Open access • Proceedings Article • DOI:10.1109/ICASSP.2005.1416243

Deadlock detection for distributed process networks — Source link

A.G. Olson, Brian L. Evans

Published on: 18 Mar 2005 - International Conference on Acoustics, Speech, and Signal Processing

Topics: Deadlock prevention algorithms, Deadlock, Edge chasing, Concurrency control and Concurrent computing

DEADLOCK DETECTION FOR DISTRIBUTED PROCESS NETWORKS

Alex G. Olson and Brian L. Evans

Embedded Signal Processing Laboratory
The University of Texas at Austin, Austin, TX 78712 USA
{aolson,bevans}@ece.utexas.edu

ABSTRACT

The Process Network (PN) model, which consists of concurrent processes communicating over first-in first out unidirectional queues, is useful for modeling and exploiting functional parallelism in streaming data applications. The PN model maps easily onto multi-processor and/or multi-threaded targets. Since the PN model is Turing complete, memory requirements cannot be predicted statically. In general, any bounded-memory scheduling algorithm for this model requires run-time deadlock detection. The few PN implementations that perform deadlock detection detect only global deadlocks. Not all local deadlocks, however, will cause a PN system to reach global deadlock. In this paper, we present the first local deadlock detection algorithm for PN models. The proposed algorithm is based on the Mitchell and Merritt algorithm and is suitable for both parallel and distributed PN implementations.

1. INTRODUCTION

Many successful approaches to high-performance computing exploit available parallelism in a computation. One approach finds parallelism by analyzing sequential programs, as done by modern compilers and processors. Another approach relies on the programmer to expose the parallelism. An example of the latter is Kahn's Process Network (PN) model [1], which consists of concurrent processes communicating over one-way first-in first-out (FIFO) unidirectional channels.

The PN model captures parallelism among the processes (functions) performed by the computation, but does not expose parallelism available in the way data is represented. The PN model is well suited to data-intensive processing, including streaming media processing. One of the earliest PN models deployed in the field was for a real-time 3-D sonar beamformer running on a 12-processor Sun workstation, designed for deployment in submarines [2]. This 3-D sonar beamformer deployment occurred nearly 26 years after Kahn proposed the PN model.

The PN model has a natural block diagram graphical syntax in which processes are blocks and communication channels are unidirectional arcs. On a channel, tokens (samples) are received in the order they were transmitted, but the tokens do not carry any time information. This abstracting away of time coupled with a mild restriction on how a process accesses data on its input port(s) gives the PN model an important property of being *determinate*. The property of determinism means that the behavior of a PN program does not depend on the way the processes are scheduled. Hence, PN programs can be targeted onto a wide range of implementation architectures and yet behave in the same way.

For targeting desktop and multi-processor computers, several (non-distributed) PN frameworks exist. Many frameworks map each Kahn process in a PN program to a thread. These frameworks rely on the operating system to distribute the threads to achieve load balance among the available processors [2]. However, large multi-processor systems are extremely (economically) expensive. Currently, multiple general-purpose workstations may be purchased for fraction of the price of a large multi-processor system, and the multiple workstations offer greater scalability and combined performance. For many applications, a *distributed* Process Network (DPN) framework allows for greater performance at a reduced economic cost over PN implementations running on multi-processor systems. Unfortunately, few DPN frameworks exist [3, 4, 5].

Deadlock detection is required for scheduling PN programs in bounded memory. One example of deadlock is that process A is blocked waiting for data from process B while process B is blocked waiting for data from process A. Bounded memory PN implementations place bounds on queue sizes [6], which can create "artificial deadlocks". However, distributing the PN model incurs challenges as channels take on unpredictable latencies. Even though the model itself is untimed, a distributed implementation contains many challenges relevant to distributed systems. The lack of a global clock and shared memory make correct deadlock detection more difficult.

In this paper, we present the first local deadlock detection algorithm for PN models. The proposed algorithm is based on the Mitchell and Merritt algorithm and is suitable for both parallel and distributed PN implementations. We also describe a distributed PN implementation using the proposed deadlock detection algorithm.

2. THE PROCESS NETWORK MODEL

2.1. Kahn

In 1974, Kahn proposed [1] a model of computation based on data tokens (and their flow). The term *token* is a general term for any unit of data. Kahn suggested that multiple processes executing concurrently and communicating over unidirectional channels could perform a computation. His channels may be modeled as reliable FIFO queues, but may be unbounded in length. The channels/queues provide a loose coupling between producers (processes emitting tokens) and consumers (processes receiving tokens). Reads from a channel are destructive in the sense that tokens are consumed (dequeued).

Kahn's PN model is determinate since the *history* of tokens on any channel does not depend on how the processes are sched-

uled. The *correctness* of a computation under the PN model is not affected by the rates or order in which processes execute. A necessary condition for a PN to be determinate is the use of blocking reads. If a process attempts to read more tokens from a channel than are available, the process blocks until enough data becomes available. Furthermore, a process is not allowed to test a channel for the presence of tokens. A process may attempt to read from at most one input port or write to at most one output port at a time.

2.2. Parks

Kahn's assumption of "unbounded channel capacities" translates into "unbounded memory" usage when the PN model is implemented on a computer. Since the PN model is Turing complete [1], one cannot predict memory requirements statically [7]. In 1995, Parks proposed [6] an algorithm for scheduling PN's under bounded channel capacities. He proposed that a process is also blocked if it attempts to write one or more tokens to a channel lacking sufficient available capacity. This has the potential to create, using Parks' terminology, *artificial* deadlock. Artifical deadlocks only arise from placing bounds on channel capacities. They do not occur in the original Kahn model. Park's algorithm first waits until the system reaches global deadlock and then considers all channels to which a blocked process is writing. Of these, the algorithm increases the capacity of the smallest full channel, so the write to that channel can complete. He proves this algorithm finds a set of bounded channel capacities whenever such bounds exist.

2.3. Geilen and Basten

Geilen and Basten [8] show that Parks' algorithm can be applied when *local* deadlocks are detected, instead of waiting for detection of *global* deadlock. Their scheduling algorithm also maintains bounds on channel capacities when such bounds exist. This is significant because not all local deadlocks will eventually cause global deadlock. For example, if a system is composed of two disjoint computations, deadlock in one will not cause deadlock in the other.

2.4. Allen and Evans

Allen and Evans combined [2] the PN model with Karp and Miller computation graphs[9]. In the resulting model, known as Computational Process Networks (CPN), a process may *consume* fewer tokens than it *reads*. This model is highly useful for many computations, such as FIR filters. The performance of some computations is limited more by memory bandwidth rather than CPU performance. The CPN model allows memory bandwidth to be halved as it often eliminates the need to copy tokens from the channel buffer into an application-specific buffer. Artificial deadlock, arising from placing bounds on channel capacities, is still possible.

3. DEADLOCK DETECTION

3.1. Previous Work

Since the PN model is Turing complete[1], deadlock is only detectable at run-time. Only global deadlock detection has been implemented in a few non-distributed PN implementations [10, 11, 12, 13]. These detect only global deadlock. A few DPN implementations exist [3, 4, 5]. Of these, none detects deadlocks. What follows in the next section is an original application of an existing deadlock detection algorithm to DPN's.

3.2. Mitchell and Merritt's Algorithm

Kahn's specifies in his PN model that a process may be blocked on at most one other process at a time. From the large body of distributed deadlock detection algorithms, we consider the set commonly known as 'single-resource' algorithms [14]. This class of algorithms assumes that a process is waiting on at most one other process at a time. An important aspect of these algorithms is they are not concerned with actual management of resources, but only the manner in which a process *waits* on another process. One very simple algorithm was developed by Mitchell and Merritt [15] and was developed for distributed databases. We show a novel application of this algorithm to the PN model.

In their algorithm, each process contains two labels: a public label and a private label. In this context, a label is just an abstraction for a numeric value. Initially, the public labels of all processes are initialized to unique values. Each process's private label is set equal to its corresponding public label. When a process X begins waiting on another process Y, process X sets both its labels to a value greater than the public labels of both X and Y. This step is known as the *blocking* step and the waiting process, X, is said to be *blocked*. While some process X is *blocked*, it periodically polls the public label of the process,Y, for which it is waiting. During this time, if the public label of process Y becomes greater than that of X, X sets only its public label to be equal to that of Y. This action is known as the *transmit* step. For a cycle of N waiting processes, the transmit step will be invoked at most $N-1$ times before deadlock is detected. This algorithm also ensures that in a cycle of waiting processes, exactly one process detects deadlock. Furthermore, false deadlock detection is impossible. These two qualities make this algorithm an ideal deadlock detection scheme in the implementation of a PN scheduling algorithm.

This algorithm creates larger labels in the system every time a process becomes blocked (during the execution of the blocking step). While a process is blocked, the largest label tends to propagate in the opposite direction of a sequence of waiting processes. Essentially deadlock is detected when a label makes a round-trip. A graphical example is depicted in Figs. 1, 2, and 3. In the following discussion, p_n will refer to the process with n as the second component of its private label.

Fig. 1 shows the state of the labels at system initialization; each process's public label is unique. Although each label is shown as an ordered pair, the two components may be considered as the high-word and low-word portions of a single integer [15]. The essential requirement is that all private labels remain unique throughout the execution of the algorithm. Fig. 2 shows the state of the system after a cycle of three waiting processes perform the blocking step of the algorithm. To arive at this state, p_1 begins waiting (performs the 'blocking step") on process p_3. Then process p_5 begins waiting on p_1. Lastly, p_3 begins waiting on p_5. Finally, Fig. 3 shows the state of the system after two processes have performed the transmit step. This state results after p_5 performs the transmit step, followed by p_3. Here, only process p_3, with private label $(7, 3)$, detects the deadlock. Note, in a real system, some processes may be executing the blocking step while others are concurrently executing the transmit step. This may cause the sequence of label changes to differ. However, the correctness of the algorithm is maintained, mainly due to the fact that label values are non-decreasing.

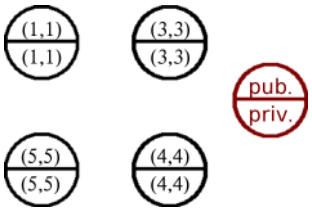

Fig. 1. Initial state. Each process's public labels are unique.

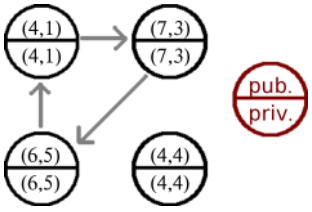

Fig. 2. State after the 'blocking' step has been performed by each process in the cycle.

4. DEADLOCK DETECTION THEORY IN DISTRIBUTED PROCESS NETWORKS

The fact that the Mitchell-Merritt algorithm is resource-based does pose a subtle problem in a DPN. A cycle of blocked Kahn processes does not directly imply the existence of deadlock. If there are any tokens *en route* (or untransmitted) to any blocked process, the system is not necessarily deadlocked. Conceptually, the 'resource' in a PN is the token a blocked process is waiting to read or the queue slot a blocked process is waiting to write. Therefore, it is necessary for a blocked Kahn process to ensure that the relevant channel is empty before performing the blocking step of the Mitchell-Merritt algorithm. This is a simple task if two assumptions are made. First, the channels guarantee reliable and FIFO delivery. This assumption can be met by many communication protocols, such as TCP. Second, a total ordering exists over all control messages and token data exchanged between any two processes. This second assumption can be met easily by sending token data and control messages along a single, reliable, FIFO channel. When a process becomes blocked, it requests the public label of the process on which it is waiting. Upon receipt of such a request, the waitee will reply with its public label. Between the request and the reply for the label, if the blocked process becomes unblocked, it will not perform (upon receipt of the label) the blocking or transmit step of the Mitchell-Merritt algorithm. The FIFO delivery of the communication channel ensures that any *en route* token data will be processed before the requested label. In the context of distributed systems, the label request message functions as a flush message.

A characteristic of the Mitchell-Merritt algorithm is that the timing of when a blocked process performs the blocking or transmit steps is not critical. To reduce network traffic, it may be desirable for a blocked Kahn process to delay the sending of a 'label request' message before performing either the blocking step or transmit step of the algorithm. Good choices of the delay depend on the communication channel latency and tolerable deadlock detection latency.

In addition, (for deadlock detection) the quantity of state information required is independent of the number of channels. Furthermore, the amount of state each process is required to store is independent of the number of processes in the system. The simplicity of the Mitchell-Merritt algorithm makes it ideal for both parallel and distributed PN implementations.

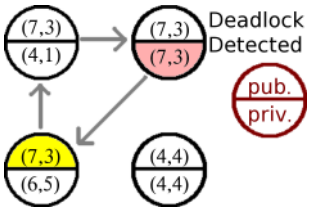

Fig. 3. State after two executions of the 'transmit' step. Deadlock has been detected.

5. IMPLEMENTATION

5.1. Channel Description

We have implemented a high-performance DPN framework in C++ using POSIX threads.

In our implementation, we use the TCP/IP protocol for communication links between two processes. Rather than operating at the token-level, our framework operates at the byte-level. This design choice allows our framework to handle both large and small tokens efficiently. We also define two types of channels (relative to each process): input channels and output channels. Input channels are those channels from which a Kahn process reads tokens. Output channels are those channels to which a Kahn process writes tokens. Each channel, input or output, contains a queue. For output channels, the queue aggregates tokens before transmission to better utilize network bandwidth. For input channels, the queue functions as the Kahn channel queue. By explicitly modeling the Kahn channel queue as a real queue, we allow control data to be exchanged over the communication link even when both processes are blocked.

The communication between any two processes is fairly simple. When the communication link is first established, the input channel sends the capacity of its queue over the link. The output channel uses this value to maintain a lower bound on the available capacity of the input channel's queue. Each time a process consumes a token from its input channel, the channel, after a very small delay, sends a message across the link that indicates how many bytes were consumed. This is essentially a simple sliding window algorithm being used for flow-control. Since we use TCP/IP for the communication link, reliable and FIFO delivery of data is guaranteed.

Each process in our framework consists of two threads: a computation thread and a communication thread. The computation thread is implemented by the user and performs the computation, such as FIR filtering. The communication thread is implemented by the framework and manages communication between the TCP/IP link and the queues on each end. The queues provide an interface between the two threads. For reads, the computation thread blocks until a sufficient amount of data is available in the input channel's queue. When token data is written to an output channel, the channel decrements its estimate of the available capacity of the input channel. When token data is read from the input channel, the channel sends a message indicating the number of bytes that were consumed. Upon receipt of such a message, the

output channel increments its estimate of the capacity of the input channel by that amount. Thus at all times, this count is a lower bound on how much token data (in bytes) can be immediately sent over the link.

5.2. PN and CPN Support

Channels in our implementation support both Kahn PN semantics (where all read tokens are consumed) as well as CPN semantics (where not all read tokens are consumed). We also provide a zero-copy operation for CPN's, which is similar to [2]. That is, a process acquires read and write pointers into the queues rather than copying data in or out. This reduces memory bandwidth requirements in high data-rate applications,

5.3. Deadlock Detection

When a process blocks for a sufficient amount of time, Mitchell and Merritt's algorithm is activated. We currently use a fixed threshold of time. The blocked process sends a "Label Request" message to the process on which it is blocked. Upon the message's receipt, a process will transmit any queued tokens and then reply with its public label. If the blocked process remains continuously blocked between the request and receipt of the label, it will perform the blocking step of the algorithm. Otherwise, the label is discarded and another request will be sent after some time. The same technique is used for successive executions of the transmit step of the deadlock detection algorithm.

5.4. Performance and Memory Requirements

The memory requirement of the deadlock detection algorithm is very low and independent of the number of processes or channels in a system. Each process stores only two pairs of labels, which we represent as an ordered pair of 32-bit integers. The network and CPU overhead is negligible since the algorithm only activates intermittently and only while a process is blocked. While most of the discussion has focused on DPN implementations, this algorithm is equally suitable for parallel PN implementations. In a non-distributed parallel implementation, the communication channel could be a shared block of memory. Events and callbacks could replace the sending of label requests and replies. Any local deadlock would be detected almost instantaneously.

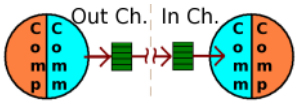

Fig. 4. Structure of our implementation. Each circle represents a process. Each process contains two threads, for computation and communication. For each channel of each process, a queue provides the interface between the two threads.

6. CONCLUSION

For many applications, distributing the Process Network model offers greater (scalable) performance at reduced economic cost over a non-distributed implementation. However, bounded memory execution of Process Networks requires run-time deadlock detection. In this paper, we illustrate that the Kahn PN network model is really a single-resource model for the purposes of deadlock detection. An original contribution of this paper is the application of a distributed deadlock detection algorithm to the Process Network model. This algorithm detects both local and global deadlock. Finally, we present the design of a DPN implementation that integrates this algorithm with techniques for achieving high-performance. Our implementation is available for download at **http://www.ece.utexas.edu/∼bevans/projects/pn**

7. REFERENCES

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

[2] G. E. Allen and B. L. Evans, "Real-time sonar beamforming on workstations using Process Networks and POSIX threads," in *IEEE Trans. Signal Processing*, Mar. 2000, pp. 921–926.

[3] A. Amar, P. Boulet, J.-L. Dekeyser, and F. Theeuwen, "Distributed Process Networks using half FIFO queues in CORBA," INRIA, Tech. Rep. RR-4765, Mar. 2003. [Online]. Available: http://www.inria.fr/rrrt/rr-4765.html

[4] T. Parks and D. Roberts, "Distributed Process Networks in Java," in *International Workshop on Java for Parallel and Distributed Computing*, Nice, France, Apr. 2003.

[5] J. Vayssière, D. Webb, and A. Wendelborn, "Distributed Process Networks," University of Adelaide, Australia, Tech. Rep. TR 99-03, Oct. 1999, draft. [Online]. Available: http://www.cs.adelaide.edu.au/∼dpn/documents/tr9904.ps

[6] T. M. Parks, "Bounded scheduling of Process Networks," Ph.D. dissertation, University of California at Berkeley, 1995. [Online]. Available: citeseer.ist.psu.edu/parks95bounded.html

[7] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, University of California at Berkeley, 1993. [Online]. Available: citeseer.ist.psu.edu/buck93scheduling.html

[8] M. Geilen and T. Basten, "Requirements on the execution of Kahn Process Networks," in *Programming Languages and Systems, 12th European Symposium on Programming*, vol. 2618. Berlin, Germany: Springer-Verlag, 2003. [Online]. Available: http://www.ics.ele.tue.nl/∼mgeilen/publications/esop03.pdf

[9] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," in *SIAM Journal*, vol. 14, Nov. 1966, pp. 1390–1411.

[10] M. Goel, "Process Networks in Ptolemy II," Master's thesis, University of California at Berkeley, Dec. 1998. [Online]. Available: http://ptolemy.eecs.berkeley.edu/publications/papers/98/PNinPtolemyII/

[11] B. Vaidyanathan, "Artificial deadlock detection and correction in bounded scheduling of Process Networks," Oct. 1999, uT Austin EE 382C-9 Embedded Software Systems Course Project. [Online]. Available: http://www.ece.utexas.edu/∼bevans/courses/ee382c/projects/fall99

[12] R. Stevens, M. Wan, P. Laramie, T. M. Parks, and E. A. Lee, "Implementation of Process Networks in Java," University of California, EECS Dept, Berkeley, CA, Tech. Memo. No. M97/84, 1997. [Online]. Available: http://www.ait.nrl.navy.mil/pgmt/PNpaper.pdf

[13] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink, "Yapi: application modeling for signal processing systems," in *IEEE Int. Design Automation Conference*. ACM Press, 2000, pp. 402–405.

[14] K. M. Chandy, J. Misra, and L. M. Hass., "Distributed deadlock detection," in *ACM Trans. on Comp. Systems*, vol. 1, no. 2, May 1983, pp. 144–156.

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