A dynamic and adaptive quiescence detection algorithm

Amitabh B. Sinha Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
email: {sinha,kale}@cs.uiuc.edu

Balkrishna Ramkumar
Department of Electrical and Computer Engineering
University of Iowa
Iowa City, Iowa 52242
email: ramkumar@hitchcock.eng.uiowa.edu

Abstract

A large number of quiescence detection algorithms with good theoretical upper-bounds have been proposed before. However the metric used to measure the performance of these algorithms is not suitable. We propose a new metric for measuring the performance of a quiescence detection algorithm. We also present an algorithm to detect quiescence in an asynchronous and dynamic model of parallel computation. The algorithm has been implemented for a machine independent parallel programming system, Charm. Quiescence detection is provided as a feature in Charm to conduct a variety of operations like collecting statistics about user computation, initiating new phases of computation, or just terminating the user computation.

1 Introduction

Parallel and distributed computing has sparked off the development of new algorithms in which many computing elements interacting synchronously or asynchronously attempt to solve problems. In such algorithms one key issue is to decide when the algorithm has become quiescent. Quiescence detection has been extensively studied; the computational models used have been either synchronous[1] or asynchronous[2]. In this paper, we deal with the problem of detecting quiescence in an asynchronous model of computation.

Our model of execution is distributed and asynchronous. Initially only one process, the *main* process, exists. Further activity is generated by creating *activation* messages. An *activation* message can be either a message to create a new process (*creation* message) or a message to an existing process (*response* message). A process is said to be *active* when it is processing an activation message addressed to it. At all other times a process is *passive*. An *active* process can generate new activation messages during its execution. A *passive* process can be transformed into an *active* process only by an activation message addressed to it. Activation messages are not generated spontaneously, i.e., activation messages cannot be created on a processor where all processes are passive.

In addition to activation messages, which are generated by the application program in a dynamic fashion, the system uses *control* messages which are interleaved with the activation messages. *Control* messages can be used by the system and cannot generate user activity, i.e., control messages can neither activate passive user processes nor create new activation messages. Control messages are used for system related activities like dynamic load balancing and the detection of system properties, such as quiescence.

In our model of parallel execution, we define a user computation to be *quiescent*, when the following conditions are met simultaneously:

1. All processes are passive.
2. There are no activation messages in the system, i.e., there are no activation messages in transit or buffered in system queues. This does not preclude the existence of control messages in the system.

Quiescence detection, then, is the process by which the system detects quiescence in the user computation. The problem of quiescence detection in distributed systems becomes interesting because the two conditions for quiescence must be met *simultaneously*.

Much work has been done before on quiescence detection [1, 2, 3, 4, 5, 6, 7, 8], both for synchronous and asynchronous systems. We shall briefly discuss some previous work on quiescence detection in asynchronous computational models.

Lai [8] and Huang [7] present schemes which use distributed snapshots to detect quiescence in asynchronous distributed systems. In Lai's approach, a predefined process combines local snapshots (taken spontaneously by processes) into a global snapshot, which it then uses to determine if quiescence has occurred. Huang's method is essentially similar, the only difference being that any processor can initiate the collection of the global snapshot. Lai and Huang use different techniques to ensure that the global snapshot is *feasible*, i.e., it does not contain processing of messages if the corresponding creation is not also part of the snapshot. One drawback of both their schemes is that they do not extend to systems where processes can be created dynamically. In addition, Huang's scheme can become very expensive because each processor can initiate a

[1]A *send* event in a process is synchronized with the corresponding *receive* event in another process, e.g., in CSP a send blocks till the corresponding receive is executed, and vice versa.

[2]A *send* event in a process can happen without the corresponding *receive* event being executed on another process. However, the receive event does not occur unless the corresponding send event has occurred.

global snapshot if it is idle. In the worst case, when all processes go idle everyone will initiate a broadcast. Lai's scheme does not suffer from this drawback, however the lack of coordination between the collection of local snapshots means that a considerable number of global snapshots may be collected.

Mattern[6] presents an elegant credit based scheme to detect quiescence in dynamic, asynchronous, distributed systems. In the worst case, the number of control messages needed by Mattern's algorithm is the number of activation messages. The scheme works by distributing one unit of credit amongst active processes and activation messages. When a process creates an activation message, it divides its credits equally between itself and the message. A process returns its credits to the monitoring process (the quiescence detection algorithm) when it becomes passive. An activation messages' credit is passed on to its receiving process if the receiving process is passive; otherwise (if the receiving process is active) the credit is returned to the monitoring process. When the monitoring process has received the original one unit of credit, it reports quiescence.

Whenever processes split up their credits between themselves and activation messages they create, fractions are generated. Fractions cannot be accurately computed with the current representation of floating point numbers. Mattern presents an approach wherein fractions need not be explicitly computed. Notice that all fractions are of the form 2^{-n}, and hence can be represented by the negative of the logarithm (called *credits*), i.e., n. The scheme that Mattern outlines to solve the problem of having to compute fractions exactly involves computing the set of missing credits (credits possessed by activation messages, active processes and control messages at that time). The set is updated whenever credits are returned to the monitoring process. The set of missing credits is maintained by the monitoring process, which is a single process running on one processor. The cardinality of this set is bounded by the sum of the number of activation messages, active processes and control messages over the entire system. Assuming that the memory requirements of each activation message is constant, it would mean that the memory requirements of the monitoring process to maintain the set of missing credits is of the same order as that of the memory requirements of all the messages in the entire system. In addition, the monitoring process becomes a bottleneck in bigger distributed systems, since all credits are being returned to the one processor on which the monitoring process is located.

Chandy and Misra [9] have proved that given any quiescence detection algorithm for an asynchronous system in the worst case the number of control messages needed cannot be less than the number of activation messages in the user computation. Chandrasekaran and Venkatesan [10] extended this result and proved that the number of control messages needed in the worst case by any quiescence detection algorithm for asynchronous systems is also bounded below by the number of communication links. The proof, however, is incorrect. The proof relies on the fact that for the algorithm to know that a message is in transit through a link it is necessary to send a control message through that link. However, the knowledge that a message has been sent, and not yet received is sufficient to know that a message is still in transit in the communication link, and this information may be/is often available without having to send a message through that particular communication link.

In the above papers, the number of control messages used by an algorithm has been identified as an indication of the performance of a quiescence detection algorithm. However, the true metric for performance of a quiescence detection algorithm depends on two factors. First, it depends on the extent to which a quiescence detection algorithm interferes with (and slows down) the user computation. The absolute number of control messages is not an accurate characterization of this quantity, for control messages processed and/or sent by **idle** processors do not adversely affect the performance of the user computation, and are not overheads in terms of computational performance. We can identify the performance of a quiescence detection algorithm by observing the following quantities:

- the increase in the execution time of a program when it is run without and then with the quiescence detection algorithm.
- the time it takes for the algorithm to report quiescence, once the system is actually quiescent.

In Section 2, we present a two-phase distributed quiescence detection algorithm called *Counting.* Our quiescence detection algorithm is unique in two significant ways:

- Our quiescence detection algorithm does not distinguish between the two types of activation messages. Neither does it depend on knowledge about the placement of processes or how they communicate. Hence it will work in a dynamic and asynchronous model of computation.
- The algorithm automatically adapts to system loads — it generates very few control messages when the system is busy, and more control messages when the system is lightly loaded. Control messages generated in the latter case do not adversely affect system performance, because they only occupy computational resources of idling processors. In most cases the algorithm needs much fewer control messages than the lower bound presented by Chandy and Misra; in the worst case the number of messages may be higher, but these are sent/received by idle processors, thus contributing little to the overhead.

In Section 3, we describe the implementation of the algorithm for a machine independent parallel programming system called Charm. In Charm, quiescence detection is provided as a feature to conduct a variety of operations like collecting statistics about user computation, initiating new phases of computation, or just terminating the user computation. Finally, in Section 4, we summarize our work and discuss other future applications of the *Counting* algorithm.

2 The Counting algorithm

The *Counting* algorithm is a two-phase distributed algorithm; a copy of the algorithm, or *component*, runs on each processor. Figure 1 shows a model of the activity on each processor. Each processor has a collection of passive processes and a pool of messages for these processes. There is a scheduler that decides which message should be processed next, and activates the necessary process. Each component interacts with the scheduler and the user processes running on that processor. All communication between the components occur along a spanning tree covering the processors. In the description below all references to the *parent*, the *children*, the *root*, or the *sub-tree* of a processor are with respect to the corresponding entities in the spanning tree on all the processors.

We denote the first and second phases of the *Counting* algorithm as Phase 1 and Phase 2, respectively. The *Counting* algorithm use three kinds of control messages:

1. *initialization:* these are broadcast to all components, and result in the initialization of Phase 1 or Phase 2 on all the components.
2. *idle:* these messages are sent up to the parent during Phase 1. An idle message signifies that each processor in the sub-tree below has been idle at least once since the last *idle* message. It does not mean that all the processors were idle *simultaneously.*
3. *activity:* these messages are sent up to the parent during Phase 2 and contain a report of activity (creation and processing) in the sub-tree rooted at the sending processing element.

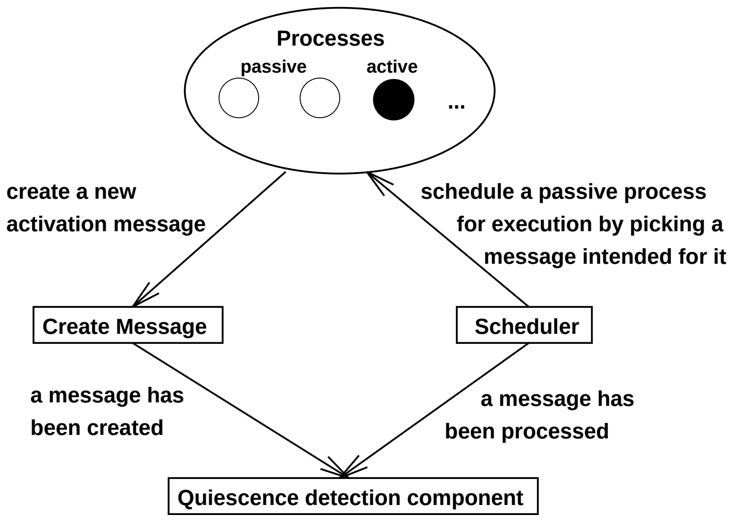
Processes
passive
active
...
create a new activation message
schedule a passive process for execution by picking a message intended for it
Create Message
Scheduler
a message has been created
a message has been processed
Quiescence detection component

Figure 1: The figure shows the interactions between a component, the processes on that processor, the scheduler and the part of the run-time system responsible for creating new messages.

We use the construct — **wait until (condn)** — in the description of our algorithm. The process executing the **wait until** is suspended till such time as the **condn** becomes true. In the *Counting* algorithm, each component maintains the following counts:

- n_c: this is the sum of the number of *activation* messages **created** on this processor.
- n_p: this is the sum of the number of *activation* messages **processed** on this processor.

Each component also has two other counts N_c and N_p — they are used to estimate the number of messages created and processed, respectively, in the sub-tree rooted at itself. These are initialized to zero at the beginning of Phase 1 and Phase 2, and are sent up with *idle* and *activity* messages.

The *Counting* algorithm appears in Figure 2. Phase 1 is called on each processor immediately before the user computation begins. Only one phase of the quiescence detection algorithm will be active at any time.

In Phase 1, each leaf component waits until its processor is idle and then sends an idle message to its parent with the counts N_c and N_p initialized to n_c and n_p, respectively. All other components wait until they receive one idle message from each child, adding the values of N_c and N_p in these idle messages to their local values. Having received idle messages from all its children, the component waits until its processor is idle, and then it sends an idle message to its parent. The idle message contains the values of the counts N_c and N_p, which have been incremented with the values of n_c and n_p, respectively, on that component. When the root has received idle messages from all its children components, it decides whether the system can be *idle* by comparing the values of N_c and N_p. If they are equal then there's a high probability (but not a certainty; see explanation of Figure 3 below) that all *activation* messages have been processed in the system. If the two counts are not equal then the root initiates Phase 1 again, otherwise the root initiates Phase 2 on all the components.

In Phase 2, the components send up their *activity* report messages containing the new values of N_c and N_p. *Activity* messages from components are combined in the same way as in the first phase of the *Counting* algorithm. When the root component has received one activity message from each of its children, it compares the old and the new values of N_c and N_p. If these values are the same it implies that there has been no new activity in the system, and the root reports *quiescence*; otherwise the root initiates Phase 1

```
Phase 1()
{
    $N_c = 0$;  $N_p = 0$;
    wait until (RecdMsgsFromChildren()); /* wait until messages have */
                                         /* been received from all children */
    add to local $N_c$ and $N_p$ the values recd.  from children;
    wait until (Idle()); /* wait until this processor has no activation messages */
    $N_c = N_c + n_c$;  $N_p = N_p + n_p$;
    if (RootSpanTree()) /* check if this processor is the root of the spanning tree */
        if ($N_c \neq N_p$)
            Broadcast message to begin Phase 1
        else
            $N^{old} = N_c$ /* $N_c == N_p$ */
            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()); /* wait until messages have */
                                         /* been received from all children */
    add to local $N_c$ and $N_p$ the values recd.  from children;
    wait until (Idle());
    $N_c = N_c + n_c$;  $N_p = N_p + n_p$;
    if (RootSpanTree()) /* check if this processor is the root of the spanning tree */
        if ($N^{old} == N_c$)
            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: The *Counting* Algorithm

again.

Note that a single phase is not sufficient to guarantee that the system was quiescent, because the counts N_c and N_p might match at the end of Phase 1 even though all messages were not processed. Figure 3 shows the 'wave' of *idle* messages being passed up the spanning tree in Phase 1 — the processors below the wave have already sent out their *idle* messages, while the processors above the wave haven't yet sent out their *idle* messages. Consider the following scenario in Figure 3: message m_1 is created on a processor above the wave of Phase 1 messages and sent to a processor below it, while message m_2 is created on a processor below the wave of Phase 1 messages and sent to a processor above it. Further, assume that the processing of m_2 did not create new activation messages. In this case, when the wave reaches the root, the following is true:

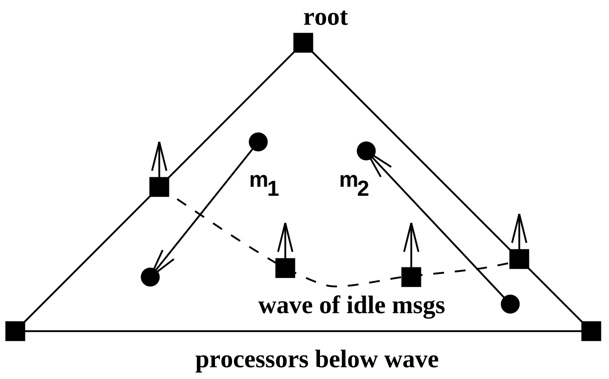

Figure 3: The Wave of idle messages in Phase 1

- Creation of m_1 is counted.
- Processing of m_1 is not counted.
- Creation of m_2 is not counted.
- Processing of m_2 is counted.

At the end of Phase 1, the counts may[3] match even though m_1 was processed after the idle message was sent. The processing of message m_1 could have generated more new activity, and therefore it is incorrect to infer from the counts matching at the end of Phase 1 that the system is quiescent.

3 Quiescence Detection in Charm

We have implemented the *Counting* algorithm discussed above for Charm [11]. Charm is a machine independent parallel programming language. Programs written in Charm run unchanged on shared memory machines including Encore Multimax and Sequent Symmetry, nonshared memory machines including Intel iPSC/860 and NCUBE/2, UNIX based networks of workstations including a network of IBM RISC workstations, and any UNIX based uniprocessor machine.

The basic unit of computation in Charm is a *chare*. A chare is created dynamically, and has associated with it a data area, and a set of entry functions that can access this data area. The entry functions can be executed by addressing a message to an entry function of a particular chare instance, which can be uniquely identified with its *identifier*.

In terms of our execution model, chares are processes, and messages to create new chares or to send messages to existing chares are activation messages. In the shared memory implementation of Charm, activation messages are delivered by enqueuing them directly into shared queues. On nonshared memory machines, messages are delivered across processors using the native communication primitives available on the machine. After delivery, messages are queued up in local queues. In both the shared and the distributed memory machine implementations, a scheduler picks up messages from the queue and either creates a new chare, or activates the chare to which it is addressed.

There is another type of process in Charm called a *branch office chare*. A branch office chare is similar to a chare, except that an identical version (branch) of the chare executes on each processor. In the quiescence

[3]If we assume that no other activity is occurring in the system then the counts will match; however if there is other activity in the system the counts may still match because there are more than one pair such as m_1 and m_2.

detection algorithm, a message to create a branch office chare is treated as p messages to create chares, where p is the number of processors being used. The *Counting* algorithm itself has been implemented as a branch office chare — a branch corresponds to a component.

The quiescence detection feature in Charm has been used in the implementation of a wide variety of real-life applications including parallel algorithms for logic synthesis [12] and for test pattern generation of sequential circuits[13]. In order to measure the performance of the quiescence detection algorithm in varying program contexts, we tested its performance for the following four synthetic benchmark problems[4] on a nonshared memory machine:

1. Problem A is a parallel divide and conquer application. Computation starts with an initial problem, which is recursively divided to create sub-problems which are executed in parallel. The solutions from sub-problems are then combined. In the initial phases of the computation when sub-problems are being created there isn't enough work for all processors. The situation is similar at the very end when solutions are being combined and sub-problems finish executing.

2. Problem B is a multi-phase application, where each phase is itself a divide and conquer application. There are six phases of the divide and conquer application. Parallelism and processor utilization varies between near-idleness to total-utilization several times before termination. Thus, Problem B is good benchmark to test the efficacy and correctness of the quiescence detection algorithm.

3. The task graph for Problem C is a finite length chain of processes with the property that each process in the chain is created by the process preceding it in the chain (the first process in the chain is created by the main process), and only one process in the chain is active at any instant of time. Each new process is created on a randomly chosen processor.

4. In Problem D, each processor has one process, and the processes communicate along a directed cycle on the processes. The computation consists of a pre-determined number of iterations of sends and receives. In an iteration each process sends a message to the next process in the cycle, and receives a message from the previous process in the cycle.

Table 1 shows the performance results of the *Counting* quiescence detection algorithm for program runs on the NCUBE/2, a nonshared memory machine. A comparison of the number of control messages needed to detect quiescence with the number of activation messages in that execution run shows that in all cases, except Problem C, the number of control messages used are substantially lower than the number of activation messages. Problem C is a 'hard' problem for the *Counting* algorithm, for there is only one active process on one processor at any time, and all other processors are idle. The number of control messages generated and processed in Problem C occupy computational resources of idle processors, and they shouldn't be considered as an indication of the overhead of the quiescence detection algorithm. This claim is substantiated by the results in Table 2. For Problem C, the number of control messages used by the quiescence detection algorithm is substantially more than the number of activation messages (in most cases); however the average overhead of the quiescence detection algorithm for Problem C is only about 12%.

Table 2 shows the execution times for Problems A, B, C and D with and without the (Counting) quiescence detection algorithm on a nonshared memory machine, NCUBE/2. For problems A and B, the application performs better in some cases with the quiescence detection algorithm running than without it. This is not as surprising as it may seem, because Charm provides dynamic load balancing strategies whose

[4]We do not need to use a quiescence detection algorithm to detect quiescence for any one of the benchmark problems; the problems are only used as a controlled experiment, where the onset of quiescence can be independently ascertained.

#PE	**Control Msgs**	**Iterations Phase1+Phase2**	**Time (ms)**
2	20	8+2	5
4	40	9+1	6
8	88	10+1	7
16	144	8+1	11
32	256	7+1	16
64	384	5+1	25
128	996	6+1	41
256	1992	6+1	73

Problem A
Activation Msgs: 6389

#PE	**Control Msgs**	**Iterations Phase1+Phase2**	**Time (ms)**
2	90	44+1	8
4	172	42+1	16
8	264	32+1	20
16	416	25+1	62
32	608	17+2	23
64	1024	15+1	30
128	2304	16+2	23
256	3328	12+1	29

Problem B
Activation Msgs: 9036

#PE	**Control Msgs**	**Iterations Phase1+Phase2**	**Time (ms)**
2	26	12+1	14
4	92	21+1	15
8	232	28+1	17
16	496	30+1	28
32	960	29+1	23
64	2112	32+1	20
128	2688	20+1	29
256	4096	15+1	45

Problem C
Activation Msgs: 42

#PE	**Control Msgs**	**Iterations Phase1+Phase2**	**Time (ms)**
2	6	2+1	15
4	12	2+1	16
8	24	2+1	18
16	48	2+1	24
32	96	2+1	32
64	192	2+1	30
128	384	2+1	29
256	768	2+1	30

Problem D
Activation Msgs: 2000/processor

Table 1: The tables show performance results of the *Counting* algorithm for four problems A-D, on the NCUBE/2, a nonshared memory machine. Column 1 shows number of processors. Column 2 shows the number of control messages that were used to detect quiescence. Column 3 shows the number of iterations of Phase 1 and Phase 2 performed by the algorithm for that execution run. Column 4 shows the time in milliseconds that elapsed between the onset and detection of quiescence by the algorithm. In Problem D the number of activation messages varies with the number of processors used — there are 2000 activation messages per processor.

behavior may change because of minor delays in processing other messages due to the quiescence detection messages. For Problem C, the overhead of the quiescence detection algorithm ranges from 10% to 17%. For Problem D, the overhead of the quiescence detection algorithm ranges from 3% to 4%.

Our quiescence detection algorithm adapt automatically to system loads. When the system is heavily loaded very few control messages are generated thus not interfering with the user computation. More control messages are generated and processed when the system is lightly loaded, but this time there isn't enough user work with which the control messages can interfere.

#PE	2	4	8	16	32	64	128
Problem A	8982/9128	4775/4861	2547/2618	1452/1417	889/861	593/646	511/429
Problem B	55234/53293	29618/27068	16850/16173	9512/9689	5758/5731	3732/3610	2710/2525
Problem C	43/39	56/51	57/53	68/61	73/67	84/75	108/91
Problem D	983/944	984/945	988/948	993/954	1007/967	1034/995	1088/1048

Table 2: The table shows the execution times for Problems A, B, C and D with/without the *Counting* quiescence detection algorithm on a nonshared memory machine, NCUBE/2. All the times are in milliseconds.

4 Summary and Discussion

In this paper, we have presented an algorithm that detects quiescence — a condition when all the processes are idle and there are no messages in transit. This algorithm works with dynamic creation of processes, and allows multiple processes on each processor. The salient features of the algorithm, called *Counting*, are:

1. It maintains separate counts of messages sent and messages processed.
2. The counts are collected along a spanning tree.
3. The algorithm is intentionally slowed down by stopping its progress through each processor until that processor becomes idle, thus making the algorithm adaptive to the load conditions.

The algorithm detects and reports quiescence very quickly after the onset of quiescence, and does not cost a significant overhead. It is implemented in the Charm parallel programming system, and was used effectively in many application programs, including many state-space search programs. Although the algorithm was presented as a two-phase one, it is possible to formulate it with just one recurring phase. The two-phase formulation was chosen for the ease of presentation.

The *Counting* algorithm was motivated by an earlier quiescence detection algorithm [14]. In Shu's method, quiescence was reported if the system was in a state of idleness for some specified interval δt. The choice of the time interval was critical — δt had to be greater than the worst case delivery time of a message. The primary drawback of Shu's algorithm was that it was not easy to estimate the worst case message delivery time and hence δt.

The basic ideas in our algorithm can be applied to other situations involving distributed information gathering. In particular, we are interested in pursuing its application to distributed garbage collection.

A Correctness proof for Counting algorithm

In this section, we offer an informal proof that the *Counting* algorithm will correctly detect quiescence. The proof is in two parts: first, we prove that if the system is quiescent the algorithm will detect it, and, next we prove that the algorithm detects quiescence *only* when the system is indeed quiescent.

Theorem 1 If quiescence has occurred then the *Counting* Algorithm will report it.

Proof: If quiescence has occurred, then there are no activation messages remaining to be processed. Since messages are not created spontaneously or lost, the counts for the creations and processings of activation messages must match, and the algorithm will detect quiescence in at most two iterations of Phase 1 followed by one iteration of Phase 2. □

Theorem 2 The *Counting* Algorithm will not report quiescence unless the system has been quiescent.

Proof: The proof is by contradiction. Assume that even though the *Counting* algorithm has reported quiescence, the system is not quiescent. Since the system is not quiescent, atleast one of the following must be true (from the conditions for quiescence) : there are unprocessed activation messages or there is atleast one busy processor.

We introduce some notation first. Let τ_i and υ_i denote the time at which the last instances of Phase 1 and Phase 2, respectively, were completed on processor i. Let Υ denote the time at which processor 0 made the broadcast to initiate Phase 2. And for a message m, let c_m denote its time of creation and p_m its time of processing. Since the broadcast to begin Phase 2 occurred *after* all processors had completed Phase 1 and *before* any processor began Phase 2, therefore:

$$(\forall i, j)(\tau_i < \Upsilon < \upsilon_j) \tag{1}$$

Can there be any unprocessed activation messages in the system after Phase 1 if quiescence has been reported? Let us assume that there are unprocessed messages in the system after Phase 1. Since each processor was idle at the end of Phase 1 on that processor, and no messages can be created spontaneously, atleast one of the unprocessed messages, say a, would have to have been created before Phase 1 on some processor. At the end of Phase 1 the counts for number of messages created and processed were the same (otherwise Phase 2 would not have been started). Therefore, for every message whose creation, but not processing, occurred before Phase 1, there would be a corresponding message for which the processing, but not the creation, occurred before Phase 1; otherwise the counts would not match. Let b be a message whose processing, but not creation, occurred before Phase 1. Then the following is true:

$$(\exists i)(c_a < \tau_i) \tag{2}$$

$$(\exists i)(\tau_i < p_a) \tag{3}$$

$$(\exists i)(p_b < \tau_i) \tag{4}$$

$$(\exists i)(\tau_i < c_b) \tag{5}$$

At the end of Phase 2, the counts for total number of messages created and processed in the system, N_c and N_p, have the same values as they had after the last iteration of Phase 1. Since n_c and n_p are monotonically non-decreasing, their values must have remained unchanged on each processor, implying that no messages were created or processed on any processor between Phase 1 and Phase 2. Therefore a and b must have been created and processed, respectively, after Phase 2.

$$(\exists i)(\upsilon_i < c_b) \tag{6}$$

But b could not have been created after Phase 2, for by combining Equations 1, 4, and 6, we get:

$$(\exists i, j)(p_b < \tau_i < \Upsilon < \upsilon_j < c_b) \tag{7}$$

Since a message could not have been created after it had been processed, no message such as b could exist. But the messages a and b exist in pairs; therefore there can be no messages which are unprocessed after Phase 1 if quiescence has been reported.

Can any processor be busy after Phase 1? The end of Phase 1 on a processor implies that the processor is idle (and has received messages from its children), therefore if it is busy after that it must be because it created or processed some activation message. However no messages could have been created or processed after Phase 1, otherwise there would have been an increase in the counts n_c or n_p, and quiescence would not have been detected. Therefore no processor could have been busy after Phase 1.

We have proved that after quiescence has been reported there are no unprocessed activation messages and no busy processors in the system. Therefore when quiescence is reported, the system is indeed quiescent. $\square$

References

[1] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11, 1, August 1980.

[2] N. Francez. Distributed termination. *ACM TOPLAS*, pages 42–55, Vol. 2, No. 1, January 1980.

[3] J. Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM TOPLAS*, pages 37–34, Vol. 4, No. 1, January 1982.

[4] C. Hazari and H. Zedan. A distributed algorithm for distributed termination. *Information Processing Letters*, pages 293–297, 24, 1987.

[5] S. P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, pages 43–46, 17, 1 (July 1983).

[6] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, pages 195–200, 30, 1989.

[7] S. T. Huang. Termination detection by using distributed snapshots. *Information Processing Letters*, pages 113–119, 32, 1989.

[8] T. H. Lai and T. H. Yang. On distrbuted snapshots. *Information Processing Letters*, pages 153–158, 25 (1987).

[9] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, pages 40–52, 1, 1(1986).

[10] S. Chandrasekharan and S. Venkatesan. A message-optimal algorithm for distributed termination detection. *Journal of Parallel and Distributed Computing*, pages 245–252, 8, 1990.

[11] L. V. Kale. The Chare Kernel Parallel Programming System Programming System. In *International Conference on Parallel Processing*, August 1990.

[12] K. De and B. Ramkumar and P. Banerjee. ProperSYN: A Portable Parallel Algorithm for Logic Synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[13] B. Ramkumar and P. Banerjee. Portable Parallel Test Generation for Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[14] W. Shu. *Chare Kernel and its implementation on multicomputers.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1990.