

Performance Evaluation of Hierarchy Annotation and Credit Distribution Quiescence Mechanisms

Ronald F. DeMara, Kenneth Drake, Abdel Ejnoui

Department of Electrical and Computer Engineering
University of Central Florida
4000 Central Florida Blvd
Orlando, Florida 32816-2450
U.S.A

Abstract—*This paper evaluates the execution characteristics of two high-capability software-based approaches for detecting termination in distributed environments. The Tiered Algorithm relies on use of a global invariant that indicates equality between process production and consumption at each level of process nesting. The Credit Algorithm relies on the distribution of a unit value from the initial parent process that can only be reconstituted if the barrier is complete. While both strategies can detect termination correctly regardless of the execution ordering, avoid potential race conditions caused by unpredictable transit times, and support arbitrary run-time binding of logical processes to physical processors, they each exhibit different message count complexity, message bit complexity, controller overhead, and detection delay. These metrics were assessed under 100 randomly generated trials consisting of 101 to 703 tasks under a variety of task creation and termination ordering scenarios. The Tiered Algorithm exchanged an average 12.9% fewer synchronization messages and 24.1% fewer bits in total than the Credit Algorithm. Results indicate that while the Tiered Algorithm required 80% fewer controller operations, the Credit Algorithm contributed an average of 3.9 fewer operations affecting detection latency with less variability.*

I. INTRODUCTION

Efficient detection of process termination is essential for throughput optimization in distributed computer architectures and networks. An ensemble of Processing Elements (PEs) is said to be *synchronized*, or to have reached a *quiescent state*, upon completion of each interval of concurrent activity [1-3]. Points at which synchronization occur are referred to as *barriers* [4-10]. The *quiescence detection problem* [1, 3] or *termination detection problem* [2, 11-13] has been studied extensively at several levels of detail and sophistication. In conjunction with such studies, various detection

algorithms have been proposed both as software [1-4, 11, 13-15] and hardware [7-10, 12, 16-18] approaches. Termination detection performance can significantly influence the overall throughput since idle PEs cannot proceed to subsequent operations in the current thread until completion of the barrier has been signaled. Even when PEs are reactivated to perform tasks from another thread in order to utilize these processing cycles, significant overhead can be incurred. In addition to execution overhead, the interchange of synchronization messages during the detection process can degrade the message transmission capacity for the underlying computations being performed [12].

In this paper, we evaluate the performance of two algorithms with provably correct operation under the most demanding conditions corresponding to Dijkstra's diffusing computation model [17]. These algorithms can efficiently detect termination without requiring a-priori knowledge of the mapping of processes to physical processors, without underlying assumptions about the message transit time or out-of-sequence delivery, and without any global clock or time reference. As described below, the *Credit Algorithm* [1] relies on the distribution of a unit value from the root process that can only be reconstituted if the barrier is complete. On the other hand, the *Tiered Algorithm* [12, 14, 19] relies on an invariant of equality between process production and consumption at all levels of process nesting. While trivial examples can indicate the relative behaviors of these two algorithms for specific task creation scenarios, it is more useful to study their performance for a wide distribution of scenarios. For instance, the Credit Algorithm can return less data to the controller PE in terms of the length of synchronization messages, however it may do so more frequently. Furthermore, the data returned by the Credit Algorithm can require more processing at the controller, thus increasing the controller's workload to detect termination. Given the computing resources of the controller and its workload, the time spent in detecting termination may vary more or less in which case trivial cases do not indicate the extent of this variation. While

the answers to these questions are difficult to obtain with theoretical examples, they can be more readily obtained if experimental approaches are used. These results can form the basis of a fair assessment of the performance tradeoffs associated with each algorithm.

II. THE CREDIT ALGORITHM

The Credit Algorithm is a global quiescence detection algorithm based on a straightforward principle of credit distribution [1]. Before computation begins, the controller is assigned to a dedicated or non-dedicated PE. The controller distributed a fixed amount of *credit*, namely a value of 1.0, i.e. 2^0 , to be shared by all PEs which work on a given task. PEs can share their credit when spawning new sub-processes. However, they are not allowed to create more credit, but only to divide their share. As each PE becomes idle, its share of the credit is returned to the controller. Since the latter knows the total amount of credit initially distributed, quiescence can be detected by noting when the sum of the returned credit is equal to the initial value of 1.0. To ensure proper credit distribution, the algorithm complies with the following rules:

- (i) When a process becomes passive, it transmits its credit share to the controller.
- (ii) When an activating message with credit share C arrives to an active process, C is transmitted to the controller.
- (iii) When an activation message with credit share C arrives to a passive process, C is transferred to the activated process.
- (iv) 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.

Based on these rules, the following properties remain valid during the execution of the algorithm:

- (i) The sum of all credits at all PEs of the network, in transit through the network, and held by the controller is always 1.0
- (ii) When a process is active, it holds a credit share $C > 0$.
- (iii) A message, which spawns a process at a destination PE, always holds a credit share $C > 0$.

By observing that the rules actually generate as many synchronization messages as activation messages, the author of the Credit Algorithm suggests several possible optimizations of the algorithmic rules shown above. One such optimization improves rule (ii) by creating a set of credits at each PE where the newly received credit is added. Members of the credit set held by a PE with the same value are combined. When the PE creates a

spawning message, a member is extracted from the credit set and sent with the spawning message granted that the resulting credit set is not empty. If the credit set contains a single member when the spawning message is created, the member is split as before. It can be verified that, in spite of this optimization of rule (ii), the three properties of the algorithm shown above remain valid. Motivated by implementation concerns, the author points out that representing credit share with floating point values is not suitable due to the possibility of rounding errors. Instead, he suggests using the negative of base 2 logarithm. In this case, a process generating a spawning message simply increments its local credit by a count of +1 and sends it with the spawning message. On the other hand, the controller keeps track of the returning credit as follows:

$Credit = Credit - 2^{-Cr}$ where $Credit$ is initialized to 1 and Cr is the credit returned from an idle PE. When $Credit$ reaches 0, the barrier has been reached. While the controller conceptually uses the previous equation to detect quiescence, precision limitation prevents its actual use. Instead, the author suggests maintaining a list of integers at the controller, called *DEBTS*, which represent the amount of outstanding credit yet to be returned from the PE participating in the barrier. To update *DEBTS* with a credit Cr returned from a recently idle PE, the controller performs the following algorithm:

```

K = Cr
while K ∉ DEBTS
begin
    DEBTS = DEBTS ∪ {K};
    K = K - 1;
end
DEBTS = DEBTS - {K};
if DEBTS = ∅ then quiescence;

```

Figure 1. Credit Algorithm of the controller.

The algorithm shown in Figure 1 performs a binary subtraction of the returned credit from the quantity of credit known to be outstanding in the network. When all credits have been received and subtractions have been completed, the set is empty and the barrier is reached. Although not mentioned in the original paper, the Credit Algorithm can easily support multiple barriers by attaching a barrier ID to each credit share.

III. THE TIERED DETECTION ALGORITHM

As with the Credit Algorithm, the Tiered Algorithm [12, 14, 19] is a processor-centered algorithm where each PE reports activities to a central controller. In the Tiered Algorithm, every participating PE reports the numbers of locally consumed and produced tasks at *each level* of process nesting to the controller whenever it becomes

idle. This process hierarchy identifier is required to be transmitted atomically along with the consumption and production counts. After all PEs have turned idle and completed their reporting, the controller determines whether the global consumption count and production count at each nesting level match. If they do, the controller announces global termination. Otherwise, it waits for the next round of checking. The controller maintains a data structure that counts the difference between the numbers of consumed and produced tasks at each level of process nesting. No difference indicates that all spawned tasks are consumed, hence no tasks are in transit and the global termination is reached.

A. PE Operation

Each PE maintains a command queue of processes to be executed. A process is entered into the queue by a process spawn message received from the controller or a process operating on this or another PE. Associated with each process spawn message is a number indicating the level of the process that created the spawn message. The level number to be associated with the newly spawned process is obtained by adding a value of one to the number of the requesting process. Figure 2 shows the algorithm for the local PE.

The consumption count represents the number of tasks that are consumed for any specific level at this local PE. Likewise, the production count represents the number of tasks spawned for any specific level at the local PE. The production count exploits a unique relationship by which the tasks dispatched by the k^{th} level are also the tasks created on the $(k + 1)^{th}$ level as shown in Figure 3(a). Since the equality or inequality of the number of tasks spawned to 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 communicated is reduced. Hence, a one-dimension table, shown in Figure 3(b), is maintained for the difference between the local consumption and production counts at each level of process nesting.

Whenever a creation message is received by a PE, the procedure *Receive_TaskSpawn_Message* is called to update the local activity table in accordance with the level number accompanying the creation message. This is equivalent to incrementing the level number in the corresponding table cell by one. 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 that is associated with the finished task. This update consists of decrementing the number in the corresponding table cell by one. The consumption and production counts are kept current by the execution of the procedures *Receive_TaskSpawn_Message* and *Finish_A_Task*. After the PE finishes all the tasks in its

execution queue and turns idle, the procedure *Upon_Idle* is called to report the difference between the numbers of consumed and produced tasks for each level to the controller. To reduce network traffic, only levels with nonzero value are reported. This technique allows a PE to be reactivated by a process spawning message that is received after the PE has become idle.

B. Operation of the Controller

The order of terminate messages received by the controller can not be assumed to be deterministic, so the controller maintains a *ledger table* to keep track of the global consumption and production counts using the information reported by all PEs. Using the same rationale as for the activity table for a PE, a one-dimension table serves as a ledger table where only the difference between the consumption and production counts for each level is maintained. Figure 4 shows the algorithm for the controller.

Whenever a PE reports to the controller, the controller calls the procedure *Receive_Report* to respond. 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 number stored in the corresponding level cell of the ledger table by the amount reported. Next, it checks the ledger table. If the difference values in all cells of the ledger table are null, meaning all tasks spawned 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, the controller exits the procedure and waits for the next report.

IV. TRACE OF THE TIERED AND CREDIT ALGORITHMS

The Credit and Tiered Algorithm rely both on the use of a global invariant whose properties allow the algorithms to detect the barrier without explicitly determining the state of each PE as shown in Table 1.

The trace of both algorithms is illustrated by showing the processing required when a synchronization message is received. Both algorithms maintain a list data structure at their respective controllers that is used to detect the barrier. In the Credit Algorithm, this list represents the amount of outstanding credit to be returned to the controller while in the Tiered Algorithm this list contains a single entry for every level of processing that takes place in the network. To show the operation of both algorithms relative to their controller list data structures, consider the simple example shown in Figure 5. It shows a simple program initiated by the controller, denoted by C , which creates two tasks, namely T_0 and T_1 , executing on two PEs, namely PE_0 and PE_1 .

```

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 2. PE operation in the Tiered detection algorithm.

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

(a) Theoretical table.

DIFF(1)
DIFF(2)
•
•
•
•
•
DIFF(D-1)
DIFF(D)

(b) Implementation table.

Figure 3. Activity table.

```

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 4. Operation of the controller in the Tiered detection algorithm.

Algorithm	Invariant	Properties of the Invariant
Tiered	Number of tasks spawned and consumed at each level of the task hierarchy.	At the end of processing of each level, the total spawned at that level equals the total consumed.
Credit	Total credit distributed throughout the network and controller.	Sum of all credit is fixed and known by the controller.

Table 1. Global nvariant used by the Tiered and Credit algorithms.

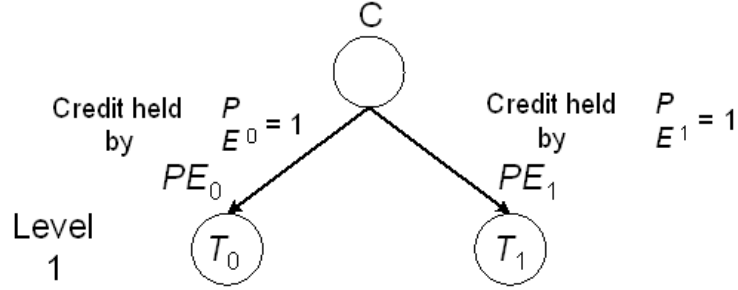


Figure 5. Credit vs. Tiered Algorithm example.

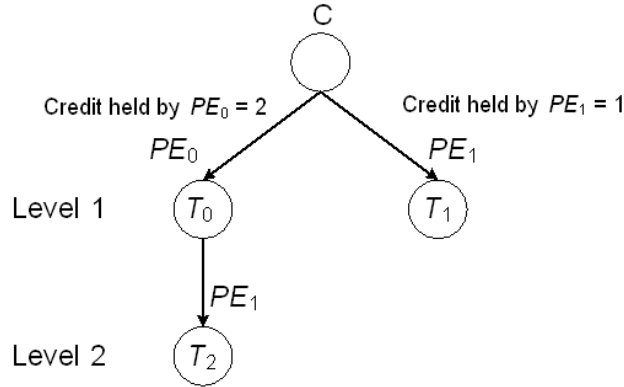


Figure 6. Task T_0 spawns T_2 on PE_1 .

Initially, the list data structures maintained by both algorithms are: Credit: $\{0\}$, Tiered: $\{2\}$. At the start of the Credit Algorithm, the list is initialized to 0 thus representing the initial total outstanding credit value of 2^0 . For the Tiered Algorithm, the list is initialized to show that the controller has generated two level-1 tasks. The value of 2 is associated with level 1 by its position in the list. The credit value assigned to the two tasks by the controller in the Credit Algorithm is 1. This satisfies the invariance requirement that the total credit, both returned and distributed, equals the amount originally created, namely $2^{-1} + 2^{-1} = 1$. In this example, the first significant event occurs when T_1 operating on PE_0 creates a task spawning message which enqueues a new task T_2 on PE_1 as shown in Figure 6. PE_0 splits its locally held credit while attaching half of the result to the spawning message. Since PE_1 is already active when the spawning message is received, a synchronization message is created returning the credit associated with the spawn message to the controller. For the Tiered Algorithm, no synchronization message is generated. As a result of the credit returned by PE_1 , the controller of the Credit Algorithm updates the list accordingly, namely Credit: $\{1, 2\}$. The Credit list shows a total outstanding credit of $2^{-1} + 2^{-2} = 0.75$. If the next significant event to occur is the completion of task T_0 , PE_0 returns a task termination

message to the controller. For the Credit Algorithm, the resulting controller data list contains Credit: $\{1\}$. The message returned for the Tiered Algorithm notifies the controller that one level-1 task has been consumed and one level-2 task has been created. After this notification, the Tiered list becomes Tiered: $\{1, 1\}$. When PE_1 completes both task T_1 and T_2 , it composes a synchronization message for the Credit controller which includes the credit held by the PE. For the Tiered controller, the PE creates a message which signals that two tasks have been completed, one at each level. After this event has been signaled and processed by the controller, the two lists become Credit: $\{\}$, Tiered: $\{0, 0\}$. The Credit controller detects the barrier when the controller data structure list becomes empty while the Tiered controller detects the barrier when every element of the data structure maintained by the controller becomes zero.

V. EVALUATION OF THE TIERED AND CREDIT ALGORITHM

A close examination of the trace of both algorithms may reveal that while the Credit Algorithm returns less data to the controller during signaling, it nevertheless returns it more often.

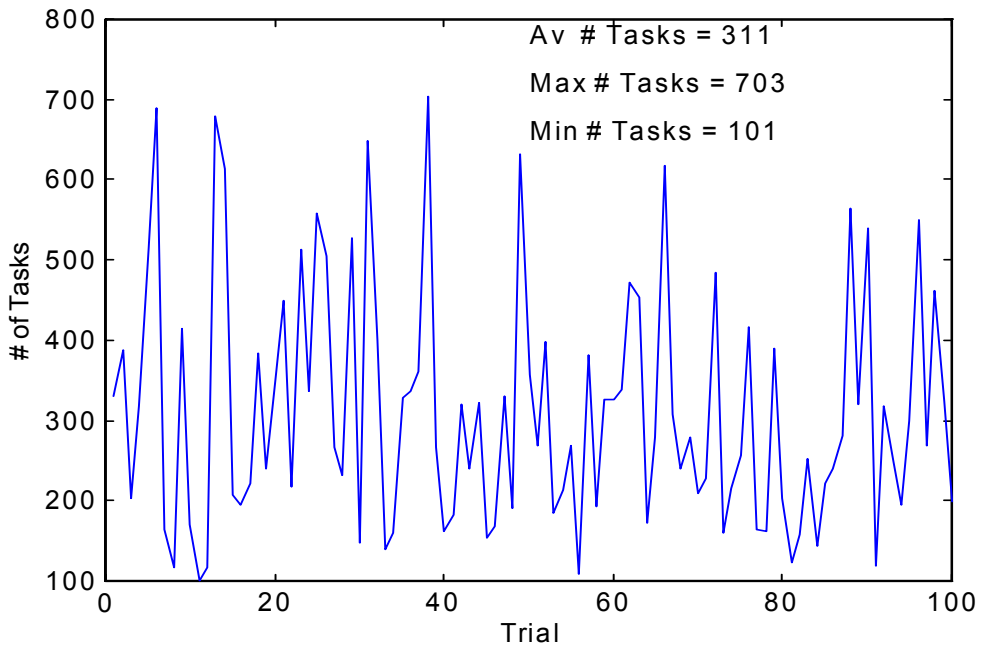


Figure 7. Size of the application used in the simulation experiment.

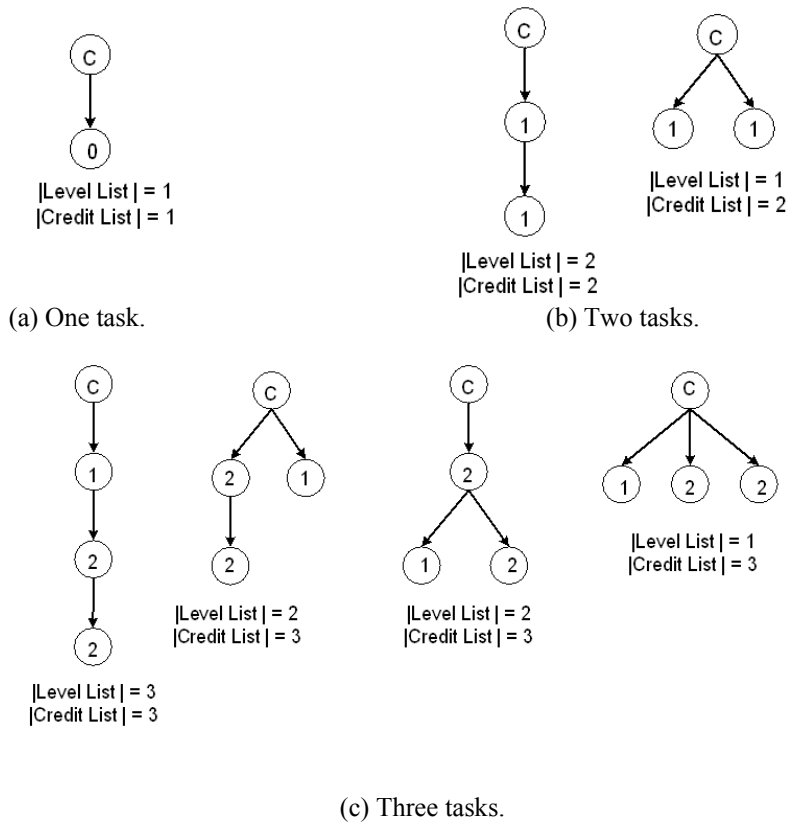


Figure 8. Size of the controller data structure in a small application.

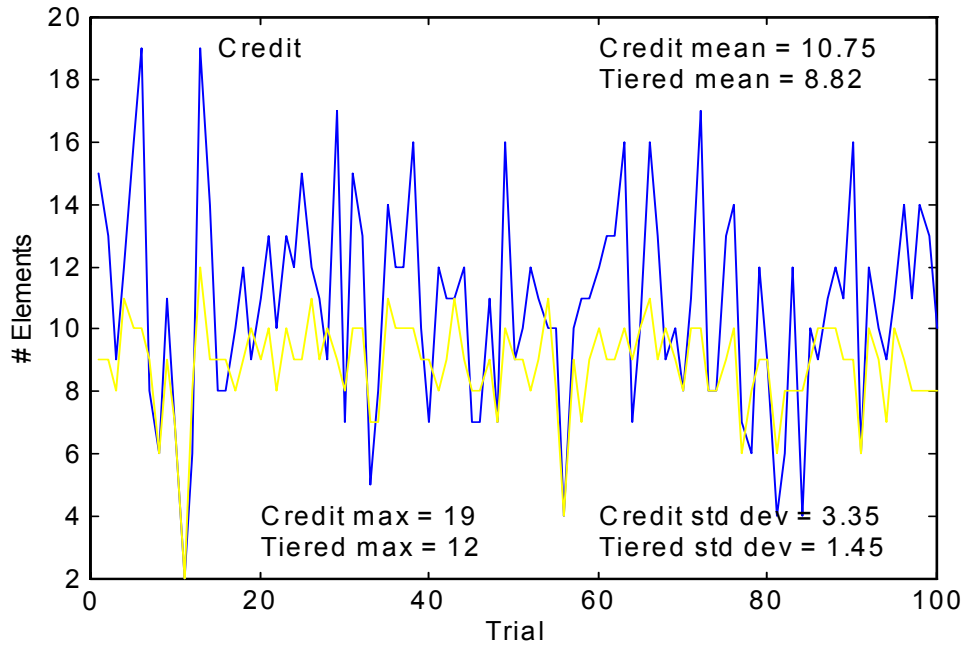


Figure 9. Maximum size of the controller data structure obtained with simulation.

In addition, the returned data requires more processing operations in order to incorporate the returned credit into the data structure maintained by the controller. The experiment models a network consisting of 100 PEs executing a number of randomly generated tasks hierarchies. Figure 7 shows the application size in terms of the number of tasks generated across the 100 test cases.

A. Size of the Controller Data Structure

In the Tiered Algorithm, there is a one-to-one correspondence between the number of elements in the data structure kept in the controller and the number of task levels created for an application. However, in the Credit Algorithm, the size of the controller data structure is upper-bounded by the maximum credit value that is created during the execution of the application while it is limited from below by the number of levels in the task hierarchy. Figure 8 shows small applications consisting of one, two, and three tasks. While the size of the controller data structure can be easily determined for small applications, it is not quite clear how it evolves in larger applications. To this end, a simulation of 100 random generated task hierarchies is performed while monitoring the maximum size of the controller data structure in both algorithms as shown in Figure 9. The credit curve in the figure represents the largest size of the data structure throughout the 100 trials of the simulation. It is clear from the figure that the size of the list in the Credit Algorithm changes across a wide range of values throughout the course of the controller operation.

B. Controller Workload

To evaluate the workload required of the controller in both algorithms, a simulation of 100 runs is performed to monitor the number of operations performed by the controller. In this evaluation, the simulation modules are carefully implemented to perform only scalar and low-level operations which could be expected in a typical implementation of a controller. Figure 10 shows the results of the simulation. Because the Credit Algorithm relies on compound operations, such as the combining of members in the credit set held by a PE, it tends to generate a significantly larger workload for the controller than the Tiered Algorithm. In addition, the variation in workload imposed on the controller by the Credit Algorithm is greater than the one imposed by the Tiered Algorithm. It seems that the Credit Algorithm would require a controller with a larger computation capacity since, in fact, the computational resources of the controller must be sized proportionately in order to properly handle the largest anticipated workload.

C. Detection Latency

To evaluate detection latency in both algorithms, a simulation of 100 runs is performed to monitor the number of operations performed by the controller to detect latency after it receives a synchronization message from the last active PE in the network. Figure 11 shows the result of this simulation.

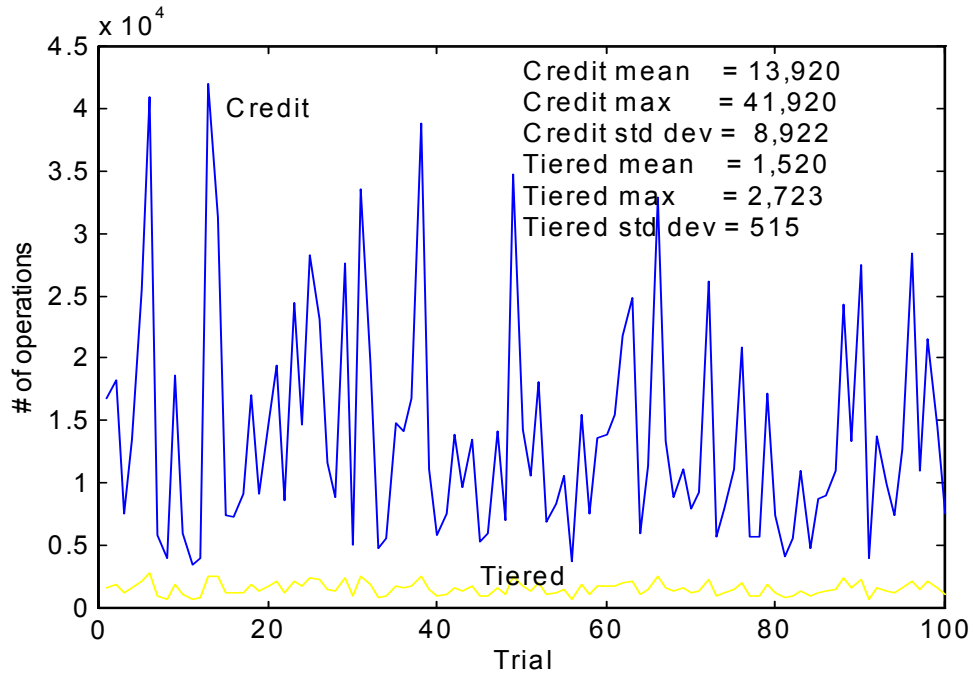


Figure 10. Number of controller operations obtained with simulation.

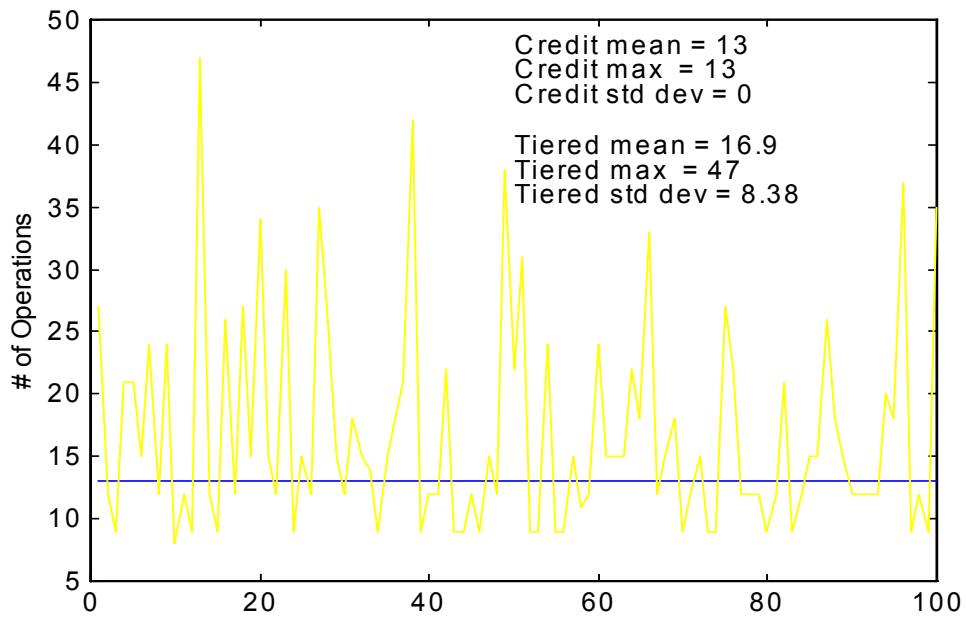


Figure 11. Number of controller operations to detect latency.

The figure shows that the controller in the Credit Algorithm performs a constant number of operations, namely 13 operations, to detect latency. This constant value indicates that the controller data structure contains

only a single element at the time the controller receives the last message from the last active PE.

D. Volume of Synchronization Messages

The volume of a synchronization message can be quantified using three metrics:

- (i) The number of *strings* where a string is a collection of data elements returned to the controller by a given PE during synchronization. These strings are returned when a PE becomes idle in both algorithms.
- (ii) The number of *elements* returned in the strings where each returned element increments the count by 1.
- (iii) The number of bits required to represent each string element based on the size of the element.

Based on these metrics, simulations of 100 trials are performed by which Figure 12, 13, and 14 show the number of returned strings, elements, and bits respectively to the controller of each algorithm throughout the trials.

Figure 12 shows two curves where the first one represents the number of strings returned to the controller in the Credit Algorithm while the second one represents the difference between the number of strings returned to the controllers in the Credit and the Tiered Algorithm. A positive difference in the latter curve indicates an advantage for the Tiered Algorithm. This figure clearly shows that the Credit Algorithm produces a higher number of strings since it may return received credits at the same time with task spawning messages. In order to clearly contrast the performance of the two algorithms, the second curve in Figure 13 and 14 represents the difference between the number of returned elements or bits by the Credit and the Tiered Algorithm. The difference curve in Figure 13 shows that the Tiered Algorithm returns a larger number of elements than the

Credit Algorithm does. However, the difference curve in Figure 14 shows that the Tiered Algorithm returns fewer bits than the Credit Algorithm does. In the case of the Tiered Algorithm, the maximum value of the element returned to the controller typically follows the maximum number of task levels created. Note that the maximum task level establishes a lower limit on the maximum size of the credit list. Therefore, while the Tiered Algorithm may return more elements to the controller, the small values of these elements allow them to be encoded with fewer bits. It should be noted that the smallest word size of a message used in an implementation of these two algorithms may eliminate any advantage for the Tiered Algorithm as suggested in Figure 14.

VI. CONCLUSION

Figure 15 through 20 summarize the results of the comparison of the Tiered Algorithm to the Credit Algorithm in terms of the size of the data structure used by the controller, the workload required of the controller, detection latency, and the volume of the synchronization message. A lower value of any of the parameters indicates an advantage in each figure. The simulation experiments show that the Tiered Algorithm outperforms the Credit Algorithm in terms of the size of the data structure handled by the controller and the controller workload. On the other hand, the Credit Algorithm outperforms the Tiered Algorithm in detection latency operations. However, the complexity of operations and number of clock cycles required to execute them is also different. Given an application, the Tiered Algorithm can be used if the workload of the controller is a primary concern. However, if very predictable latency is desired, the Credit Algorithm is the best choice.

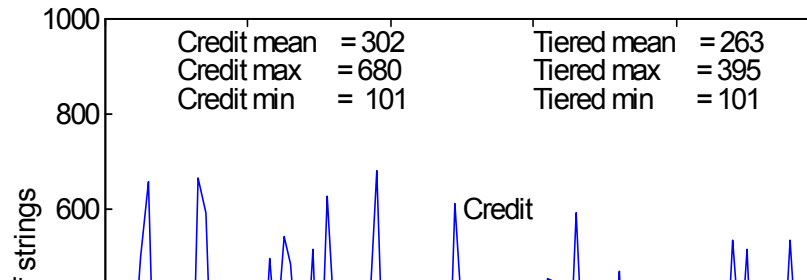


Figure 12. Number of strings over 100 trials.

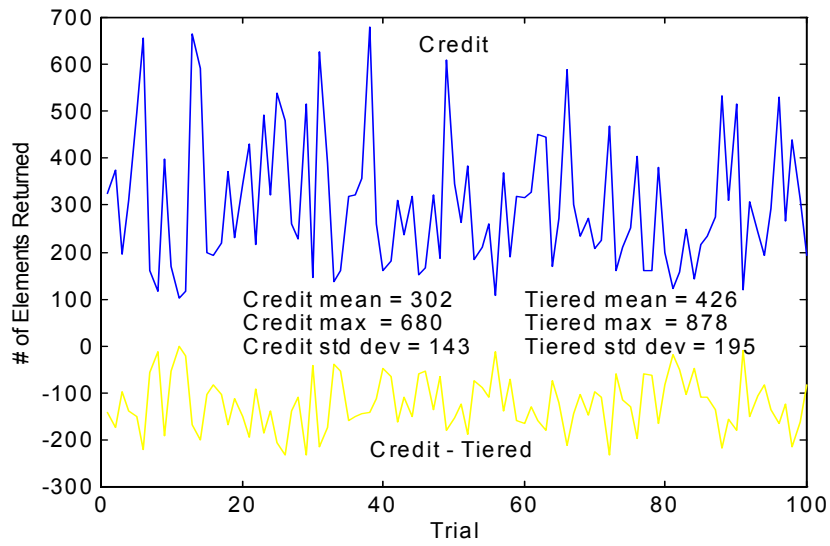


Figure 13. Number of elements over 100 trials.

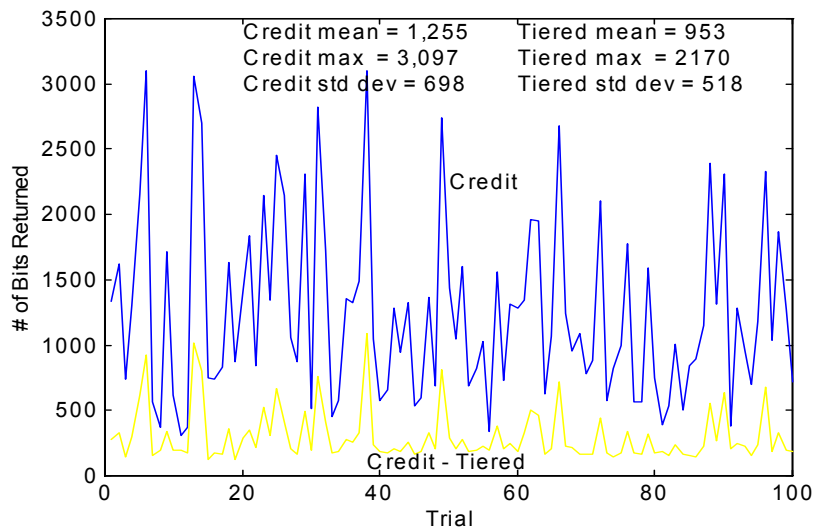


Figure 14. Number of bits over 100 trials.

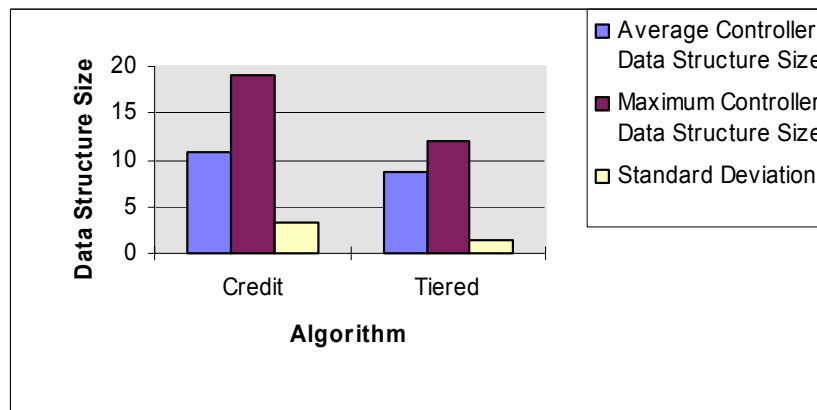


Figure 15. Size of the controller data structure.

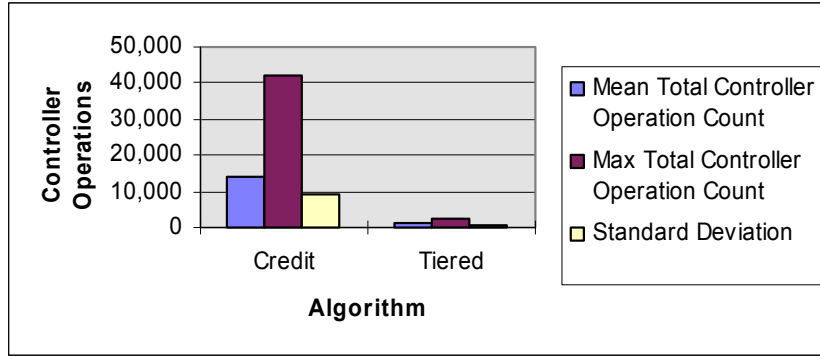


Figure 16. Number of controller operations.

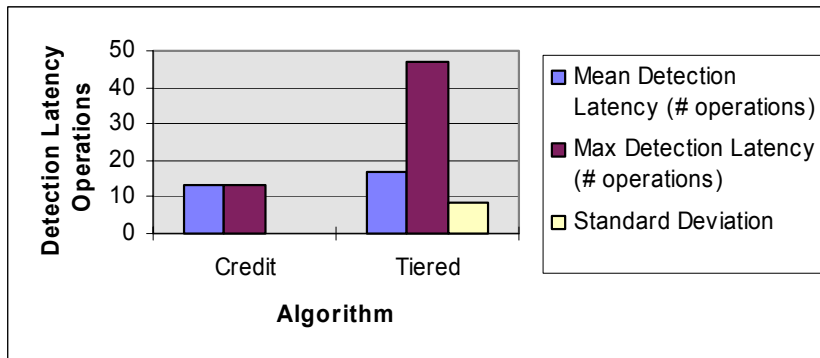


Figure 17. Detection latency of the synchronization barrier.

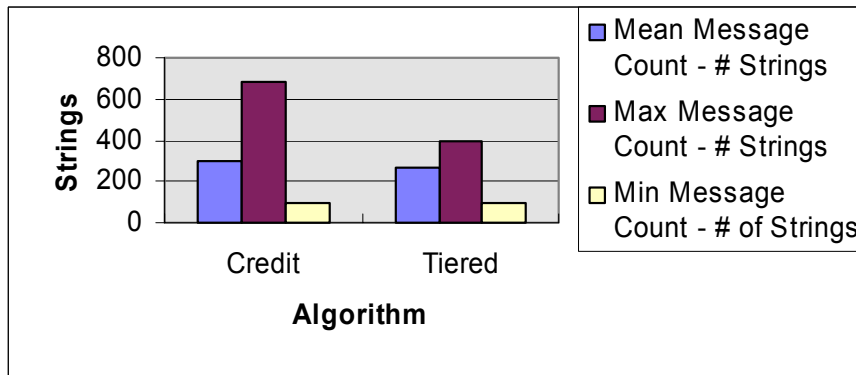


Figure 18. Number of synchronization message strings.

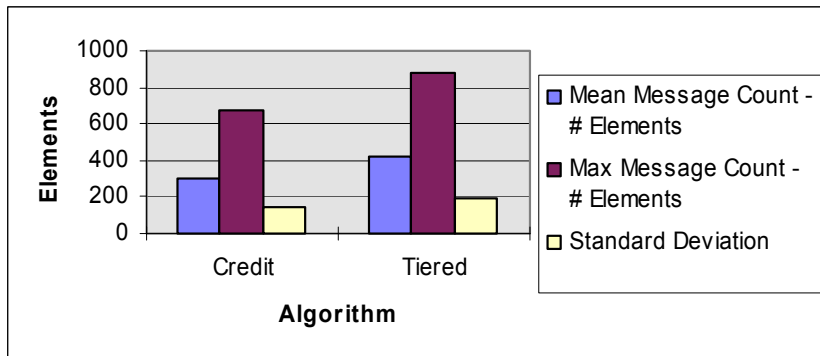


Figure 19. Number of synchronization message elements.

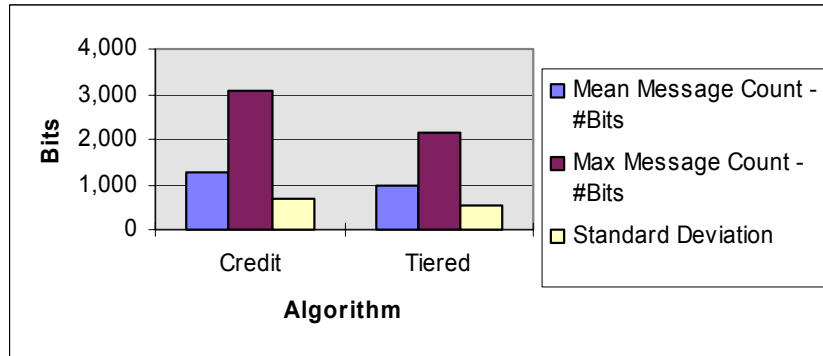


Figure 20. Number of synchronization message bits.

REFERENCES

- [1] F. Mattern, "Global quiescence detection based on credit distribution and discovery," *Information Processing Letters*, vol. 30, no. 4, pp. 195-200, Feb. 1989.
- [2] Y.-C. Tseng and C.-C. Tan, "Termination detection protocols for mobile distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 558-566, Jun. 2001.
- [3] M. P. Wellman and W. E. Walsh, "Distributed quiescence detection in multiagent negotiation," *4th International Conference on Multiagent Systems*, 2000, pp. 317-324.
- [4] E. D. Brooks III, "The butterfly barrier," *International Journal of Parallel Programming*, vol. 15, no. 14, pp. 295-307, 1986.
- [5] N. S. Arenstorf and H. F. Jordan, "Comparing barrier algorithms," *Parallel Computing*, vol. 12, pp. 157-170, 1989.
- [6] 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.
- [7] 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.
- [8] J.-S. Yang and C.-T. King, "Designing tree-based barrier synchronization on 2D meshes networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 6, pp. 526-533, Jun. 1998.
- [9] 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.
- [10] Y. Sun, P. Y. S. Cheung, and X. Lin, "Barrier synchronization on wormhole-routed networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 583-597, Jun. 2001.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] R. DeMara, B. Motlagh, C. Lin, and S. Kuo, "Barrier synchronization techniques for distributed process creation," *International Parallel Processing Symposium*, Apr. 1994, pp. 597-603.
- [15] T.-H. Lai, Y.-C. Tseng, and X. Dong, "A more efficient message-optimal algorithm for distributed termination detection," *Sixth International Parallel Processing Symposium*, 1992, pp. 646-649.
- [16] 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.
- [17] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1-4, Aug. 1980.
- [18] 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.
- [19] R. DeMara, Y. Tseng, and A. Ejnoui, "Tiered algorithm for distributed process quiescence and termination detection," University of Central Florida, Orlando, Florida, UCF-ECE-0402, 2004, available at <http://netmoc.cpe.ucf.edu:8080/internal/yearReports.jsp?year=2004>.

This document is an author-formatted work. The definitive version for citation appears as:

R. F. DeMara, K. Drake, and A. Ejnoui, "Performance Evaluation of Hierarchical Annotation and Credit Distribution Quiescence Detection Mechanisms," submitted to *Distributed Computing* on November 20, 2004 and available as UCF Technical Report UCF-ECE-0405 online at

<http://netmoc.cpe.ucf.edu:8080/internal/yearReportsDetail.jsp?year=2004&id=0405>

This work has been submitted to the *Distributed Computing* for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible
