshot of the system state, the maximum propagation delays dictate 190 ns cycle time to ensure data integrity. Because synchronization behaviour of parallel applications can vary widely, a probability-based estimate of the typical number of bit-checkup cycles in a barrier cycle provides a fair and equiprobable estimation. The calculation using of the arithmetic-geometric series is given in Eq. 5.2.

$$\overline{T^n_{word}} = \{\frac{1}{2}\cdot 1 + \frac{1}{2^2}\cdot 2 + \cdots + \frac{1}{2^{q-1}}\cdot(q-1) + \frac{1}{2^{q-1}}\cdot q\}\cdot T^n_{bt\_chk} \tag{5.2}$$

The first parameter in each term except the last term is the probability to finally get a value of 1 from bits to be checked after having i values of zero in a row. The second parameter is the number of checkup cycles to be accounted for, which ranges from 1 to q. Take the second term for example, the probability of having the value of one after having one value of zero is $\frac{1}{2}\cdot\frac{1}{2} = (\frac{1}{2})^2$. That situation results two bit-checkup cycles because the first value of zero makes checking the next bit necessary; however the fact that the second value is one clears the need to check next bit. The probability for the last term includes those of both value zero and one because t bits are checked regardless of if the value is zero or one. The rearrangement and summation of the series in Eq. 5.2 allows it to be expressed as Eq. 5.3.

$$\overline{T^n_{word}} = \{2 - (\frac{1}{2})^{q-2}\cdot q + (\frac{1}{2})^{q-1}\cdot(2q-1)\}\cdot T^n_{bt\_chk} \tag{5.3}$$

Assume that in the dual-port RAM m words are utilized, implying support for m barriers. The best case happens when the current barrier completes while the previous barrier is being checked. The next barrier-checkup cycle will detect the termination

without waiting, hence $T_{wait} = 0$. The worst case occurs when the barrier is completed while the current barrier is being checked. The termination of current barrier cannot be detected immediately because the previous task count snapshot is being checked, therefore it has to wait for $(m-1)$ barrier-checkup cycles before it can be checked again. Likewise, an equiprobable analysis provides a fair estimation of the typical wait time. The previous evaluation of the typical number of the bit-checkup cycles in a barrier-checkup cycle, $\overline{T^n_{word}}$, should serve the purpose well also. Hence the calculation can be performed as:

$$\begin{aligned}
\overline{T^n_{wait}} &= \{\frac{1}{m}\cdot 0 + \frac{1}{m}\cdot 1 + \cdots + \frac{1}{m}\cdot (m-1)\}\cdot \overline{T^n_{word}} \\
&= \frac{(m-1)}{2}\cdot \overline{T^n_{word}} \\
&= (m-1)\{1-(\frac{1}{2})^{q-1}\cdot q + (\frac{1}{2})^q\cdot(2q-1)\}\cdot T^n_{bt_chk} \qquad (5.4)
\end{aligned}$$

The expected value for barrier detection time using DSBC logic can be obtained by substituting Eq. 5.4 into Eq. 5.1. The derivation can found in any mathematical handbook of formulas, such as [7].

$$\begin{aligned}
\overline{T^n_{DSBC}} &= T_{RAM} + \overline{T^n_{wait}} + T^n_{word} + T_{FIFO} \\
&= T_{RAM} + \{(m-1)[1-(\frac{1}{2})^{q-1}\cdot q + (\frac{1}{2})^q\cdot(2q-1)] + q\}\cdot T^n_{bt_chk} \\
&\quad + T_{FIFO} \qquad (5.5)
\end{aligned}$$

Thus, $\overline{T^n_{DSBC}}$ scales linearly with respect to m and is independent or at most only weakly dependent on $\lg n$.

5.4.2 Comparisons of Performance and Features

Based on Eq. 5.5 and information from [34] [26], the detection times for four approaches are plotted in Figure 5.8. Curves are shown for DSBC logic supporting 16 barriers, DSBC logic supporting 32 barriers, Wired-NOR logic replicated for an arbitrary number of barriers, and a Test-and-Set software-based scheme [45]. The three hardware-based schemes outperform the software-based scheme in termination detection, as the number of PEs increases. The ratio of benefit increases superlinearly from 10-fold for 20 PEs to 1000-fold for 512 PEs. Both Wired-NOR logic and DSBC logic have nearly constant detection times in the respective ranges of the number of PEs. Their detection times increase slightly above some specific PE numbers because new levels of propagation delay are added. In particular, to accommodate more PEs, additional levels of adders are required for the DSBC logic, and a new repeater board is required for the Wired-NOR logic configuration. Theoretically, the detection time of the Wired-NOR logic is independent of the number of supported barriers, m; however it is restricted to 16 with current technology [26]. Although the detection time of the DSBC logic increases as m increases, the version of the DSBC logic supporting 16 barriers takes less detection time than the Wired-NOR logic while the version of the DSBC logic supporting 32 barriers needs slightly more time than the Wired-NOR logic. The problem can also be easily remedied by duplicating the DSBC logic. Since the DSBC logic works independently of the local PEs, more sets of them can be implemented and execute in parallel without degrading the performance of the local PEs. The only drawback is the line complexity is increased. However each DSBC logic requires only $3n$ lines, where n is the number of PE of the system, between its local hardware and global hardware while the Wired-NOR logic requires $m \cdot n$ lines. Therefore the DSBC logic still can perform better with less line complexity.

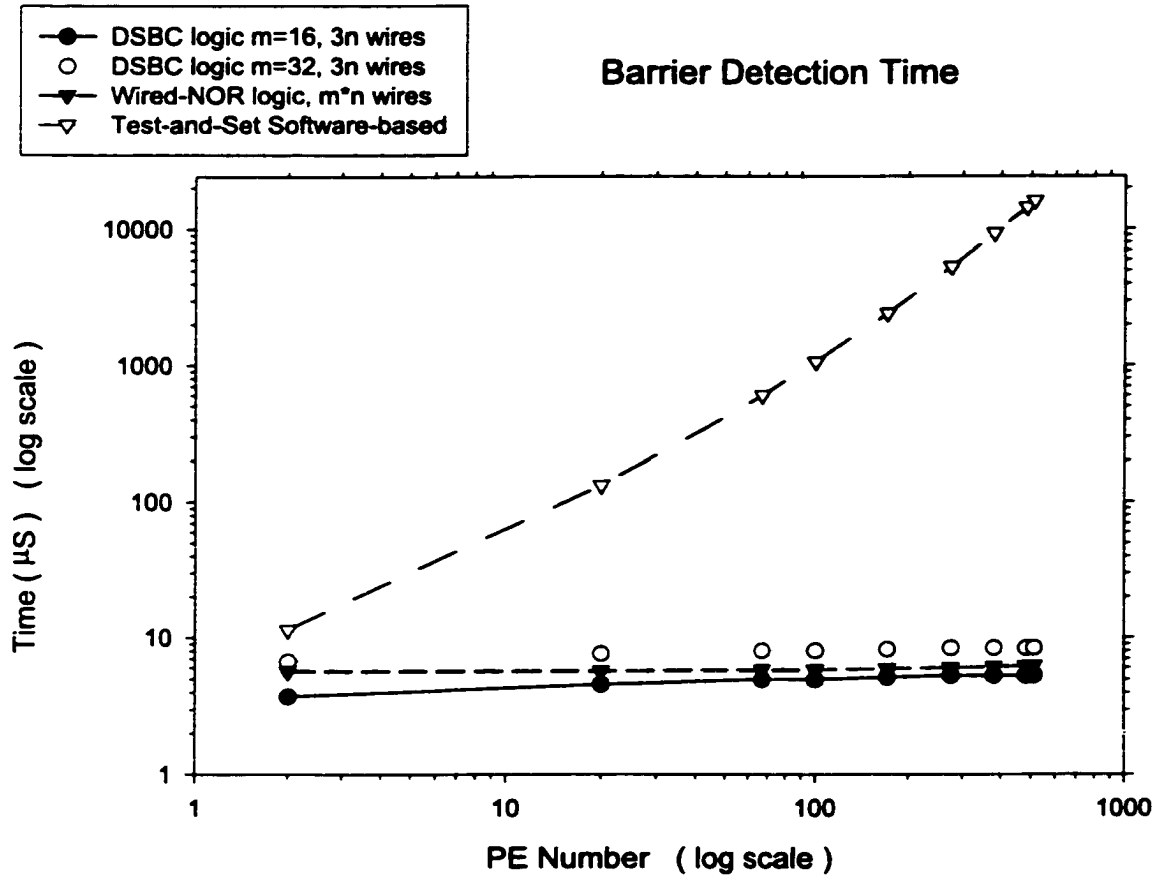


Figure 5.8: Detection Time Comparison

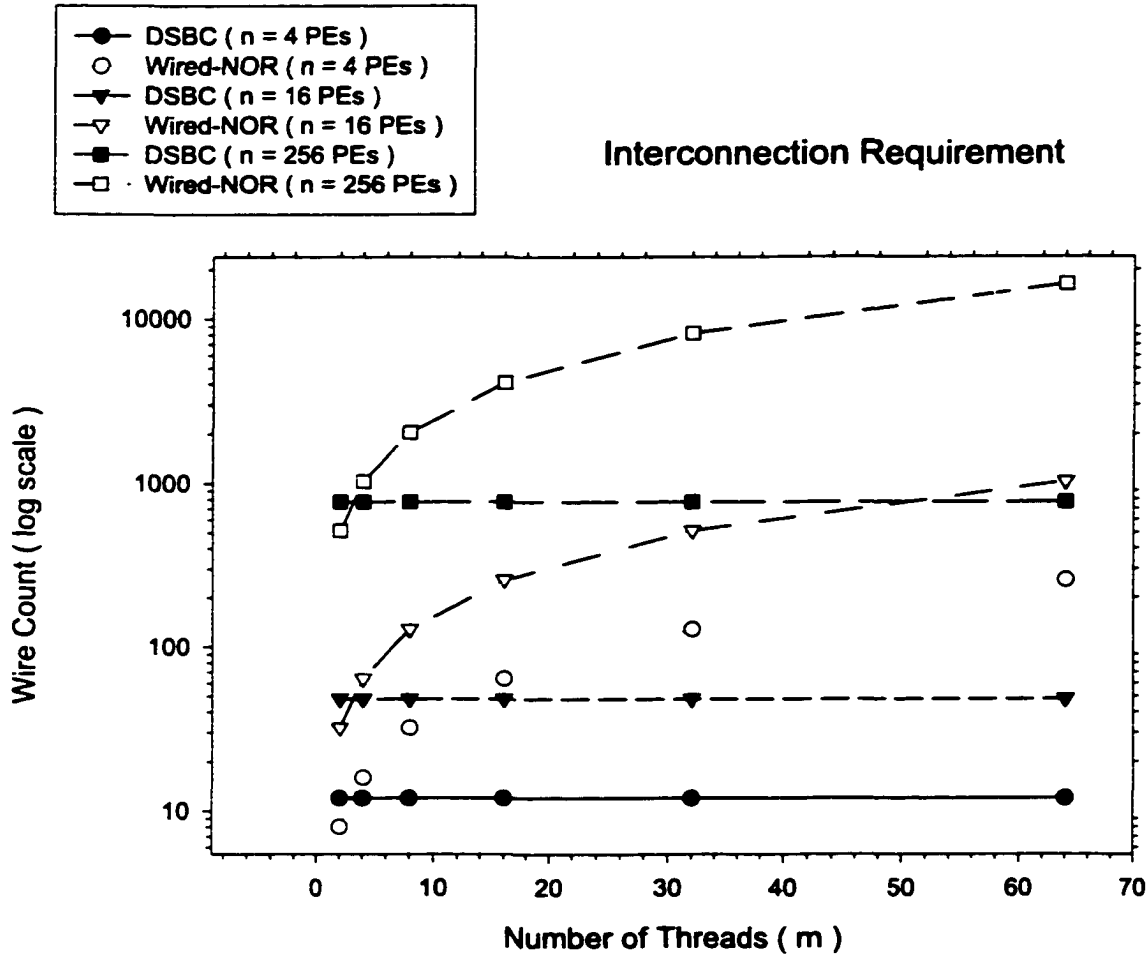


Figure 5.9: Interconnection Requirement Comparison

There are additional features of the DSBC logic which are not revealed by the detection time comparison. The Wired-NOR logic supports only one task dispatched to each PE; that is effective for computations that are statically-scheduled and allocated at compile time. The DSBC logic can support execution of multiple tasks dispatched to each PE that contribute to the same barrier dynamically at run time; that makes it suitable for all kinds of applications. DSBC also provides adaptability, since the only interaction between the local PE and the DSBC logic is the writing into the dual-port RAM and reading the FIFO register of the Local Hardware of the DSBC logic, which is distributed to that node. That fact makes both applications on message-passing architectures [49] [44] and applications on shared-memory architectures [49] [44] easy to adapt to the DSBC logic methods. Actually, DSBC logic blends aspects of both synchronization approaches, message-passing and shared-memory. In the DSBC logic approach, the PE places an encoded message in the dual-port RAM to indicate producing or consuming of a task; the action is similar to that of the message-passing architectures. However, the fact that the completed barrier number is stored in the local FIFO register and waits to be inspected acts somewhat like the behaviour of a shared-memory approach. Both the operating system and the compiler can readily adopt the DSBC logic because multithreaded synchronization can be implemented by simply accessing specific memory locations.

5.5 Delay-Insensitive Design

Delay-insensitive circuits [35] can eliminate time dependencies in digital logic circuits because of its clockless design. All concerns about timing, for example clock skew, can be cleared since they are completely insensitive to the propagation delays among its gates. That fact is also potential to improve the performance of their Boolean logic counterparts if properly designed. *Null Convention Logic* (NCL) [35]

is the first technique able to economically implement the delay-insensitive circuits. In the following sections, NCL is introduced and a delay-insensitive DSBC logic is developed utilizing NCL components.

5.5.1 Null Convention Logic

Traditional Boolean logic is not symbolically complete because it needs the assistance of other control components [33], such as clocks, to coordinate the gates in order to get valid results. NCL implements different approaches to accomplish a symbolically complete logic without clocks. NCL can utilize dual-rail encoding for each Boolean variable instead of low and high voltages on a single rail as in the traditional Boolean logic. The high voltage in a wire represents the validity of the data or DATA while the low voltage means invalidity of data or NULL. Thus the arrival of the data wavefront can be clearly identified by existence of DATA or NULL. Since each wire in NCL can only express the validity or invalidity of a data value, unlike that each wire in traditional Boolean logic can express two values, namely 0 and 1; threshold gates [36] are utilized to sense how many DATA values are present. A threshold gate will assert DATA when sufficient or more DATA values are input. A 5 input/threshold 3 gate is shown in Figure 5.10. Threshold gates with hysteresis are required to synchronize the DATA or NULL wavefronts. A threshold gate with hysteresis is actually a threshold gate with weighted feedback of (threshold-1) as shown in Figure 5.11. It acts somewhat similarly to a latch in a clocked logic. It asserts DATA only after required DATA values arrive and DATA keeps being asserted until all input DATA values transform into NULL because of the weighted feedback. Thus, the DATA or NULL wavefronts can be synchronized. The NCL asynchronous register as shown in Figure 5.12 provides means to control the flow of the DATA or NULL wavefronts. A combinational network can be realized with NCL pipelines as shown in Figure

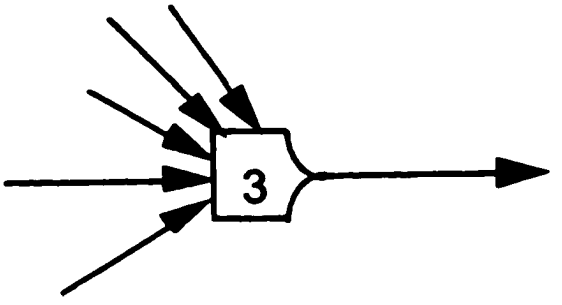


Figure 5.10: 5 Input/Threshold 3 gate [35]

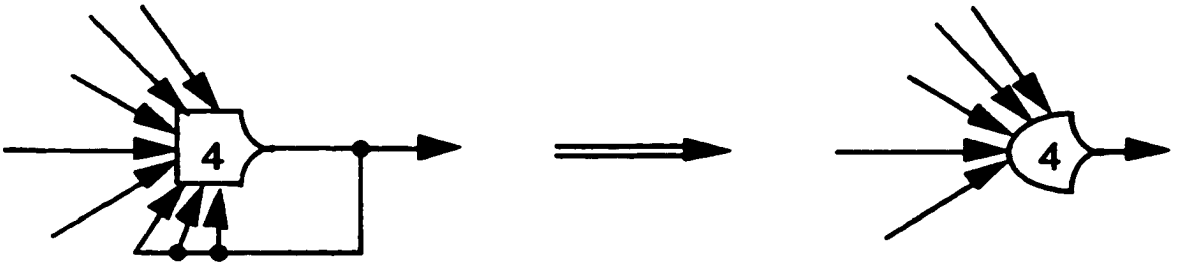


Figure 5.11: Threshold Gate with Weighted Feedback of (Threshold-1) [35]

5.13 while a sequential network can be implemented with NCL with the arrangement shown in Figure 5.14. With those building blocks, virtually any device can be built with NCL and functions as its Boolean logic counterpart, only without help of the clock.

5.5.2 NCL Version DSBC Logic

The DSBC Logic adopting NCL components is designed here. The basic layout is sketched in Figure 5.15. For simplicity, full ranks of wires, i.e. DATA 0 and DATA 1, are shown as one wire in all NCL diagrams. Since the summation module plays no important role in the performance issue, only the Reporting and Recording Module,

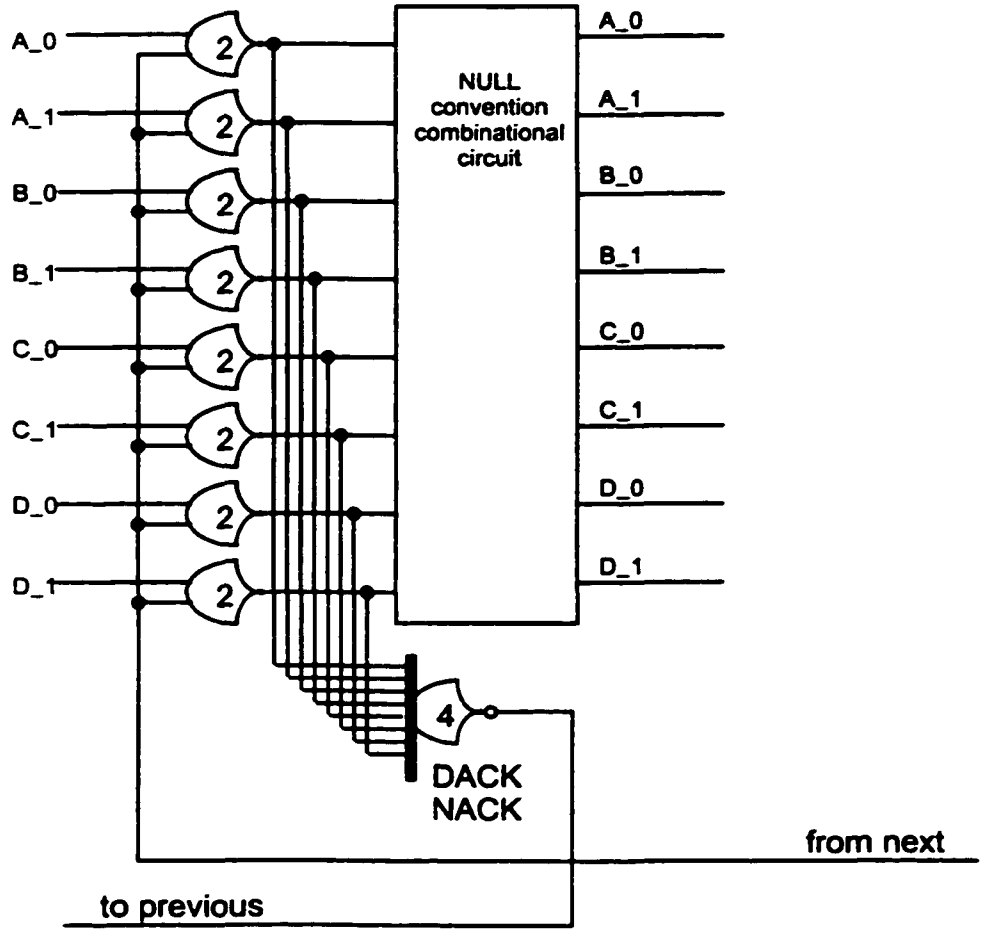


Figure 5.12: Null Convention Logic Register [35]

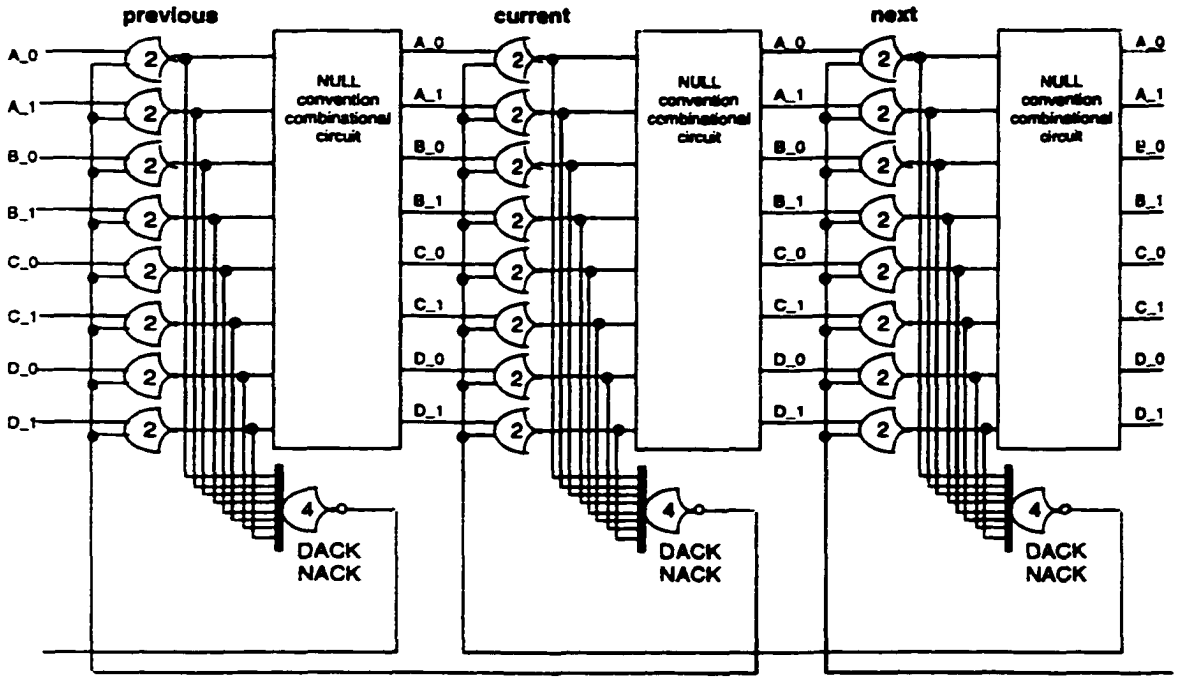


Figure 5.13: NCL Combinational Network [35]

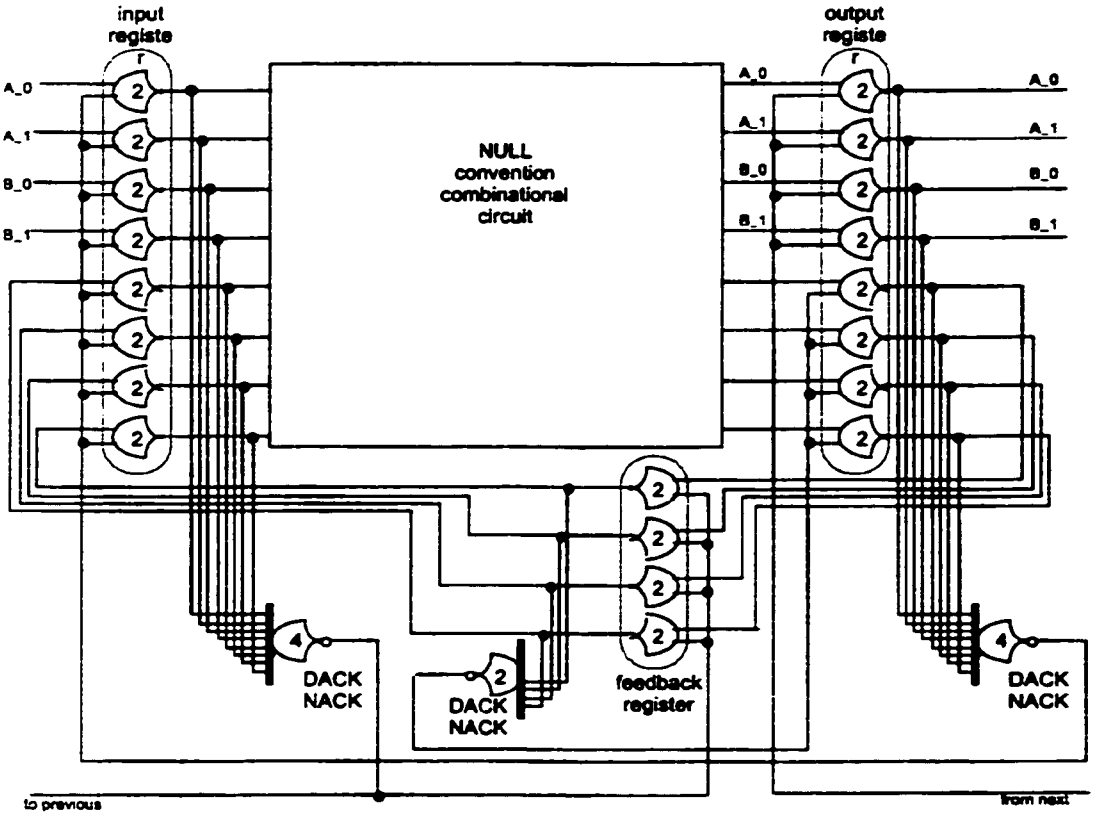


Figure 5.14: NCL Sequential Network [35]

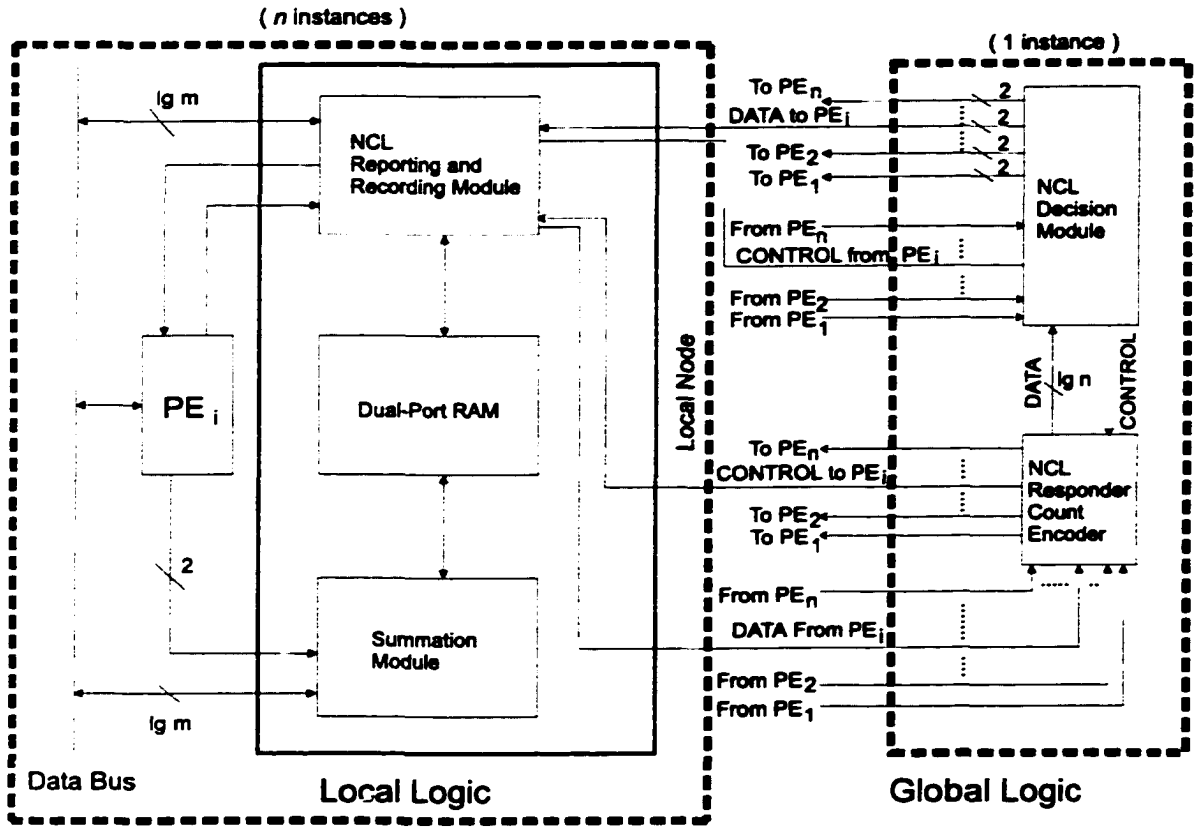


Figure 5.15: NCL Version DSBC Logic Basic Layout

Responder Count Encoder, and Decision Module are implemented with NCL. Also for sake of simplicity, some logic and gates still adopt traditional Boolean logic symbols while they are fully implementable with NCL.

NCL Reporting and Recording Module Encoder

The NCL Version Reporting and Recording Module layout is identical to the original Reporting and Recording Module except NCL registers are inserted among gates to mediate the flow of DATA and NULL wavefronts. Since the dual-port RAM is not a NCL, the interface between it and the NCL R & R Module needs converters to transform DATA values. A delay device is also required in parallel with the Dual-

port RAM to ensure that the output data of it is synchronized with the DATA front. The NCL Version Reporting and Recording Module is shown in Figure 5.16.

NCL Version Responder Count Encoder

The layout of NCL Version Responder Count Encoder is also identical to the original Responder Count Encoder except one NCL register is inserted to mediate the flow of DATA and NULL wavefronts from Reporting and Recording Module and to the Decision Module. All adders in the Responder Count Encoder are implemented with NCL version gates. The NCL Version Responder Count Encoder is shown in Figure 5.17.

NCL Version Decision Module

The NCL Version Decision Module is shown in Figure 5.18. It is also identical to the original Decision Module except Always-1 logic is added to generate the required DATA to actuate Counter 1 in the NCL Reporting and Recording Module because there is no clock in the NCL version DSBC Logic. The latch in the original design can be replaced by a NCL register with the NCL sequential network arrangement of control wires and a Reset logic to perform the original clear function.

5.6 Summary

In this chapter, we have introduced hardware support for termination detection capable of supporting multithreading. It supports dynamic allocation of multiple barriers and multiple tasks per barrier while remaining scalable in time and space complexity. Through theoretical analysis and calculations, DSBC is shown to outperform existing termination detection hardware while providing additional capability. The speedup of the DSBC logic supporting 16 barriers over a Test-and-Set software-based

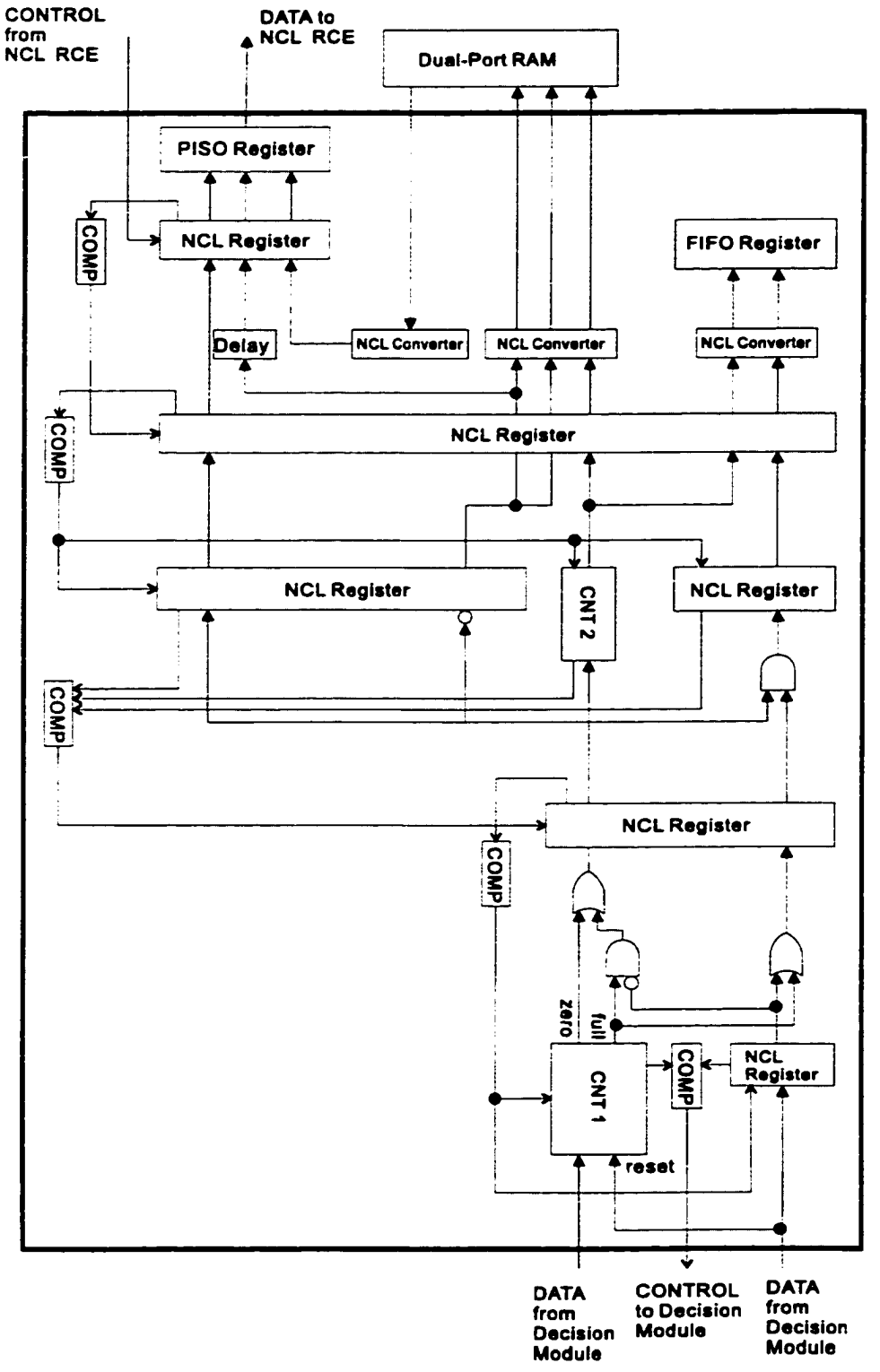


Figure 5.16: NCL Version Reporting and Recording Module

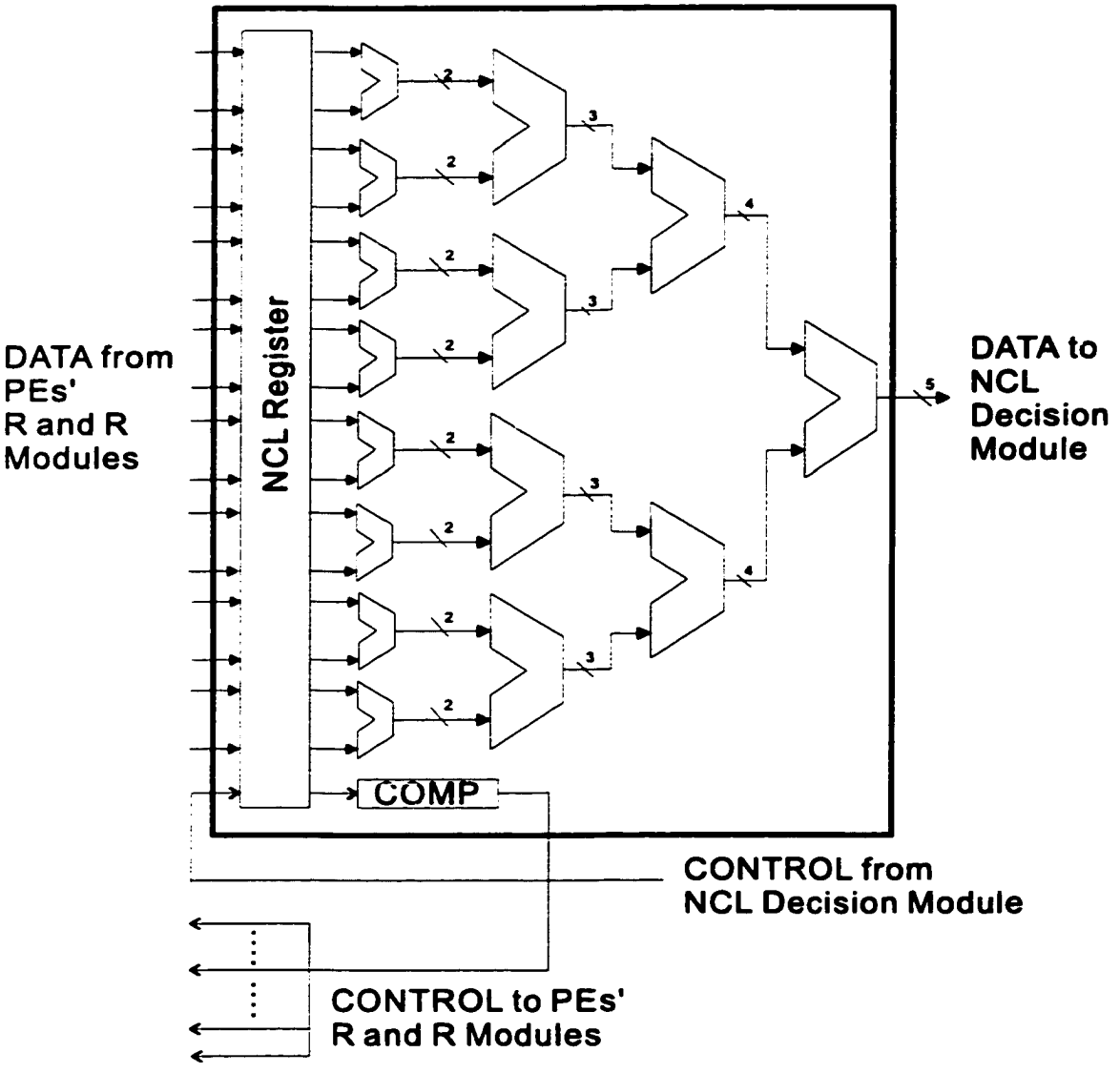


Figure 5.17: NCL Version Responder Count Encoder

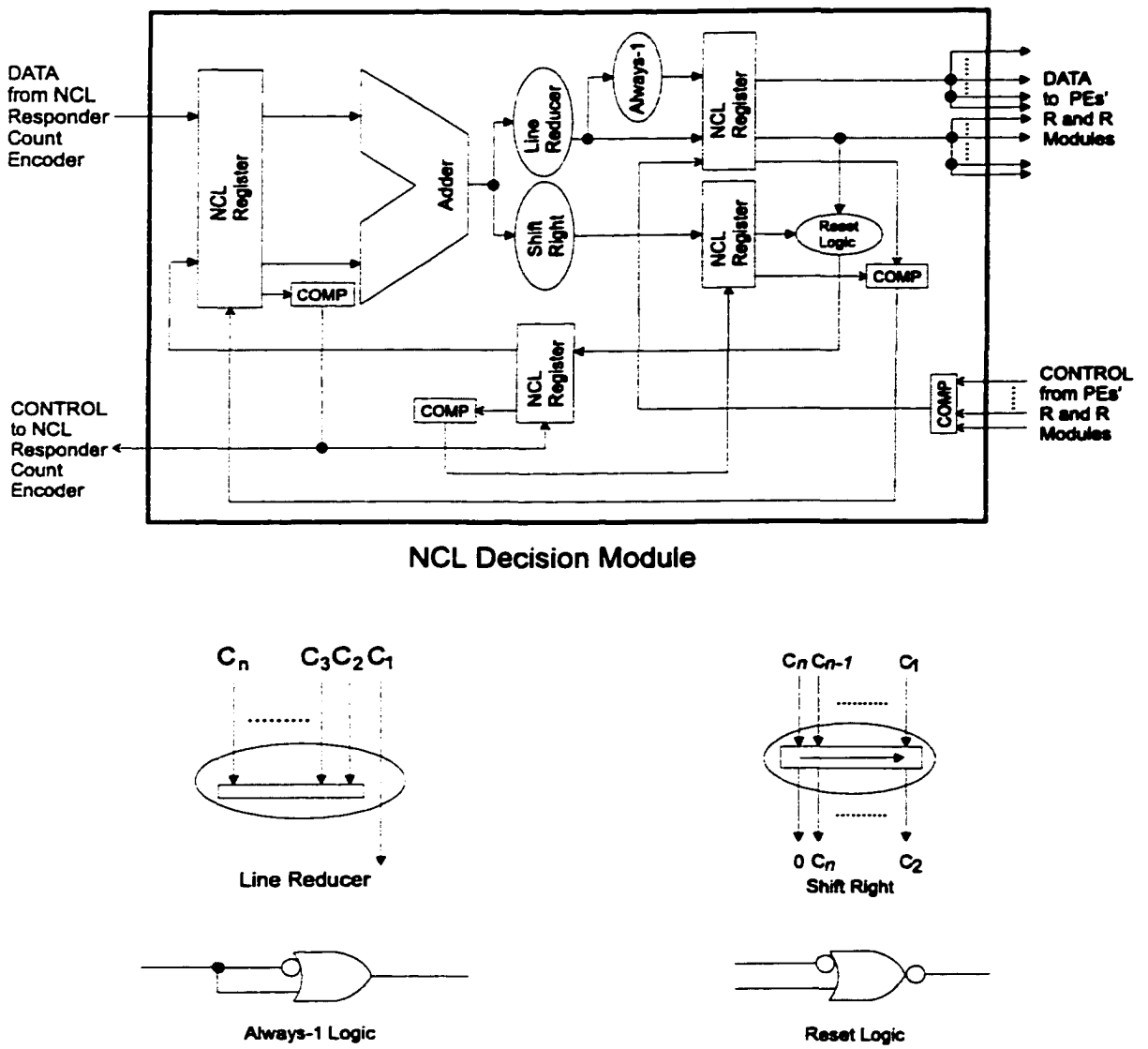


Figure 5.18: NCL Version Decision Module

scheme is 29-fold for a 20-processor system to 3,008-fold for a 512-processor system. With consideration of the number of barriers supported and the number of tasks allocated to each PE, DSBC logic can be advantageously applied. Because of the low line complexity, even the DSBC logic configuration provided here can be duplicated to boost overall performance of the parallel application being executed by inspecting more barriers at the same time. The clockless logic version[?] intended to cut the propagation delays with the NULL Convention Logic has also been designed to give another practical alternative where interconnection delays are significant.

The software interface to the DSBC logic consists of writing to the dual-port RAM and reading the FIFO register. That relieves the compiler and the programmer of all activities except for accessing certain memories. The fact that the task count is distributed in the memories of each PE and each PE needs to check its local FIFO register resembles a distributed shared-memory lock access. Meanwhile, the practice of summing local task counts and returning the identified completed barrier number back to each node resembles message-passing synchronization protocols. Those characteristics encourage adaptability of both the message-passing and shared-memory applications to adopting DSBC logic.

CHAPTER 6

CONCLUSION

6.1 Summary

The performance of termination detection is fundamental to the throughput of the parallel and distributed applications. An optimal termination detection algorithm should be sought to keep the detection overhead minimized in order to reduce the impact to the underlying computation. In this dissertation, we proposed a capability taxonomy of the termination detection techniques in parallel and distributed computation. The classification is based on the characteristics of process allocation and degree of processor reactivation support. There are eight classes in the taxonomy, namely SBIT, SBST, SBDT, SBAT, DBIT, DBST, DBDT, and DBAT. A capability class hierarchy is formed as a result of the taxonomy. The classification along with hierarchy facilitate the recognition of the capability class of any existing termination detection algorithm. Any advantage or limitation of it can be easily identified, which will provide valuable experience in the design course of a new algorithm.

Thirteen popular termination detection algorithms in literature were then introduced and carefully studied. Their strong points were identified while their limitations were analyzed. The knowledge contributes valuable guidelines for what to include and what to avoid during the implementation of a termination detection technique. An

optimality analysis was performed to find the optimum which serves as the ultimate design goal of termination detection techniques. Traditionally the quest for an optimal algorithm is focused on the message complexity concerned with the high overhead of message transmission in communication channels. A lower bound of T messages for T tasks in a barrier was established [11]. However the difference of the overhead for delivering external messages and internal messages is ignored. Therefore, we started the optimality research by identifying the difference of delivery overhead between the internal notifications and external messages. The lower bound of T appears to make network traffic unnecessarily large under our investigation. That fact inevitably will effect the underlying computation. We presented new lower bounds for static-binding and dynamic-binding termination detection techniques after making tradeoff and examining relevant performance deviations. The new lower bound for a static-binding algorithm is $\min(T, N)$. The new lower bound for a dynamic-binding algorithm is $\min(T, E)$.

With the guidelines and specific goal collected from the previous study, we refined the Tiered Detection Algorithm, which is a software-based approach supporting dynamic allocation of multithreaded processes and classified as DBAT class. The key point that was learned from other algorithms and applied on the Tiered Detection Algorithm is the implementation of a global invariant. This invariant, equal production count and consumption count in each level of the task dispatching tree, allows our design to detect the barrier without explicitly obtaining the status of all processing elements, and to detect spawn messages in transit in communication channels without reading the status of the network. This attribute combined with the processor-centered signaling approach and the manner of reporting only after the local node becomes idle significantly reduce the number of external messages required. The performance of Tiered Detection Algorithm is compared with those of three other more

efficient algorithms, namely Credit Algorithm, CV Algorithm, and LTD Algorithm. In message complexity aspect, although Credit Algorithm requires the least messages, they are all external messages which will cause heavy network traffic and make overall performance suffer. The Tiered Algorithm needs only E external messages plus T internal messages just as anticipated. The other two algorithms demand more than E external messages in addition to T internal notifications. In bit complexity, the requirement for each algorithm varies with each individual case; none has significant advantage over others and all bit requirements are around the order of $T \lg T$. As the bandwidth of most networks is more than sufficient for these algorithms, there will not be noteworthy difference. In the network protocol which adopts the fixed-length packet, it is hardly an issue since the capacity of the packets are much more than each message requests. Tiered Detection Algorithms excels in detection latency with only one step necessary. Credit Algorithm requires one step theoretically while practical overhead could be large. The other two algorithms both have to transmit the status report through the hierarchy of their logical trees of processing elements. The space complexity is actually a cost factor rather than performance factor. Recognizing the fact that the space requirement of Tiered Detection Algorithm is merely several KBytes although larger than those of the other three algorithms, that introduces no concern either. The fact that Tiered Detection Algorithm outperforms others in critical fields while making tradeoff in negligible aspects makes it a promising practical choice.

The global invariant implemented in Tiered Detection Algorithm was extended and utilized to develop another hardware-based approach, the Distributed-Sum Bit-Comparison (DSBC) Logic. It supports dynamic allocation of multithreaded processes on shared-memory [44], message-passing [44], and/or single-chip multiprocessors. The invariant property employed in DSBC Logic is that the instantaneous task

consumption count equals the instantaneous task production count upon barrier completion. The independently working DSBC Logic cyclically collects and examines the task counts stored by the local nodes in the dual-port RAMs for each barrier without interfering their execution. The performance of DSBC Logic through theoretical analysis and calculations is compared with that of Wired-NOR Logic, a hardware-based approach impressive for its low detection latency, and Test-and-Set scheme, a software-based approach. The performance analysis reveals that the hardware-based schemes outperform software-based scheme superlinearly as the number of PE increases. The ratio of benefit ranges from about 10-fold for 20 PEs to about 1000-fold for 512 PEs. The DSBC Logic supporting 16 barriers takes less detection time than Wired-NOR Logic while the DSBC Logic supporting 32 barriers requires slightly more time than Wired-NOR Logic. However, Wired-NOR Logic is limited to support 16 barriers with current technology because of its high wire complexity of $B \cdot N$. The DSBC Logic owns larger freedom of supporting more barriers with its modest wire requirement of $3N$. It can even be duplicated to multiply the throughput because of its low wire complexity. A new version of DSBC Logic adopting Null Convention Logic, a symbolically complete logic, was designed to supply a delay-insensitive alternative which eliminates all timing concerns and can potentially improve the performance if properly designed.

6.2 Future Work

The execution of both the Tiered Detection Algorithm and DSBC Logic depends on reliable message delivery and error-free PE execution. Thus, the termination detection mechanism may malfunction if any node fails to function normally. Therefore, fault-tolerance capability [46] [51] [17] [48] would be beneficial if integrated into both designs. The improvements can be applied on the Tiered Detection Algorithm

through encoding mechanisms to save reporting transmission bits and information storage requirements on the controller and local nodes. As for DSBC Logic, although a NCL version has been developed, the NCL implementation can be optimized in both layout and NCL components used. A critical phenomenon observed during the development of this dissertation is the need for globally-accepted benchmarks for evaluating termination detection algorithms. In particular, a parallel and distributed application benchmark would assist significantly in validating evaluation of termination detection algorithms in practice and in relevant implementation optimizations.

LIST OF REFERENCES

[1] A.S. Tanenbaum, "Computer networks," 3rd Edition, Prentice-Hall, 1996

[2] B. Furht et al., "Handbook of internet and multimedia, systems and appications," IEEE press, 1999

[3] D.E. Comer, "Computer networks and internets," 2nd Edition, Prentice-Hall, 1999

[4] D. Sima, T. Foutain, and P. Kacsuk, "Advanced computer architectures, a design space approach," Addison-Wesley, 1997

[5] T.H. Cormen, C.E. Leiserson, and, R.L. Rivest, "Introduction to algorithms," MIT Press, 1991

[6] J.R. Smith, "The design and analysis of parallel algorithms," Oxford University Press, 1993

[7] M.R. Spiegel, "Mathematical handbook of formulas and tables," McGraw-Hill, 1968

[8] R.M. Hord, "Understanding parallel supercomputing," IEEE Press, 1999

[9] F. Mattern, "Global quiescence detection based on credit distribution and discovery," *Information Processing Letters*, Vol.30, No.4, pp.195-200, Feb., 1989

[10] S. Chandrasekaran and S. Venkatesan, "A message-optimal algorithm for distributed termination detection." *Journal of Parallel and Distributed Computing*, Vol.8, No.3, pp.245-252, March, 1990.

[11] K.M. Chandy and J. Misra, "How process learn." *Distributed Computing*, Vol.1, pp40-52, 1986.

[12] T.-H. Lai, Y.-C. Tseng, X. Dong, "A more effient message-optimal algorithm for distributed termination detection," Technical Research Report, OSU-CISRC-1/92-TR-2, Ohio State University, 1992.

[13] T.-H. Lai, Y.-C. Tseng, X. Dong, "A more effient message-optimal algorithm for distributed termination detection," *Proceedings of Sixth International Parallel Processing Symposium*, pp. 646-649, IEEE CS Press, March 1992.)

[14] S. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, C-30(3):pp. 207-214, 1981.

[15] I. Foster, "Designing and Building Parallel Programs – Concepts and Tools for Parallel Software Engineering," Addison-Wesley, 1995

[16] K. Drake, "Time and Space Efficient Multiprocessor Synchronization and Quiescence Detection," M.S. Thesis, University of Central Florida, 1995

[17] A.Y.H. Zomaya et al; *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996

[18] N.S. Arenstorf and H.F. Jordan, "Comparing Barrier Algorithms," *Parallel Computing* 12, pp.157-170, 1989

[19] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984

[20] E.D. Brooks III, "The Butterfly Barrier," *International Journal of Parallel Programming*, Vol.15 No.14, pp.295-307, 1986

[21] D. Hensgen, R. Kinkel, and U. Manber, "Two Algorithms for Barrier Synchronization," *International Journal of Parallel Programming*, Vol.17 No.1, pp.1-17, 1988

[22] K.H. Cheng and Q. Wang, "A Simultaneous Access Design for Idle Processor Reactivation and the Detection of the Termination of a Parallel Activity," *Journal of Parallel and Distributed Computing* 17, pp.370-373, 1993

[23] H. Xu, P.K. McKinley, and L.M. Ni, "Efficient Implementation of Barrier Synchronization in Wormhole-Routed Hypercube Multicomputers," *Journal of Parallel and Distributed Computing* 15, pp.172-184, 1992

[24] K. Ghose and D.-C. Cheng, "Efficient Synchronization Schemes for Large-Scale Shared-Memory Multiprocessors," *Proceedings of the 1991 International Conference on Parallel Processing*, pp.153-158, 1991

[25] K. Hwang and S. Shang, "Wired-NOR Barrier Synchronization for Designing Large Shared-Memory Multiprocessors," *Proceedings of the 1991 International Conference on Parallel Processing*, pp.171-175, 1991

[26] S. Shang and K. Hwang, "Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters," *IEEE Transactions on Parallel and Distributed Systems*, Vol.6, No.6, pp.591-605, June 1995.

[27] C.J. Beckmann and C.D. Polychronopoulos, "Fast Barrier Synchronization Hardware," *CSRD Report No. 986*, University of Illinois, Urbana-Champaign, November 1990

[28] H.G. Dietz et al., "Purdue's adapter for parallel execution and rapid synchronization: the TTL_PAPERS design,' Jan. 1995

[29] J.-S. Yang and C.-T. King, "Designing tree-based barrier synchronization on 2D mesh networks," *IEEE Transactions on Parallel and Distributed Systems*, Vol.9, No.6, pp.526-533, June 1998.

[30] T. Muhammad, "Hardware Barrier Synchronization for A Cluster of Personal Computers," M.S. Thesis, Purdue University, May 1995

[31] A. B. Sinha and L.V. Kale, " A dyanmic and adaptive quiescence detection algorithm," Deparment of Computer Science, University of Illinois, Urbana, IL

[32] R. DeMara, B. Motlagh, C. Lin and S. Kuo, "Barrier synchronization techniques for distributed process creation," *8th International Parallel Processing Symposium Proceedings*, pp.597-603, April 1994.

[33] C.H. Roth, Jr., " Fundamentals of Logic Design," 3rd Edition, West Publishing Company, 1985

[34] T.E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessor," *IEEE Transactions on Parallel and Distributed Systems*, Vol.1, No.1, pp.6-16, Jan. 1990.

[35] K.M. Fant and S.A. Brandt, "NULL Convention Logic," Theseus Logic, Inc. document, www.theseus.com

[36] G.E. Sobelman and K. Fant, "CMOS circuit design of threshold gates with hysteresis," Theseus Logic, Inc. document, www.theseus.com

[37] I. Foster et al.,"The Grid, Blueprint for a new computing infrasturcture," Morgan Kaufmann, 1999

[38] D. I. Moldovan, "Parellel processing, from application to systems," Morgan Kaufmann, 1993

[39] W.J. Dally and C.L. Seitz, "The torus routing chip," *Distributed Computing*, Vol. 1, pp.187-196, 1986

[40] H.G. Dietz, R. Hoare, and T. Mattox, "A fine-grain parallel architecture based on barrier synchronization,' Proceedings of the Int'l Conf. on Parallel Processing, pp.247-250, Aug. 1996

[41] R. Hoare et al., "Bitwise aggregate networks," 8th IEEE Symposium on Parallel and Distributed Proceedings, Oct. 1996

[42] M.R. Zargham, "Computer architecture, single and parallel systems," Prentice-Hall, 1996

[43] G. Brassard and P. Bratley, "Algorithmics, theory and practice," Prentice-Hall, 1988

[44] D.E. Lenoski and W.-D. Weber, "Scalable shared-memory multiprocessing," Morgan Kaufmann, 1995

[45] K. Hwang and Z. Xu, "Scalable parallel computing," McGraw-Hill, 1998

[46] T.H. Lai and L.-F. Wu, "An $(N-1)$-resilient algorithm for distributed termination detection," *IEEE Transactions on Parallel and Distributed Systems*, Vol.6, No.1, pp.63-78, Jan. 1995.

[47] K. Hwang, "Advanced computer architecture, parallelism, scalability, programmability," McGraw-Hill, 1993

[48] J. Bacon, "Concurrent systems," 2nd Edition, Addison-Wesley, 1997

[49] D.E. Culler, J.P. Singh and A. Gupta, "Parallel computer architecture, a hardware/software approach," Morgan Kaufmann, 1999

[50] A.P. Malvino, "Electronic Principles," 2nd Edition, McGraw-Hill, 1979

[51] S. Venkatesan, "Reliable protocols for distributed termination detection,' *IEEE Transactions on Reliability*, Vol.38, No., pp.103-110, April 1989.

[52] T.L. Casavant, P. Tvrdik, and F. Plasil, "Parallel Computers, theory and practice,' IEEE Computer Society Press, 1996