A DISTRIBUTED DEADLOCK DETECTION AND RESOLUTION ALGORITHM FOR PROCESS NETWORKS

Gregory E. Allen, Paul E. Zucknick, and Brian L. Evans

Applied Research Laboratories, and Dept. of Electrical and Computer Engineering
The University of Texas at Austin
P.O. Box 8029, Austin, TX 78713-8029
{gallen,zucknick}@arlut.utexas.edu, bevans@ece.utexas.edu

ABSTRACT

In the Process Network (PN) model, multiple concurrent processes communicate over unidirectional FIFO queues. PN is useful for modeling signal processing systems of streaming data, and naturally captures parallelism in these systems. PN provides formal execution properties to alleviate the difficulties of threaded and distributed programming, and naturally maps onto parallel and distributed targets. For a large class of PN, clever run-time scheduling can permit execution in bounded memory. In general, PN termination and boundedness cannot be statically determined, so correct bounded scheduling of PN requires run-time deadlock detection. We present the first algorithm that correctly performs dynamic deadlock detection and resolution for bounded scheduling of PN. The proposed algorithm is a modification of a distributed deadlock detection algorithm by Mitchell and Merritt.

Index Terms— Kahn process networks, deadlock resolution, dynamic scheduling, distributed computing, signal processing

1. INTRODUCTION

The Process Network (PN) model was introduced by Kahn in 1974 [1]. It is a dataflow model in which concurrent processes communicate (only) over FIFO queue channels. Process Networks naturally model functional parallelism, and can model data parallelism. They are well-suited for modeling signal processing systems, including parallel and distributed systems. It has widely been observed that concurrent and distributed programming is difficult [2]. The formal underpinnings of PN are a potential aid to easing this difficulty.

A Process Network is represented as a directed graph, where each node represents a process and each arc (or edge) represents a queue channel, directed from producer to consumer. This model is natural for describing the streams of data samples in a signal processing system. Typically, a system block diagram can naturally map onto a Process Network.

Kahn's model, in general, requires infinite memory for execution. As work toward bounded-memory scheduling of PN has progressed, several implementations have been published [3][4][5][6]. One of the earliest PN models reported to be deployed in the field was our real-time 3-D sonar beamformer on a 12-processor Sun workstation [7].

A required feature for correct bounded-memory scheduling of PN is dynamic deadlock detection. We present a simple, distributed, scalable algorithm that can detect deadlock in a Process Network, locate the culpable process and queue, and resolve the deadlock if it is possible.

2. THE PROCESS NETWORK MODEL

2.1. Kahn Process Networks

Kahn proposed the Process Network model [1], in which multiple concurrent processes are connected via unidirectional FIFO queues to form a network. Each process may have any number of incoming or outgoing queues, and may communicate to other processes only via these queues. Kahn showed that a Process Network program is *determinate* – the results produced on all queues are the same for every possible execution order, including concurrent execution.

PN uses simple, local rules for scheduling, and is therefore distributable and scalable. A node suspends execution when it attempts to read from an empty queue. When writing to a queue, a node can never be suspended. Hence, queues can become infinitely long. A node cannot check for the presence of data on a queue, nor can it read from more than one queue at a time.

Infinitely large queues are undesirable, as execution in bounded memory is necessary for any practical implementation. Termination and total stream lengths are properties of the program and do not depend on the execution order of the nodes in the network. However, the number of unconsumed samples that can accumulate on queues does depend on the execution order [8].

Supported by the Independent Research and Development program at Applied Research Laboratories: The University of Texas at Austin.

2.2. Parks's Bounded Scheduling

Parks proposed [8] a scheduling policy to yield a bounded execution of a PN if it is possible. Because it is undecidable whether a PN can be scheduled in bounded memory, the scheduler must work dynamically as the program executes. Parks' policy suspends a node for writing to a full queue. If a queue is chosen to be too small, *artificial deadlocks* (processes permanently blocked on a full queue) may occur that were not present in the original (unbounded) PN. Parks proposed a global dynamic deadlock detector that increases the size of the smallest blocked full queue.

Although counterexamples to Parks's bounded scheduling policy eventually surfaced, his work was a major step toward making the model more than an academic curiosity.

2.3. Geilen and Basten

Geilen and Basten further examined [9] bounded scheduling of PN, and introduced an additional requirement: that the PN be *effective*. They call a PN effective only if every token that is produced on a queue is eventually consumed. Bounded scheduling of a non-effective PN may result in an incomplete execution – that is, execution may terminate earlier than it would have in the unbounded model. Unfortunately, it is undecidable in general whether a PN is effective.

Geilen and Basten also showed [9] that a dynamic deadlock detector should detect *local* deadlocks, not only global ones as proposed by Parks.

If a process is blocked reading from an empty queue (or writing to a full queue), it is dependent on a unique other process – that at the other end of the queue. The only other resolution is to increase the size of a blocked full queue. Because any blocked process depends on some unique other process, this gives rise to chains of dependencies. If a dependency chain is cyclic it indicates a local deadlock, and no further progress can be made without external resolution.

By detecting and resolving local deadlocks, we can execute bounded, effective PNs to completion in bounded memory. It is impossible to schedule all PNs in bounded memory. However, this is a very large set of PNs, and arguably the most interesting set for practical applications.

3. DYNAMIC DEADLOCK DETECTION

Proper bounded-memory scheduling of PN requires dynamic detection of local deadlocks. If the detected deadlock is artificial, then it must be resolved in order to preserve the determinacy of the PN. We present an algorithm that performs distributed dynamic deadlock detection and resolution (D4R).

Olson and Evans presented [6] an application of an algorithm by Mitchell and Merritt [10] to deadlock detection for PN. Although originally designed for distributed database applications, this Mitchell and Merritt algorithm appropriately detects deadlocks for PN. The Mitchell and Merritt algorithm is a *single resource* algorithm – at any given time, a process is waiting on at most a single resource. This is also the model specified for PN – a node may be blocked on at most a single other process. The application presented by Olson and Evans could detect whether a deadlock was present, but it did not specifically locate or resolve the deadlock.

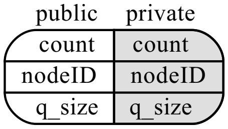

Fig. 1. D4R algorithm state data at each node.

We present an algorithm based on a different Mitchell and Merritt algorithm from the same paper [10]. This second algorithm uses process priorities, and identifies the lowest priority process in the deadlock cycle so that it can be resolved. We assign the process priorities such that the algorithm determines whether a deadlock is real or artificial, and identifies the node that is blocked writing to the queue which must be lengthened in order to resolve the deadlock.

Using the same nodes and edges as in a PN graph, we can construct a *wait-for* graph for the D4R algorithm. Here, an edge indicates that a process is blocked and waiting on a single other process. The direction of the wait-for edges are from the waiting process to the process being waited on. Note that the edges in the wait-for graph coincide with the edges in the original PN graph. For read-blocked nodes, the direction of the edge is opposite that in the original PN graph.

For our version of the algorithm each node contains algorithmic state data as shown in Figure 1, consisting of public and private sets of a triple: a non-decreasing counter *count*, a unique node identifier *nodeID*, and a queue size variable *q_size* (which serves the function of Mitchell and Merritt's priority variable). Each process is initialized with public and private sets equal. The public set changes as the algorithm progresses, but the private set remains unique to that node.

Note that *count* and *nodeID* are combined into a single variable in the Mitchell and Merritt paper. Our use is consistent with their suggestion of "keeping the low-order bits of the label constant and unique while increasing the high-order bits when desired [10]." We use the notation *count*:*nodeID* to show these variables as concatenated into one.

The node state data and wait-for edges define the state of the D4R algorithm at any time. Figure 2 shows the possible state transitions for this algorithm in the order that they occur: *Block*, *Transmit*, *Detect*, and *Activate*. State data which is unchanging or unused in a transition has been left blank.

The *Block* state transition occurs when a PN node blocks on a queue, creating an edge in the wait-for graph. The blocking node's state data is fully initialized, and its count variables are incremented to be greater than that of both nodes. Our *q_size* variable is also initialized during this state transition.

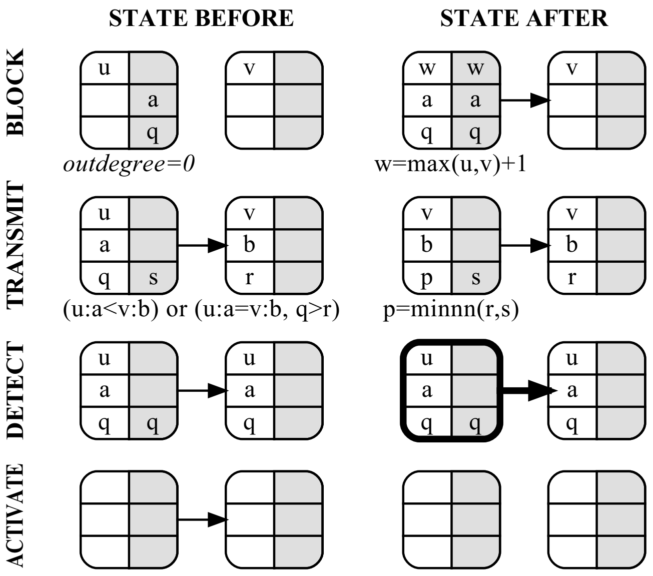

Fig. 2. State transitions for the D4R algorithm.

This is what permits our D4R algorithm to determine whether a deadlock is artificial, and localize the smallest full blocked queue. The variable *q_size* is set as follows: when a process blocks on a read, set *q_size* = -1; when a process blocks on a write, set *q_size* to the size of the blocking (full) queue.

The *Transmit* state transition occurs when a waiting process detects a change in the public state of the node upon which it is waiting, and certain criteria are met: if the other node's public *count*:*nodeID* is larger than its own, or if they are equal and *q_size* is smaller *in a non-negative sense*. That is, any positive *q_size* is "smaller" than a negative *q_size*. (This can be easily implemented as an unsigned comparison of 2's complement numbers.) If the criteria are met and the state transition occurs, then the waiting node replaces its public *count*:*nodeID* with the one it just read, and sets its public *q_size* to the non-negative minimum of the two nodes. In implementation, each time a node's public state changes, it will notify any dependent nodes. The effect is that larger *counts* and "smaller" *q_sizes* migrate along the edges of the wait-for graph, in the opposite direction.

The *Detect* state transition occurs when a waiting process sees that its entire public set matches that of the node upon which it is waiting, and its public and private *q_sizes* also match. It then knows that it is not only a part of a deadlock cycle, but that it also has the "smallest" *q_size* in the non-negative sense. Only one process will detect the deadlock, and the value of *q_size* tells the type of deadlock: if *q_size* is negative, this is a real deadlock; otherwise this is an artificial deadlock, and the smallest, blocked, full queue has been identified. This is precisely the queue that must be lengthened for correct bounded scheduling of PN [9].

The *Activate* state transition may occur after *Detect*. If the deadlock was real, the program has terminated. If it was artificial, the culpable queue has been lengthened so that the PN program may continue. Of course, *Activate* will also occur after *Block* repeatedly as the PN proceeds normally in the absence of any deadlock.

4. A PROOF OF CORRECTNESS

We have intentionally made portions of our algorithm equivalent to the Mitchell and Merritt priority-based algorithm. We therefore include the following Theorem from their paper [10] without proof.

Theorem 1 *If a cycle of N nodes forms and persists long enough, the lowest priority (smallest in a non-negative sense) process in the cycle will execute the Detect step after at least N-1 and at most 2N-2 consecutive Transmit steps.*

Mitchell and Merritt assign a fixed priority to each node, whereas we set *q_size* at each *Block* step. We therefore need to show that we have not violated the rules of their algorithm.

Lemma 2 *If a node has outdegree=0 in its wait-for graph, it can change the value of its private* q_size. *That is, a node's private* q_size *need only be fixed when it has non-zero outdegree in the wait-for graph.*

Proof The private *q_size* of a node is unused unless the node has a non-zero outdegree. No other node can access its private *q_size* at any time. The deadlock detection algorithm will therefore proceed unaffected. ∎

Theorem 3 *A node can change its* q_size *during its Block state transition (both public and private).*

Proof Just prior to a Block state transition, a node must have outdegree=0 because this is a single-resource algorithm, and a process can only block on a single queue. By Lemma 2, the node can set its private *q_size* at this time. During the Block state transition, the node also copies its private *q_size* to its public *q_size*. ∎

We have shown that we can correctly schedule bounded, effective Process Networks using the D4R algorithm as described in Section 3. Our algorithm simply modifies that of Mitchell and Merritt [10] to set the *q_size* based on the size of the queue that we are about to block on, and whether we are blocking on a read or a write.

As a further discussion topic, a *count* that continuously increases is not implementable for a program that never terminates. We wish to examine the possibility of periodically resetting the *count* variables to prevent "rolling over".

Proposition 4 *If a node has outdegree=0 and indegree=0 in the wait-for graph, it can change its public and private* count *variables. The* nodeID *variable is still unique and unchanged.*

Proof The *count* variables are only used when the node has non-zero outdegree or indegree. When both are zero, it is as if the node has never been in a wait-for graph. Adding an arc to the wait-for graph requires a Block step, which will increment the *count* variables as necessary for the algorithm. ▮

While it is easy to determine that outdegree=0, it is not obvious how to inexpensively and easily determine that indegree=0. For now, the *count* variables simply must be large enough to prevent rollover in any reasonable amount of time.

5. IMPLEMENTATION

In addition to proving correctness, we have created an implementation of the presented D4R algorithm. This implementation is integrated into an update of our Computational Process Networks (CPN) framework [7]. CPN is an extended model of PN, as well as a high-performance framework implementation using POSIX threads. The previously published version of this framework was limited to running on a single symmetric multiprocessing workstation. However, work has been underway to extend this framework to permit execution of CPN programs over a distributed network of workstations. Local queues are implemented with shared memory, and remote queues will primarily be implemented over TCP sockets. As with the original implementation, the goals have been high performance and very low overhead, with the ability to handle high-throughput streams of data for signal processing.

It is important to note that performance is not a goal for the implementation of the D4R algorithm. Artificial deadlock in a program is an undesired state, and considered an exception to normal operation. In a real deadlock, that portion of the program has terminated. In any case where there is a performance tradeoff between the D4R algorithm and normal queue operation, we choose that which is faster for normal queue operation. This makes the framework faster and lower overhead for programs where the minimum queue lengths have already been determined.

We have demonstrated that the presented D4R algorithm can dynamically detect and resolve artificial deadlocks. It also detects real deadlocks, which indicate that some local portion of the program will make no further progress. Our CPN framework therefore correctly schedules bounded, effective Process Networks in bounded memory. Because both this bounded scheduling and the presented D4R algorithm are based completely on local data between connected nodes (and do not require any global synchronization) distributed implementations are straightforward and scalable.

Our CPN framework implementation, as well as examples of the presented D4R algorithm, are available at:

`http://www.ece.utexas.edu/~allen/CPN/`

6. CONCLUSION

Process Networks are useful for modeling signal processing systems of streaming data, and naturally captures parallelism in these systems. Bounded, effective Process Networks can be executed in bounded memory, but require dynamic deadlock detection and resolution. We present the first algorithm to not only detect deadlock, but to determine whether it is real or artificial, and to resolve it. This distributed algorithm permits correct scheduling of bounded PN.

7. REFERENCES

[1] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing*, pp. 471–475, Aug. 1974.

[2] Edward A. Lee, "The problem with threads," Tech. Rep. UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan. 10 2006.

[3] T. M. Parks and D. Roberts, "Distributed process networks in java," in *Proc. Int. Workshop on Java for Parallel and Distributed Computing*, Nice, France, Apr. 2003.

[4] J. Vayssiere, D. Webb, and A. Wendelborn, "Distributed process networks," Tech. Rep. TR 99-03, Dept. of CS, University of Adelaide, Australia, Oct. 1999.

[5] A. Amar, P. Boulet, J. Dekeyser, and F. Theeuwen, "Distributed process networks using half FIFO queues in CORBA," Tech. Rep. RR-4765, INRIA, Mar. 2003.

[6] A. G. Olson and B. L. Evans, "Deadlock detection for distributed process networks," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Proc.*, Philadelphia, PA, Mar. 2005, pp. 73–76.

[7] G. E. Allen and B. L. Evans, "Real-Time sonar beamforming on workstations using process networks and POSIX threads," *IEEE Trans. on Signal Processing*, pp. 921–926, Mar. 2000.

[8] T. M. Parks, *Bounded Scheduling of Process Networks*, Ph.D. thesis, EECS Department, University of California, Berkeley, CA 94720-1770, Dec. 1995, Technical Report UCB/ERL-95-105.

[9] M. Geilen and T. Basten, "Requirements on the execution of Kahn process networks," in *Proc. European Symposium on Programming*, 2003, pp. 319–334.

[10] D. P. Mitchell and M. J. Merritt, "A distributed algorithm for deadlock detection and resolution," in *Proc. ACM Symposium on Principles of Distributed Computing*, 1984, pp. 282–284.