Tiered Algorithm for Distributed Process Quiescence and Termination Detection

Ronald F. DeMara, Yili Tseng, and Abdel Ejnioui

Abstract—The *Tiered Algorithm* is presented for time-efficient and message-efficient detection of process termination. It employs a global invariant of equality between process production and consumption at each level of process nesting to detect termination regardless of execution interleaving order and network transit time. Correctness is validated for arbitrary process launching hierarchies, including *launch-in-transit* hazards where processes are created dynamically based on run-time conditions for remote execution. The performance of the Tiered Algorithm is compared to three existing schemes with comparable capabilities, namely the *CV*, *LTD*, and *Credit* termination detection algorithms. For synchronization of T tasks terminating in E epochs of idle processing, the Tiered Algorithm is shown to incur $O(E)$ message count complexity and $O(T \lg T)$ message bit complexity while incurring detection latency corresponding to only integer addition and comparison. The synchronization performance in terms of messaging overhead, detection operations, and storage requirements are evaluated and compared across numerous task creation and termination hierarchies.

Index Terms— Synchronization, Multitasking, Distributed Architectures, Distributed Programming, Parallel Processing.

1 Introduction

Efficient detection of process termination [1] is essential for optimizing throughput in distributed computer architectures and networks. An ensemble of Processing Elements (PEs) is said to be *synchronized*, or to have reached a *quiescent state* [2], upon termination of each interval of concurrent activity. Points at which synchronization occur are referred to as *barriers* [3] and their detection can significantly influence throughput since idle PEs cannot proceed to subsequent operations in the current thread until the barrier's completion has been signaled. In addition to execution overhead, the interchange of synchronization messages during barrier detection degrades the message transmission capacity available to the underlying computations [4].

Many existing termination detection algorithms require a-priori knowledge of process creation or depend on various attributes of the network topology for correct operation [3, 5-8]. However, the *Tiered Algorithm* developed herein belongs to a class of more capable termination detection schemes [2, 4, 9-11] that support more general diffusing models of distributed computation which allow *dynamic process creation* [12, 13]. Applications in which processes are created and destroyed based on run-time conditions, such as distributed garbage collection [14, 15], network multicasting [16], parallel marker-passing [17], and parallel polygon rendering in real time scientific visualization [18], explicitly require such capabilities. Detection of termination is complicated by persistence of child processes after the completion of their parents, as is the case with orphan processes encountered in these applications and others such as distributed databases [19, 20]. Rudimentary approaches to this problem which maintain the parent process until all of its children have terminated tend to waste system resources while introducing data consistency issues [19] which can be eliminated by more efficient and powerful algorithms for termination detection.

Among the few available algorithms which can properly detect global termination in the case of orphan processes [12], the Tiered Algorithm is shown

- *R. F. DeMara is with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816, E-mail: demara@ mail.ucf.edu.*
- *Y. Tseng is with Auricular Medicine International Research and Training Center, Fern Park, FL 32730, E-mail: ytseng@ieee.org.*
- *A. Ejnioui is with the Information Technology Department, University of South Florida, Lakeland, FL 33863, E-mail: aejnioui@lakeland.usf.edu.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

to do so at minimum overhead. It requires $O(\min(N, T))$ synchronization messages where N is the number of PEs in the system and T is the number of tasks created during execution of the distributed application. To a large extent, the reduced communication overhead is achieved by employing a *processor-centered* protocol. In contrast to process-centered protocols where a control message is sent from each terminating process to a parent or a global barrier detection process, a processor-centered protocol instead allows a single control message to be sent from each PE that accounts for all processes executed on that PE. Hence, processor-centered protocols are especially suitable for the typical scenarios where the number of processes equals or exceeds the number of PEs allocated to the computation [17]. Unlike *wave-based* termination detection schemes [8, 21-23] which send control messages periodically, the Tiered Algorithm requires that a PE transmits control messages only when it becomes idle.

Finally, static environments can be considered to be an equivalent case of a dynamic environment where all processes are created upon initialization. Thus, an approach which solves the most general dynamic termination detection problem, while incurring overhead of the more restrictive static case, offers a general solution. In this paper, the Tiered Algorithm is shown to support dynamic process creation environments at comparable costs to that of only static-capable termination detection schemes. As shown in [24], the theoretical minimum in terms of communication overhead is given by the $\min(N, T)$ message and is this achieved by the Tiered Algorithm. As described below, it achieves this bound by employing a global invariant of process nesting with a straightforward integer-based accounting scheme.

2 Related Work

A wide range of termination detection algorithms have been proposed both as software-only [2, 9, 10] and hardware-specific [3, 4, 6, 7] approaches. The most general software-based algorithms consider diffusing computation models [12, 21, 23, 25-27] in which one process initiates the distributed activity and launches subprocesses dynamically by transmitting messages to remote PEs. Two broad termination detection approaches for these environments are *parental responsibility algorithms* [12, 23, 26, 28-30] and *wave-based algorithms* [8, 21-23]. Parental responsibility algorithms infer global termination status from a progression of local process initiation and completion activities on each PE. Wave algorithms do not track process transactions continually, but instead periodically propagate waves of control messages to interrogate the current status of each PE. Some of these algorithms can detect termination on arbitrary network topologies [8, 12, 16, 22, 23, 26, 27, 29] while others require maintenance of process spanning trees or a specific underlying physical network topology [21, 23, 28, 30]. For example, algorithms which detect termination by using a virtual spanning tree [11, 23, 31, 32] require that a single process be designated as the root and have direct means to track status of its children [8, 16, 22, 27, 29]. Other algorithms require maintaining a *dependent set* [21] or a *neighbor set* [12, 30] of processes; others dictate that a controller process be aware of the network diameter [26] or synchronize activity through a local clock [11, 12, 33, 34].

As in this paper, more recent algorithms address a completely asynchronous communication model in which messages arrive in arbitrary order [12, 22, 23, 27, 29] without requiring FIFO channels [16, 26, 30] or assumptions of finite transmission delays [21]. The most general of these algorithms support a dynamic environment in which processes are created and destroyed as the underlying computation progresses [8, 12, 16, 21, 22]. However, many address only special cases of dynamic environments. For example, the algorithms proposed in [13, 25, 35] allow processes to be created, but not destroyed, while the algorithm in [11] requires that a process participates in termination detection even after it has been destroyed, and in [22] only one process is allowed to execute on each PE.

The best performing previous algorithms which support dynamic execution and do not impose message ordering nor topology constraints include the *Credit Algorithm* [2] which is a parental responsibility-based approach, and the *CV Algorithm* [9] and *LTD Algorithm* [10] which are wave-based approaches. The CV Algorithm organizes all processors participating in barrier as a logical spanning tree such that when the root terminates, it declares global termination. The LTD Algorithm refines the CV Algorithm by optimizing wave operations based on a local message stack maintained at each PE. The Credit Algorithm relies instead on a credit distribution invariant where the top-level parent process is given a unit credit of 1.0. Each time a subprocess is created, a parent gives half of its remaining credit to it. Later as processes are terminated, credit portions are returned to a central controller process. When the sum of the credit values returned equals 1.0, then global termination is declared. Meanwhile, the Tiered Algorithm employs an improved global invariant that allows processor-centered reporting and replacement of a time-consuming fraction combining step with integer addition.

3 Tiered Detection Algorithm

The Tiered Algorithm supports a simultaneous-initiation diffusing computation model. The number and binding of processes need not be known a-priori, and processes can be created or completed without restriction during execution. It does not assume any network topology and exchanges messages under an asynchronous communication model without any assumptions of message delivery neither ordering nor transmission time. As in the Credit Algorithm, every participating PE reports the count of locally produced and terminated tasks at each level of process nesting to the designated process, called the *controller* process, which will announce global termination. The controller updates a ledger of count values accordingly to determine whether the global consumption count matches the production count at every nesting level. If so, the controller announces global termination. Otherwise, the controller waits for the report as some processes have not yet completed execution. So, the global invariant for termination detection is that *process create* count received by the controller equals *process terminate* count received on a nesting level-by-level basis as described below.

Tiered reporting is a processor-centered mechanism in order to reduce message traffic under the usual condition $T >> N$ for T logical tasks on N physical processors. Processor-centered reporting means that PEs report status for all processes that they have initiated or completed. Furthermore, synchronization message traffic is incurred only when a PE becomes idle, unlike wave-based approaches.

3.1 PE Operation

Under a distributed tasking model, each PE maintains a local queue of processes to be executed [36]. A process is entered into the queue by receipt of a *process launch* message from a parent process executing on this PE or on another PE. Associated with each process launch message is an integer indicating the nesting *level* of the process that created it. The level number of the child process is obtained by adding an integer value of 1 to the level number of the parent where the root process is assigned level $L=0$. Figure 1 shows the local portion of the Tiered Algorithm executed on each PE.

```
Procedure Receive_TaskLaunch_Message(L : level number)
begin
  Update row L of activity table to increase produced
   count;
end

Procedure Finish_A_Task(L : level number)
begin
  Update row L of activity table to increase consumed
   count;
end

Procedure Upon_Idle
begin
  Report to controller non-zero difference for
    previously unreported rows of activity table;
end
```

Figure 1. PE operation in the Tiered Algorithm.

Figure 2(a) shows the *activity table* maintained by each PE. The activity table records the local process consumption and production counts for each level on that PE. A PE's consumption count values indicate the number of tasks that were locally consumed at each level on this PE. Likewise, the production count represents the number of tasks launched at each level at the local PE. The count values can be stored to exploit a unique relationship by which the tasks dispatched by the k^{th} level are also the tasks created on the $(k + 1)^{th}$ level. Since the equality of the number of tasks launched at a specific level and the number of tasks consumed at the same level is critical, it is sufficient to maintain the difference between the two numbers for each level k as *DIFF(k).* As such, the number of quantities maintained and communicated is reduced in half. Hence, a one-dimension table, shown in Figure 2(b), is maintained for the difference between the local consumption and production counts at each level of process nesting. Whenever a launch message is received by a PE, the procedure *Receive_TaskLaunch_Message* is called to update the local activity table which increments the level number. Likewise, the procedure *Finish_A_Task* is called whenever a task is completed at a PE by updating the local activity table according to the level number which is associated with the finished task.

Level	Consumption Count	Production Count
0	0	4
1	4	6
2	6	8
• •		
(D – 1)	5	7
D	7	6

(a) Theoretical data structure.

DIFF(1)
DIFF(2)
• •
DIFF(D–1)
DIFF(D)

(b) Implementation table.

Figure 2. Activity tables for process creation/termination

The update consists of decrementing the number in the corresponding table cell. After the PE finishes all the tasks in its execution queue and becomes idle, the procedure *Upon_Idle* is invoked to report the difference between the numbers of consumed and produced tasks for each level to the controller. Only levels with nonzero *DIFF* values need be reported. Once a PE reports to the controller, there is no loss of availability as the PE can be reactivated by any new process launching messages that are subsequently received from remote PEs. With the exception of the Credit Algorithm, processor reactivation capability, immediately upon reporting, is not typically supported by previous termination detection schemes. In the case of the Tiered Algorithm, correctness is maintained even if reactivation occurs with new processes that contribute to the same barrier that has been previously reported by that PE.

3.2 Operation of the Controller

The controller maintains a *ledger table* to keep track of the global consumption and production counts using the control messages reported from the PEs. Using the same rationale as for the activity table for a PE, a one-dimension table suffices where only the difference between the consumption and production counts for each level is maintained.

Figure 3 shows the algorithm for the controller. Whenever a PE reports to the controller, the controller invokes the *Receive_Report* procedure.

```
Procedure Receive_Report(r : report)
begin
  Update ledger and idle table accordingly;
  if (Check_Ledger)
    Declare global termination;
  endif
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;
  endif
end
```

Figure 3. Operation of the controller in the Tiered Algorithm.

It updates the ledger table accordingly based on the information sent by the reporting PE. This can result in an increase or decrease in the value stored in the corresponding level cell of the ledger table by the amount reported. Next, the controller process evaluates Check_Ledger. If the difference values in all cells of the ledger table are zero, meaning all tasks launched to all levels have been consumed, the global termination has been reached. If the value of any cell in the ledger table is not zero, meaning that there are still messages in transit and/or PEs still active, then the controller exits the procedure as global termination cannot be declared until after the next report is received.

3.3 Correctness Proof

A correctness proof of a dynamic process creation termination detection technique needs to demonstrate that the barrier is announced if and only if all PEs have entered an idle state and simultaneously that no process launch messages are in transit in the network. The correctness of the Tiered Algorithm uses a proof by induction based on the following parameters:

- *Task launching hierarchy*: a tree-structured task graph with a root node at level 0 representing the main process in the original thread's task.
- *Level:* a positive integer associated depth of the *task launching hierarchy* assigned such that all processes operating in level $k > 0$ are launched by processes at level $k - 1$.
- *Launch message*: process create control message transmitted from parent to child process, either on the same PE or to a remote PE.
- *Launch-in-transit hazard*: occurs when PEs temporarily satisfy the idle-state condition of the barrier while the barrier is actually incomplete, i.e. one or more process launch message(s) is still in-transit in the interconnection network.
- *Terminate message*: a transmission from a PE to the controller indicating idle-status and the number of all processes locally produced and locally consumed at that level.

To determine the correctness of the Tiered Algorithm, it must be shown that the controller indicates that the barrier is completed if and only if it detects the termination of all processes at each level of the process launching hierarchy. In the case of the Tiered Algorithm, the basis statement is: *For every level $L \geq 1$ in the process launching hierarchy, the Tiered Algorithm* (i) *detects the completion of all processing at level L, and (ii) properly detects the total number of processes created at level L+1, thereby correctly determining when the synchronization barrier has been reached.* The induction proof of the algorithm follows whereby (i) it is shown that the basis statement is true for L=1, and (ii) if it is assumed that the basis statement is true for some L=k, where $k > 1$, then it is true for L=k+1:

- Step (i): L=1 activity is launched by a broadcast command from the controlling node. This activity occurs at all N nodes of the network. While this broadcast message may not cause application processing at all nodes in the network, every PE responds with at least one L=1 termination message indicating inactivity. Therefore, the controller knows how many terminate messages are to be received before L=1 processing can be considered complete. Since only an L=1 task can launch an L=2 task, all L=2 task launching will have been initiated before the time the controller detects the completion of L=1 processing, as PEs report only when they become idle. By the definition of a terminate message, the controller is able to determine the number of L=2 tasks if it has received all L=1 terminate messages.
- Step (ii): If L=k has been completed and properly handled, the controller node knows how many level L=(k+1) processes have been launched. The completion of processing at L=k+1 is detected when the number of terminate messages received for this level matches the number of processes launched by level L=k. By the definition of the terminate message, when all L=(k+1) terminate messages are received, the total number of tasks launched at L=(k+1) will be known by the controller. The barrier is reached when the total number of processes launched by L=(k+1) is zero.

Hence the barrier is known to be reached when all cells of the ledger table are zero since that implies the entry for level L=(k+1) be zero. An optimization for the *Check_Ledger* task is that a pointer can be advanced past each level in the ledger table as it becomes zero, thus reducing the number of levels still remaining to be checked. This reduces the global detection latency when all tasks finally complete by restricting ledger checking to just the levels of those tasks which were most recently executing.

4 Performance Analysis

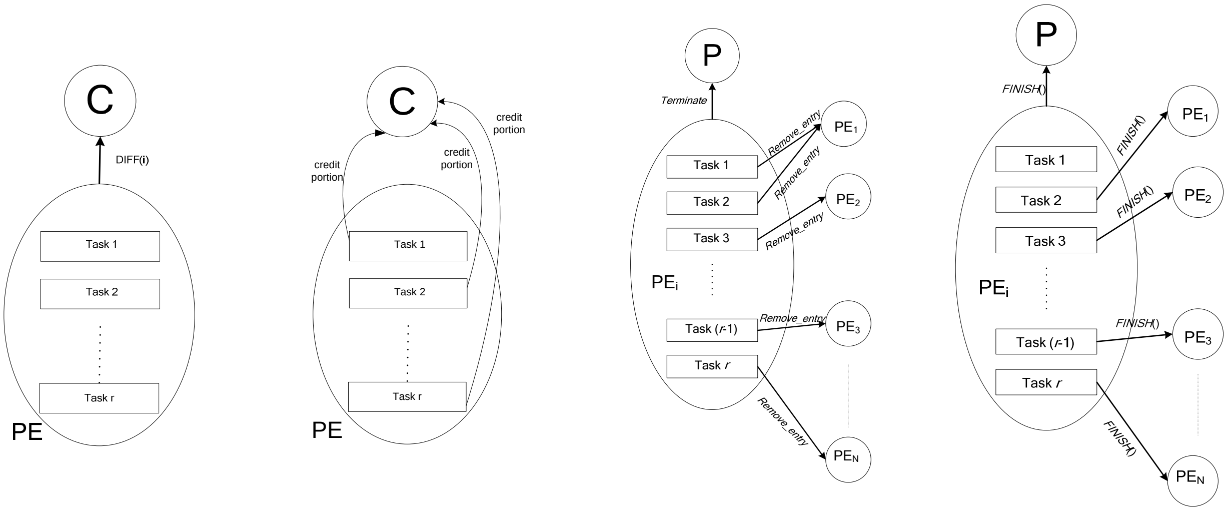
Table 1 lists the parameters used in the analysis of the four termination detection algorithms capable of supporting dynamic process creation environments. Each algorithm needs to attach specific information to the initializing messages of the underlying computation. The Tiered Algorithm attaches the level number, the Credit Algorithm attaches the credit value, and the CV and LTD Algorithm attach the PE identification number referred to as the *PE ID*. Since this information is appended to the existing task launch messages, these messages can be considered as required by the underlying computation itself, so that the overhead of synchronization-related messages only includes additional messages as required by the termination detection algorithm.

TABLE 1. PERFORMANCE ANALYSIS PARAMETERS.

Parameter	**Quantity Measured**
Epoch	Duration of processing which occurs between barriers
N	Number of physical PEs in the computing system
E	Number of idle events which occur in an epoch
M_i	Number of internal notifications during the processing interval preceding the i^{th} idle event
T	Number of logical tasks created during an epoch
D	Maximum depth of task nesting levels during any epoch
F	Fanout or number of links between physical PEs
t_{send}	Message transit time between source and destination PE
$t_{checkup}^{Protocol}$	Time required for termination criterion checkup of a specific protocol
$t_{combine}^{integer}$	Time required to perform an integer addition and determine if a ledger entry is null
$t_{combine}^{set}$	Time required to subtract an element from a set of elements
$t_{cleanup}^{stack}$	Time required to pop all the entries in the stack of a PE until a sending entry is found in the stack

4.1 MESSAGE COMPLEXITY

Message complexity accounts for the number of messages required to detect termination. To be consistent with existing literature, every terminating process is said to send one *internal notification message* to indicate completion of a process on that same PE in the PE's local queue [10]. Hence the algorithms eventually require $\sum_{i=1}^{E} M_i = T$ internal notifications for T tasks in the epoch. Since there are E events in the epoch, E quantity of *external notification messages* are required from one physical PE to another [10]. In the case of the Tiered Algorithm as depicted in Figure 4(a), a PE is allocated r tasks, but only one message is transmitted containing the *DIFF* value at nesting level i to the controller C. Thus, (T + E) messages are required for internal and external messages overall. However, in the Credit Algorithm, every task sends one external message to the controller after it terminates containing the numerical value of its credit portion. As shown in Figure 4(b), the number of external messages sent to the controller is equal to the number of tasks for each PE while the total number of tasks across all PEs totals T. On the other hand, in the case of the CV Algorithm shown in Figure 4(c), every task in an event needs to send an external *remove_entry* message to its sender so T external messages are sent. The PE, where the event resides, needs to send a terminate message to its logical parent P. Hence, (N – 1) external messages are required for (N – 1) children PEs. However, (N – 1) messages instead of (E – 1) messages are needed in this context. Combined with 2F external messages to build the logical spanning tree of PEs, (2F + 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, every task in an event needs to send one internal notification amounting to T messages. In the case of the LTD Algorithm, the number of messages required depends on the mapping of the tasks. As shown in Figure 4(d), some tasks are launched by the same PE. In this case, the event needs to report to the launching PE with only one *FINISH* message instead of several messages as is the case in the CV Algorithm. In the worst case, every task in any event is launched by a different PE to the point where the performance is similar to the CV Algorithm where $\sum_{i=1}^{E} M_i - 1 = (T - 1)$ external *FINISH* messages are generated. However, in the best case, the performance of the LTD Algorithm matches that of the Tiered Algorithm where every task in an event is executed by the same PE. Therefore, only one external message is reported by each PE except for the event occurring on the root node. Additionally, to initialize each wave of termination reporting, (N – 1) external messages are required to inform the *Detecting Termination* status [10]. As the required number of internal notifications amounts to T in all cases, the overall number of messages required by the LTD Algorithm ranges from (N + T + E – 2) to (N + 2T – 2).



(a) Tiered (b) Credit (c) CV (d) LTD

Figure 4. External messages transmission after a PE becomes idle.

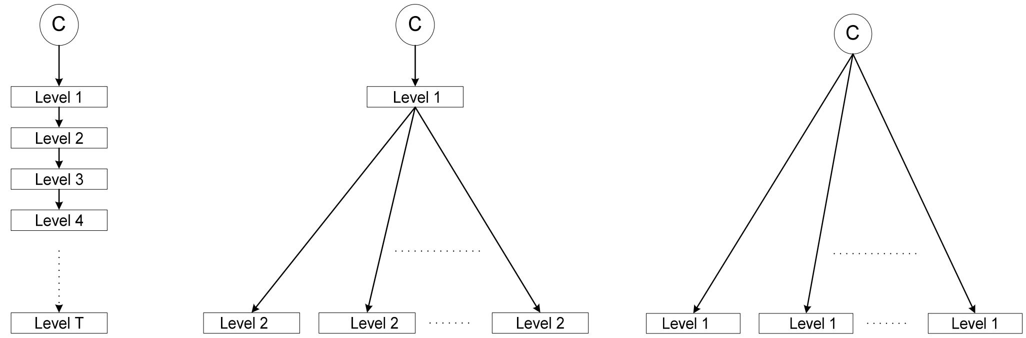
As summarized in Table 2, the Tiered Algorithm outperforms the other algorithms by incurring the same number of synchronization messages since the total number of idle reporting events, *E*, which is the least of the four algorithms. Note that by definition, $E \leq T$ while $F > 1$ and $N > 1$. The Credit Algorithm needs as many messages as tasks while the CV Algorithm needs more messages than the number of tasks. Finally, the LTD Algorithm's performance lies somewhere in between depending on the termination interleaving.

4.2 Bit Complexity

Bit Complexity accounts for the number of bits transmitted to detect termination. In the Tiered Algorithm, every report consists of two fields, namely the level number and the difference between the production and consumption counts in the matching level. The maximum level number of an epoch with *T* tasks is *T* when all tasks are dispatched sequentially to different levels as shown in Figure 5(a). Hence, $\lceil \lg T \rceil$ bits are required. The maximum value of *DIFF(i)* that can occur within an epoch having *T* tasks is $(T - 1)$. As shown in Figure 5(b), this occurs when the controller launches a single task which in turn launches all the remaining $(T - 1)$ tasks. So approximately $\lceil \lg T \rceil$ bits are also required for the difference field while a basic report unit requires $2\lceil \lg T \rceil$ bits. The worst case occurs when all tasks are dispatched to different levels of the logical tree and are physically allocated to unique PEs. In that case, the PE needs to report 2 messages consisting of *DIFF(i)=–1* and *DIFF(i)=1* corresponding to "one task consumed and one task produced" because no two tasks from adjacent levels are dispatched to the same PE. Eventually, $2T$ reports are required for *T* finished tasks. The worst case takes $4T\lceil \lg T \rceil$ bits. On the other hand, the least transmission occurs when all tasks are dispatched to the first level as shown in Figure 5(c). Since all tasks are in the first level, all tasks dispatched to the same event require only one report. Finally, *E* basic reports are required to cover all consumed tasks dispatched to the *E* events. Thus, the best case requires $2E\lceil \lg T \rceil$ bits. In the CV Algorithm, the message needs to identify its own type and from which PE it originates. To this end it is assumed that a message consists of two fields: PE ID and message ID, requiring $\lceil \lg N \rceil$ bits and $(2L + T + N - 1)(\lceil \lg N \rceil + 2)$ bits, respectively. For the LTD Algorithm, two fields are used: message ID and amount.

TABLE 2. MESSAGE COMPLEXITY.

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



(a) Case 1. (b) Case 2. (c) Case 3.

Figure 5. Extreme dispatching cases in the Tiered detection algorithm.

The amount field, which represents the number of messages reported by *FINISH*(n), needs $\lceil \lg T \rceil$ bits since the largest possible number of messages that could be reported is T. Hence, the number of bits required by the LTD Algorithm can range from $(E+N-1)(\lceil \lg T \rceil+1)$ to $(T+N-1)(\lceil \lg T \rceil+1)$. As summarized in Table 3, the Credit algorithm displays performance with a complexity of $\Theta(T \lg T)$. This indicates that it always needs $(T \lg T)$ bits. On the other hand, the CV Algorithm is slightly better than the Credit Algorithm with a complexity of $\Theta(T \lg N)$.

4.3 DETECTION DELAY

Detection delay accounts for the interval from when the last task ends until the controller process announces global termination. In all cases of the Tiered and Credit algorithms, the PE sends a report to the controller after the last task ends. The detection delay can be expressed as $\left(t_{send} + t_{checkup}^{protocol}\right)$ where t_{send} is the message transit time and $t_{checkup}^{protocol}$ is the time taken by the final execution procedure for a given protocol. In the Tiered Algorithm, the controller balances the ledger table entries for any non-zero levels and concludes global termination. In the Credit Algorithm, credits are kept as floating-point values, or more optimally as negative exponent fractions of powers of two in a set called *DEBTS* that needs to be combined at the controller [2]. As for the CV Algorithm, the detection delay depends on the location of the last task in the physical tree of PEs. The worst case occurs when only one task is dispatched to each of the first (N – 1) PEs, the remaining tasks are dispatched to the last PE in the tree of PEs while the last ending task resides in the last PE. After the last task ends, the last PE needs to first send (T – N + 1) *remove_entry* messages serially, which takes time (T – N + 1)t_{send}. Next, it checks its status and sends terminate to its parent. In return, its parent also checks its status and sends terminate one level higher.

TABLE 3. BIT COMPLEXITY.

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	$(E+N-1)\times(\lceil \lg T \rceil + 1)$	$(T+N-1)\times(\lceil \lg T \rceil + 1)$	$O(T \lg T)$

This process goes on in every PE, except in the root PE of the physical tree, thus taking $(N-1)(t^{CV}_{checkup}+t_{send})$. Receiving the terminate message from its child, the root PE checks the status and concludes global termination, which takes $t^{CV}_{checkup}$. In total, the detection delay for the worst case is $(Tt_{send}+Nt^{CV}_{checkup})$. On the other hand, the best case occurs when the last ending task resides in the root PE. The root PE checks the status and concludes global termination with the detection delay denoted by $t^{CV}_{checkup}$. In the case of the LTD Algorithm, the situation is very similar to that of the CV Algorithm since it all depends on where the last ending task is located. The worst case occurs when the tasks are dispatched where the last ending task resides in the deepest PE requiring $(2N-3)t_{send}+(N-1)t^{LTD}_{checkup}$. Both require a *stack cleanup* operation [9] which takes $t^{stack}_{cleanup}$ In the best case, the root PE checks its status and concludes global termination, which takes only $t^{LTD}_{checkup}$. These results are summarized in Table 4 where the Tiered Algorithm exhibits performance related to the complexity of an integer combining step, i.e., addition and determination if ledger entry is equal to zero. The other algorithms require more complex messaging or combining operations.

4.4 Storage Complexity

In the Tiered Algorithm, the controller needs to maintain a ledger table with space for *T* records reserved for possible *T* levels in the worst case. Because the index of the records in the table can serve as the level number implicitly, there is no need to set a field for the level number in the table. The largest possible number for level difference is (*T* − 1), hence $\lceil \lg T \rceil$ bits are sufficient for each record. In total, $T\lceil \lg T \rceil$ bits are required for the ledger table. In the Credit Algorithm, a debt bookkeeping technique is proposed [2] in order to avoid underflow problems and process exponents. This technique maintains a *DEBTS* set so whenever a task becomes idle and returns its credit share, the controller removes it from the *DEBTS* set. When the *DEBTS* set becomes empty, termination is concluded. The controller needs space to maintain the set. The worst case, similar to the case shown in Figure 5, occurs when all *T* tasks are active. Therefore, $T\lceil \lg T \rceil$ bits are needed to accommodate the worst case. As for the CV Algorithm, every PE maintains a stack to record sending and receiving activities. The stack must be sufficiently large to accommodate (*T* − *N* + 1) records, each of which are $\lceil \lg N \rceil$ bits wide. The space required is $N(T-N+1)\lceil \lg N \rceil$ bits in total for *N* PEs. Hence, the storage complexity is $O(NT \lg N)$. As previously described, every node in the LTD Algorithm has to maintain four variables [10]. 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* PEs need $N(N\lceil \lg T \rceil + \lceil \lg N \rceil + 1)$ bits. As summarized in Table 5, the Credit and the LTD algorithms require less space than the other two algorithms.

TABLE 4. DETECTION DELAY.

Algorithm	Best Case	Worst Case	Complexity
Tiered Algorithm	$\left(t_{send} + t_{checkup}^{Tiered}\right)$	$\left(t_{send} + t_{checkup}^{Tiered}\right)$	$O\left(t_{combine}^{integer}\right)$
Credit Algorithm	$\left(t_{send} + t_{checkup}^{Credit}\right)$	$\left(t_{send} + t_{checkup}^{Credit}\right)$	$O\left(t_{combine}^{set}\right)$
CV Algorithm	$t_{checkup}^{CV}$	$\left(Tt_{send} + Nt_{checkup}^{CV}\right)$	$O\left(T + N \times t_{cleanup}^{stack}\right)$
LTD Algorithm	$t_{checkup}^{LTD}$	$\left(2N-3\right)t_{send} + \left(N-1\right)t_{checkup}^{LTD}$	$O\left(N \times t_{cleanup}^{stack}\right)$

TABLE 5. STORAGE COMPLEXITY.

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

5 Experimental Evaluation

The Tiered Algorithm performance is compared directly against the Credit Algorithm since they are parent-responsibility algorithms and both outperform CV and LTD by the metrics in Section 4. Experimental evaluation consists of a benchmark of 100 task nesting hierarchies [36] of varying depth, size, and characteristics of task creation and termination as shown in Figure 6. The number of tasks in the hierarchies ranged from 101 to 703 with a mean of 311 tasks.

5.1 Synchronization Message Overhead

The volume of a synchronization messages was quantified using three metrics: (i) the number of synchronization messages; (ii) the number of element values returned over all synchronization messages; and (iii) the total number of bits based on the size of the transmitted elements. Figure 7 shows the number of messages in the Credit Algorithm with a second curve representing the difference between the number of messages in the Credit and the Tiered Algorithm. A positive difference in the latter curve indicates an advantage for the Tiered Algorithm consistent with the analysis in Section 4.1. Mean traffic was 263 messages vs. 302 messages while maximum traffic was 395 messages vs. 680 messages, for the Tiered and Credit algorithms respectively.

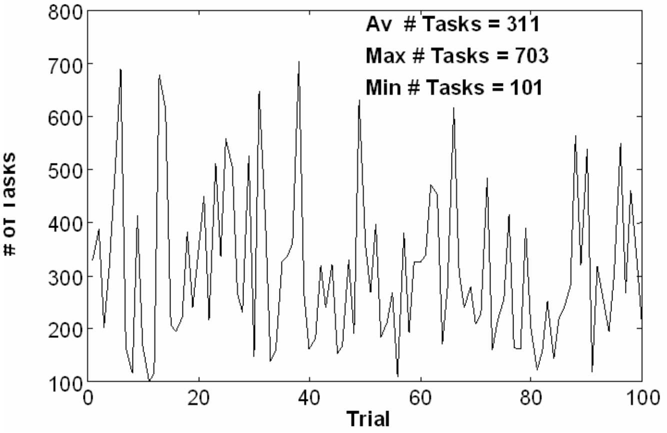


Figure 6. Task hierarchies.

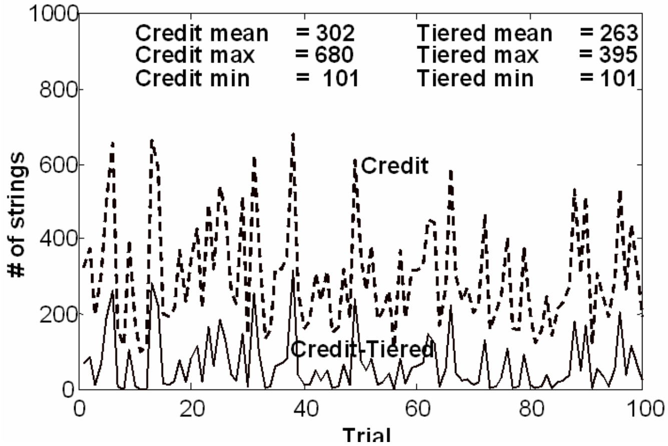


Figure 7. Synchronization messages.

The difference curve in Figure 8 shows that the Tiered Algorithm returns a larger number of elements than the Credit Algorithm does. However, the difference curve in Figure 9 shows that the Tiered Algorithm requires fewer bits to do so than the Credit Algorithm does. In the case of the Tiered Algorithm, the maximum value of the element returned to the controller typically matches the maximum number of task levels created. As shown in Figure 9, this allows the Tiered algorithm to reduce message traffic by 24% on average and by 30% in the best case when compared to Credit while exhibiting less standard deviation. Note that the maximum task level of nesting establishes a lower limit on the maximum size of the credit list. While the Tiered Algorithm may return more elements to the controller, the comparatively small values represented by these elements allow message encoding requiring fewer bits per message. Nonetheless such reductions may be eliminated during packetization on a store and forward network.

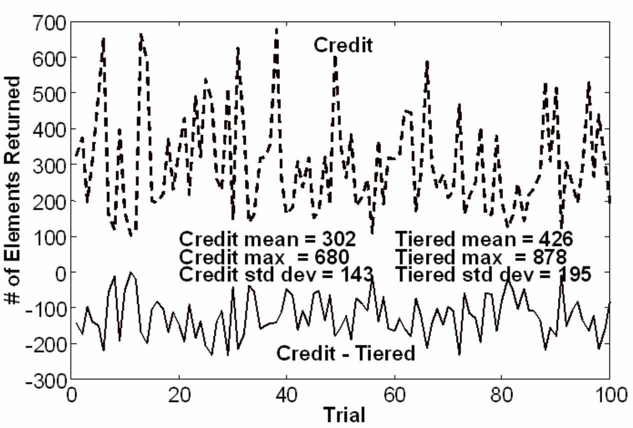


Figure 8. Synchronization elements returned.

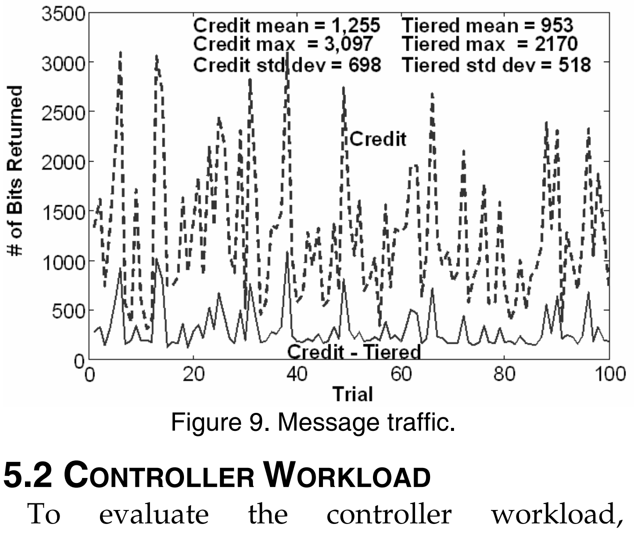


Figure 9. Message traffic.

5.2 Controller Workload

To evaluate the controller workload, equivalent machine-level instructions were tabulated for both the Tiered and Credit algorithms as shown in Figure 10. Because the Credit Algorithm relies on complex operators, such as the combining elements in the DEBTS set, it tends to generate a significantly larger workload than the Tiered Algorithm since even the optimization for the Credit algorithm requires set subtraction utilizing $O(|S|)$ operations assuming a linked list implementation of set S is maintained. In addition, the variation in workload imposed on the controller by the Credit Algorithm is significantly greater than that of the Tiered Algorithm. A worst case analysis would need to anticipate the largest of these workloads which was 15.3-fold larger in the case of the Credit Algorithm when compared to the Tiered Algorithm as depicted in Figure 10.

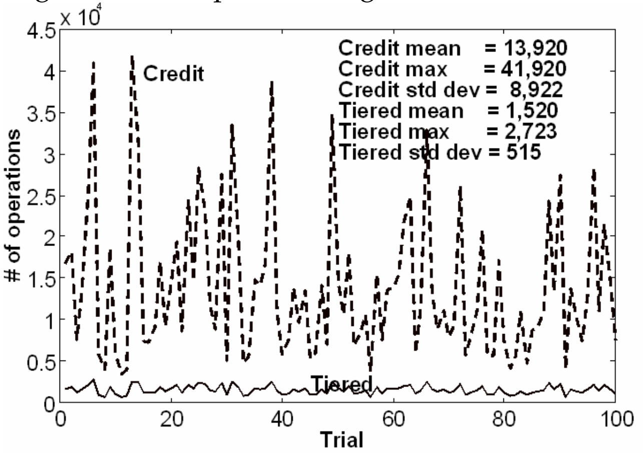


Figure 10. Controller operations to detect termination.

5.3 Size of the Data Structure in the Controller

In the Tiered Algorithm, there is a one-to-one correspondence between the number of elements in the data structure maintained by the controller and the maximum depth of nesting D that occurs during execution. However, in the Credit Algorithm, the size of DEBTS set is bounded below by D, yet can range up to the maximum credit value that is created during the execution of the application, which may be as high as T. While their worst case asymptotic storage complexities are comparable as described in Section 4.4, it is shown in Figure 11 that the Tiered Algorithm is consistently preferable for a wide range set of tasks with maximum size at 12/19 = 63% of the maximum Credit structure size.

6 Conclusion

Given its broad capabilities for supporting both static and dynamic process creation environments at low overhead, the Tiered Algorithm offers a general approach to termination detection. It performs well under widely varying characteristics of the number of created and terminated processes and depth of process nesting using metrics of message and storage complexity.

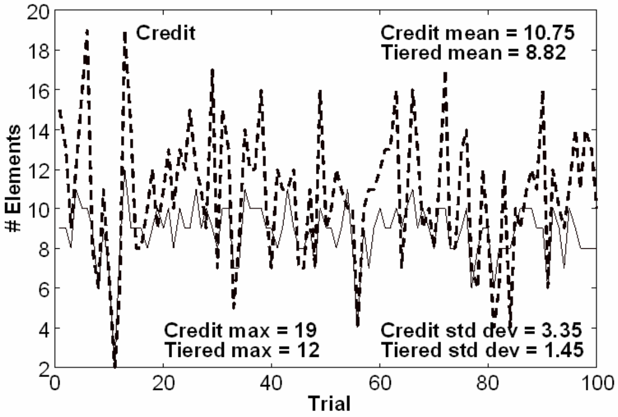


Figure 11. Controller data structure.

When compared to wave-based algorithms, the Tiered Algorithm's use of invariance among equality of production and consumption counts at each nesting level to indicate global termination eliminates the necessity to periodically interrogate the status of PEs, which suspends throughput during the checking process. When compared to a parental responsibility-based algorithm with comparable capabilities such as the Credit Algorithm, the practice of computing the difference between the production and consumption counts, instead of respective individual credit portions, reduces the bit complexity almost by half. In addition, the Tiered Algorithm allows the last finishing task report to incur integer math for just the deepest level of process nesting in contrast to the computationally intensive binary exponent *DEBTS* set subtraction and union operations encountered in the Credit Algorithm

References

[1] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, vol. 43, no. 3, pp. 207-221, Nov. 1998.

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

[3] W. E. Cohen, D. W. Hyde, and R. K. Gaede, "An optical bus-based distributed dynamic barrier mechanism," *IEEE Transactions on Computers*, vol. 49, no. 12, pp. 1354-1365, Dec. 2000.

[4] Y. Tseng, R. F. DeMara, and P. Wilder, "Distributed-sum termination detection supporting multithreaded execution," *Parallel Computing*, vol. 29, no. 7, pp. 953-968, Jul. 2003.

[5] J. Matocha, "Distributed termination detection in a mobile wireless network," *Proc. ACM 36th Annual Southeast Regional Conference*, Marietta, GA, 1998, pp. 207-213.

[6] S. Moh, C. Yu, B. Lee, H. Y. Youn, D. Han, and D. Lee, "Four-ary tree-based barrier synchronization for 2D meshes without nonmember involvement," *IEEE Transactions on Computers*, vol. 50, no. 8, pp. 811-823, Aug. 2001.

[7] 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, Jun. 1995.

[8] A. B. Sinha and L. V. Kale, "A dynamic and adaptive quiescence detection algorithm," University of Illinois at Urbana-Champaign, Urbana-Champaign, 1993, available at http://citeseer.ist.psu.edu/sinha93dynamic.html.

[9] 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, Mar. 1990.

[10] T.-H. Lai, Y.-C. Tseng, and X. Dong, "A more efficient message-optimal algorithm for distributed termination detection," *Proc.*

Sixth International Parallel Processing Symposium, 1992, pp. 646-649.

[11] T.-H. Lai, "Termination detection for dynamic distributed systems with non-first-in-first-out communication," *Parallel and Distributed Computing*, vol. 3, no. 4, pp. 577-599, Dec. 1986.

[12] D. M. Dhamdhere, S. R. Iyer, and E. K. K. Reddy, "Distributed termination detection for dynamic systems," *Parallel Computing*, vol. 22, no. 14, pp. 2025-2045, Mar. 1997.

[13] S. Cohen and D. Lehman, "Dynamic systems and their distributed termination," *Proc. Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, 1982, pp. 29-33.

[14] G. Tel and F. Mattern, "The derivation of distributed termination detection algorithms from garbage collection schemes," *Proc. Parallel Architectures and Languages in Europe*, *Lecture Notes in Computer Science*, E. H. L. Aarts, J. Van Leeuwen, and M. Rem, Eds.: Springer-Verlag 505, 1991, pp. 137-149.

[15] N. C. Juul and E. Jul, "Comprehensive and robust garbage collection in a distributed system," in *International Workshop on Memory Management*, *Lecture Notes in Computer Science*, Y. Bekkers and J. Cohen, Eds. St. Malo, France: Springer-Verlag 637, Sep. 1992, pp. 103-115.

[16] G. Stupp, "Stateless termination detection," *Proc. International Symposium on Distributed Computing*, Toulouse, France, 2002, pp. 163-172.

[17] R. DeMara, B. Motlagh, C. Lin, and S. Kuo, "Barrier synchronization techniques for distributed process creation," *Proc. International Parallel Processing Symposium*, Apr. 1994, pp. 597-603.

[18] T. W. Crockett and T. Orloff, "Parallel polygon rendering for message-passing architectures," *Proc. IEEE Parallel & Distributed Technology: Systems & Applications*, Summer 1994, pp. 17-28.

[19] M. P. Herlihy and M. S. McKendry, "Timestamp-based orphan elimination," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 825-831, Jul. 1989.

[20] L. Svobodova, "File-servers for network-based distributed systems," *ACM Computing Surveys*, vol. 16, no. 4, pp. 353-396, Dec. 1984.

[21] X. Wang and J. Mayo, "A general model for detecting distributed termination in dynamic systems," *Proc. International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, Apr. 2004, pp. 26-30.

[22] A. H. Baker, S. Crivelli, and E. R. Jessup, "An efficient parallel termination detection algorithm," Lawrence Berekeley Lab, Berkeley, California, 2000, available at http://www.cs.colorado.edu/~jessup/selected_publications.htm.

[23] A. Khokhar, S. Hambrusch, and E. Kocalar, "Termination detection in data-driven parallel computations/applications," *Journal of Parallel and Distributed Computing*, vol. 63, no. 3, pp. 312-326, Mar. 2003.

[24] K. M. Chandy and J. Misra, "How processes learn," *Distributed Computing*, vol. 1, no. 1, pp. 40-52, Mar. 1986.

[25] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1-4, Aug. 1980.

[26] N. Mittal, S. Venkatesan, and S. Peri, "Message-optimal and latency-optimal termination detection algorithms for arbitrary topologies," *Proc. Annual Conference on Distributed Computing*, Trippenhuis-Amsterdam, The Netherlands, Oct. 2004.

[27] M. Filali, P. Mauran, and G. Padiou, "Refinement based validation of an algorithm for detecting distributed termination," *Proc. International Workshop on Formal Methods for Parallel Programming*, 2000, pp. 1027-1036.

[28] D. Kumar, "Development of a class of distributed termination detection algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 2, pp. 145-155, Apr. 1992.

[29] N. R. Mahapatra and S. Dutt, "An efficient delay-optimal distributed termination detection algorithm," University of Minnesota, 1994, available at http://www.ece.uic.edu/~dutt/papers/parallel/jpdc-term-detn-2003.pdf.

[30] C. Xu and F. C. M. Lau, "Efficient termination detection for loosely synchronous applications in

multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 537-544, May 1996.

[31] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, no. 3, pp. 167-175, Sep. 1987.

[32] I. Lavallee and G. Roucairol, "A fully distributed spanning tree algorithm," *Information Processing Letters*, vol. 23, no. 2, pp. 55-62, Aug. 1986.

[33] S. P. Rana, "A distributed solution to the distributed termination problem," *Information Processing Letters*, vol. 17, no. 1, pp. 43-46, Jul. 1983.

[34] J. Mayo and P. Kearns, "Distributed termination detection with roughly synchronized clocks," *Information Processing Letters*, vol. 52, no. 2, pp. 105-108, Oct. 1994.

[35] J. Misra and K. M. Chandy, "Termination detection of diffusing computations in communicating sequential processes," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1, pp. 37-43, Jan. 1982.

[36] K. Drake, "Time and space efficient multiprocessor synchronization and quiescence detection," M.S. Thesis, Dept. of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 1995.

Ronald F. DeMara received the Ph.D. degree in Computer Engineering from the University of Southern California in 1992. Since 1993, he has been a full-time faculty at the University of Central Florida in the Department of Electrical and Computer Engineering. He is an Associate Editor of the Journal of Circuits, Systems, and Computers and IEEE Transactions on VLSI Systems. He is a Senior Member of IEEE and a Member of ACM and ASEE

Yili Tseng received the BS degree in mechanical engineering from National Taiwan University in 1985 before he served as a second lieutenant of artillery in Taiwanese Army for two years. Later he received the MS degree in engineering science from University of Florida in 1990, another MS, and PhD degrees, both in computer engineering, from University of Central Florida in 1995 and 2000, respectively. He is currently with Auricular Medicine International Research and Training Center. Prior to that, he was an assistant professor for four years in Department of Computer and Information Sciences at Florida A & M University. His research interests include high-performance computing, grid computing, parallel and distributed computing, and numerical methods. He is a member of the IEEE and the IEEE Computer Society.

Abdel Ejnioui is currently an Assistant Professor in the Information Technology Department at the University of South Florida. He was previously a faculty member in the Department of Electrical and Computer Engineering of the University of Central Florida. He obtained his M.S. and Ph.D. in Computer Science and Engineering degrees from the University of South Florida in 1995 and 1999 respectively. His research interests include reconfigurable computing, computer architecture, and VLSI design. He is a member of the IEEE and the IEEE Computer Society.