Distributed Deadlock Detection for Distributed Process Networks

Alex Olson

Abstract

This paper presents a discussion of the Kahn process network (PN) model and the challenges in distributing it onto multiple networked workstations. Compared to a non-distributed PN model, a distributed PN model is more scalable in terms of performance and has significantly lower cost. Although additional distribution-specific challenges exist, existing distributed systems algorithms can solve them. An original contribution of this paper is a distributed process network implementation that performs deadlock detection.

Index Terms

Parallel, Concurrent Programming, Distributed Programming, Models of Computation, Process networks, distributed deadlock detection, Kahn

Contents

I. Context of Research

There are a variety of approaches for high-performance computing. Many of these rely on exploiting the available parallelism in a computation and are highly effective. One can attempt to find parallelism by examining program code, as done by modern compilers & processors. Another way is to initially design the computation with parallelism in mind, by starting with a highly parallelizable model. One such model is Kahn's process network (PN) [1], which involves the idea of parallel processes communicating over one-way channels.

The beauty of Kahn's model lies in its simplicity. It guarantees determinism only based upon the flow of data – not relying on timing, mutual exclusion, or any other synchronization mechanism. The process network model is highly suitable for signal processing applications. For targeting desktop computers, several PN frameworks exist. All of these tend to map processes onto threads. Thus, symmetric-multiprocessor computers (SMP) can be used to achieve high-performance. Recently, Allen used [2] the process network model to perform real-time sonar beamforming on a UNIX workstation. However, SMP desktops are becoming extremely expensive. Currently, the price of multiple single CPU workstations equals the price one SMP workstation, and they offer greater combined performance. In addition, gigabit networking is becoming affordable. It is no wonder the idea of cluster computing is becoming popular. Thus, it is desirable to have a *distributed* process network framework (DPN). Such a framework allows for greater performance and reduced cost over PN implementations on single SMP desktops. Unfortunately, few DPN frameworks exist and they are only in early stages of development.

II. Objectives

The primary objective of this project is to research and design a high-performance distributed process network framework. Such a framework should implement deadlock detection, dynamic process migration, and load balancing. Deadlock detection is necessary to detect termination and/or the condition of a buffer being too small. This will be explained in section III. Dynamic process migration refers to the ability to relocate a process from one workstation to another at runtime. This capability is useful should the need arise to add or delete a workstation from the network at run-time. It is also a highly useful capability for the implementation of dynamic load balancing. With process networks, the behavior of any process is not statically predictable. Thus, dynamic load balancing could be highly beneficial for ensuring maximum performance.

However, the scope of this paper will be limited to a discussion of process networks, problems in distribution, and plans for distributed deadlock detection.

III. Existing Theoretical Work

A. Process Networks

1) Kahn: In 1974, Kahn proposed [1] a determinate model of computation based on data tokens (and their flow). The term *token* is a general term for any unit of data. He suggested that multiple processes executing concurrently and communicating over channels could perform a computation. The only requirements of his channels are that they be reliable FIFO queues but may be unbounded in length. Thus, the queues provide a loose coupling between producers (processes emitting tokens) and consumers (processes receiving tokens).

One can visualize Kahn's model as a group of processes which each consume (read) one or more tokens from one or more inputs and produce (write) one or more tokens to one more or more outputs. If a process attempts to read more tokens than are available on a channel, the process blocks until enough tokens are available. A process only blocks on at most one channel at a time. In addition, a process may not to test a channel for the presence of tokens. However, a process enjoys a great deal of freedom; it can read and/or write tokens at any time. Additionally, a process with multiple inputs and/or multiple outputs is not required to perform a read or write on all ports. As simple as these rules are, they do guarantee determinism. In Kahn's model, global deadlock only occurs when a computation terminates, and local deadlock is impossible. At termination, all processes are blocked on reads and all channels are empty. Another nice feature of this model is that it is completely independent of time. As long as all processes eventually have the opportunity to make progress, it doesn't matter if some processes run faster, receive more CPU time, or if they execute in varying orders. The sequence of tokens produced at all processes will be unaffected. Kahn also guarantees that his model produces complete output, regardless of scheduling. The simplicity and determinacy of process networks makes it an extremely attractive model for distributed computations, especially in heterogeneous environments.

2) Parks: There is one main drawback to Kahn's model – it relies upon unbounded channel capacities. In process networks, it is generally undecidable as to whether or not a set of bounded channel capacities exists. It is also undecidable if a process network will terminate. Both of these properties are because the model is Turing complete [3].

The assumption of unbounded channel capacities makes the model not feasible for real-world implementation. Twenty-one years after Kahn, Parks proposed [4] a bounded scheduling algorithm. The impact of Parks' algorithm is that process networks can execute under bounded memory. Thus, the PN model is no longer a theoretical toy, but a powerful & practical computational model.

Parks' algorithm puts limits on channel capacities (queue sizes). If a process attempts to write a token to a full channel, the process will block until the channel is not full. Parks also distinguishes between two types of deadlocks. 'True' deadlocks are those described in Kahn's model, while 'artificial' deadlocks are caused by bounded channel capacities. Since channel capacities are not restricted in Kahn's model, artificial deadlock is impossible. Parks' algorithm waits until the system reaches global deadlock. If all processes are blocked on reads then true deadlock exists and the computation has terminated. However, if all processes are blocked and at least one process is blocked on a write, then artificial deadlock occurs. Parks' algorithm considers all channels to which one or more blocked processes are writing. Of these, the algorithm increases the capacity of the smallest full channel by any amount. If this increase does not relieve the deadlock, the algorithm repeats. It is interesting to note that if the smallest full channel wasn't that one whose capacity should have been increased, its capacity will get increased until it is no longer the smallest channel or until it is no longer full. Thus, Parks' algorithm does not necessarily find the most optimal set of channel capacities that avoids global deadlock. The algorithm only finds a set of bounded channel capacities when such bounds exist in a system.

3) Geillen & Basten: Parks' algorithm activates only when a PN system reaches global deadlock. However, not all artificial deadlocks result in global deadlock. For example, consider figure 1. Suppose process C only reads from B just once, and then continuously reads from process A. Also assume that processes A & B attempt to write an infinite stream of tokens to all output ports. Under Kahn's model, processes C & D will receive an infinite stream of tokens. However, under Parks' algorithm, the B-C channel will fill up with tokens but it will never cause a global deadlock. Furthermore, process D will only see a finite stream of tokens. Process B will only fire a finite number of times before blocking forever. One definition for 'determinism' is that at most one behavior is possible for a given situation. One may argue the determinacy of

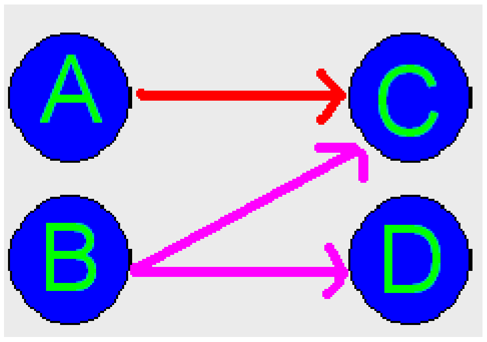

Fig. 1. Process network that does not result in global deadlock

Parks' algorithm compromises the determinacy of Kahn's model. This follows since there exists two possible behaviors (independent of time) for the stream of tokens traveling toward D.

Parks algorithm also fails to produce complete output if the system is composed of two disjoint computations. For example, if the PN graph is composed of two disjoint chains, then deadlock of one chain cannot cause deadlock in the other. Geilen and Basten present a different algorithm [5] for deadlock resolution. Their algorithm is very similar to that of Parks, with one main alteration. They increase the capacity of the smallest channel that is locally deadlocked. They also guarantee their algorithm will find a set of bounded channel capacities when such bounds exist. In terms of the counterexample shown against Parks' algorithm, it could be said that Parks prefers an incomplete & bounded execution over Geilen & Basten's complete but unbounded execution. They also show that no algorithm can guarantee a complete *and* bounded execution for all computations.

B. Problems in Distribution

However, distributing the process network model is not trivial. All of the authors above assume that two processes in a channel have identical views of their channel. That is to say, when a producer places a token in a channel, the consumer knows about it on the next examination of the channel. A similar assumption holds for when a consumer consumes a token. For a PN implementation on a single workstation, this is trivially achievable with the use of shared memory. However, the implementation of process networks on multiple networked workstations makes matters much more complicated. All processes could be blocked on reads or writes, but this would not imply deadlock if there were any messages in transit on any channel. Kahn's ‘channel’ is not really a channel but a shared-memory queue. The difference between his channel and the real world is due to the effects of latency combined with the lack of a global clock.

Furthermore, even load balancing, which an operating system automatically performs on a single-workstation, becomes non-trivial in the distributed case. A single workstation has to eventually give execution time to all running applications. However, this does not hold for a multiple-workstation setup. Here, there arises the question of how to partition the set of all applications across the multiple workstations. This is inherently a difficult problem. Things become worse in heterogeneous environments as one also has to consider the relative processing capabilities of all workstations and also the capacity of the interconnecting communication links.

C. Distributed Systems

Although a discussion of process networks has been completed, the goal of this paper is a *distributed* process network implementation. As mentioned in the previous section, the notion of a ‘channel’ becomes more complicated. Fortunately, research in the field of Distributed Systems provides some solutions. In distributed systems, there is no global clock, no shared memory, and non-instantaneous communication – just as in a distributed process network.

1) Distributed Deadlock Detection: Deadlock detection is a problem in regular PN implementations and is even a tougher problem due to the characteristics of the ‘channel’ mentioned. Fortunately, researchers in the field of Distributed Systems have come up with a variety of deadlock detection algorithms. Some algorithms require a centralized knowledge of the whole system; others do not. One centralized solution is to construct a wait-for-graph of the entire system, and determine if there are any cycles in it. This could be well suited for detecting deadlock on a single workstation, where shared memory exists. However, this is not a particularly good solution for a distributed system with a large number of nodes. One problem is this algorithm may put great stress (caused by a large number of messages) on the communication link between the central node and the rest of the network. In 1984, Mitchell presented an elegant algorithm [6] for distributed deadlock detection. An advantage to this algorithm is that it does not require a node in a system to have complete knowledge of the whole system. Each node only communicates with its neighbors. Furthermore, each node does not store information about the entire system. The algorithm is as follows. Every node has a public and private label; both are non-decreasing. Additionally, no two nodes ever have the same private label. When node

x begins to wait on node y, node x updates its the public label to be $max(x, y) + 1$. When a node discovers that the node it is waiting on has a larger public label than its own, it replaces the value of its public label with the larger one. This algorithm has the effect of circulating successively larger public labels in the reverse order of the corresponding wait-for graph. If a deadlock truly exists, then a node will eventually see its own public label on the process for which it waits. This algorithm also has the nice property that exactly one node will detect a deadlock. This property is important for deadlock resolution in process networks, as it helps to ensure that exactly one channel's capacity is increased. In addition, it is trivial to modify this algorithm so that upon detection of deadlock would reveal the smallest channel capacity.

2) Distributed Mutual Exclusion: Another problem in Distributed Systems is distributed mutual exclusion. This is also a relevant problem for the distributed process network model. Suppose it is desirable for every server to maintain a list of all servers in the network. These lists would be beneficial for implementing load balancing, as it would be undesirable if a load balancing algorithm attempted to relocate a process to a nonexistent server. However, additions and deletions must occur in a controlled manner. Another scenario is the problem of a newly instantiated process on a remote server attempting to connect to a process on a server that is pending removal/suspension. A distributed mutual exclusion algorithm could allow the user's server to effectively command the other servers to 'not touch anything 'until the user's action completed. An efficient algorithm is given by [7]. If one arranges n servers in a tree structure, mutual exclusion is achievable with a latency $O(log(n))$. In this algorithm, each node remembers which immediate neighbor leads to the node has the privilege. The author names this variable HOLDER. In this algorithm, having the privilege means having the right to be in the critical section. Each node also keeps a request queue, however the size of this queue is limited to $1+n$, where n represents the number of neighbors of a node. Like the HOLDER value, the request queue only keeps a list of the immediate neighbors that requested privilege. At any time, the series of the HOLDER values at each node will point to the privileged node. In a similar way, when a node has made a request to enter the critical section, the heads of the request queues will form a path that points to the requesting node. For N nodes in a balanced tree structure, the message complexity of this algorithm is O(log N). Raymond also points out that under heavy demand for the critical section, the performance of this algorithm actually improves to four messages per critical section entry (independent of the number of nodes)! Clearly, this algorithm should perform well for managing modifications to the structure or underlying server pool of a distributed process network.

IV. Existing Implementations

After presenting background on process networks and distributed systems algorithms, we will examine characteristics of existing process network implementations (both distributed and non-distributed).

A few known non-distributed process network implementations are described in [2], [8], and [9]. Only the later two implement deadlock detection. More specifically, they both detect global deadlocks. Both implementations map processes onto threads and use a dedicated thread for deadlock detection. Essentially, they both rely on the shared memory in which threads reside. Thus, techniques used to detect deadlocks in existing non-distributed process network models are not applicable toward a distributed implementation.

In terms of distributed process networks, some implementations are covered in [10], [11], and [12]. These three implementations seem to be the only ones that exist. None detect deadlocks and none support dynamic process migration. Thus, an original contribution of this project will be a DPN framework that provides both.

Plans for the implementation part of this project are as follows:

V. Implementation Plan

The implementation of this project will consist of a distributed process network framework, written in C++ or Java. Processes will communicate over the standard TCP/IP protocol. In addition, the framework will implement deadlock detection using a distributed deadlock algorithm similar to that discussed in this paper. The mutual exclusion algorithm discussed in this paper will regulate changes to the server pool. The deliverables of this project will include a high-performance DPN framework that implements distributed deadlock detection. Such a framework will also be compatible with implementing other capabilities described in this paper.

VI. Conclusion

In conclusion, the process network model is well suited for high performance applications. In today's world, distributing the PN model onto multiple workstations makes sense in terms of cost and performance. However, distributing the model is not trivial. Fortunately some algorithms developed for generalized distributed systems can be applied and provide good solutions to distribution-specific problems. This allows the creation of a high-performance distributed process network.

REFERENCES

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

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

[3] 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

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

[5] 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

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

[7] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," in *IEEE Trans. Comput.*, vol. 7, no. 1, Feb. 1989, pp. 61–77.

[8] 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/

[9] R. Stevens, M. Wan, P. Laramie, T. Parks, and E. Lee, "Implementation of process networks in Java," Tech. Rep., July 1997, draft. [Online]. Available: http://www.ait.nrl.navy.mil/pgmt/PNpaper.pdf

[10] D. W. Julien Vayssière and A. Wendelborn, "Distributed process networks," University of Adelaide, Austrailia, Tech. Rep. TR 99-03, Oct. 1999, draft. [Online]. Available: http://www.cs.adelaide.edu.au/ dpn/documents/tr9904.ps

[11] 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

[12] T. Parks and D. Roberts, "Distributed process networks in Java," presented at the International Workshop on Java for Parallel and Distributed Computing, Nice, France, Apr. 2003.