Distributed Deadlock Detection for Distributed Process Networks

Alex Olson

Embedded Software Systems

Abstract

The distributed process network (DPN) model allows for greater scalability and performance over a non-distributed process network model. This paper presents a distributed deadlock detection algorithm applicable to process networks. The algorithm efficiently detects both local and global deadlocks. In addition, the deadlock detection algorithm has minimal bandwidth and memory requirements. Lastly, the implementation of a high-performance DPN framework that detects deadlocks is presented. Furthermore, this implementation supports the future addition of dynamic process creation/migration (ability to create/relocate processes at run-time).

CONTENTS

I. Introduction

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 and processors. Another approach is to initially design the computation with parallelism in mind, by starting with a highly parallelizable model. One such model is the Kahn process network (PN) model [1], which consists of concurrent processes communicating over one-way channels.

The beauty of the Kahn process network model lies in its simplicity. It guarantees determinism only based upon the flow of data. The process network model is highly suitable for signal processing applications. For targeting desktop computers, several (non-distributed) PN frameworks exist. For these, symmetric-multiprocessor (SMP) workstations can be used to achieve high-performance, as shown in [2]. However, SMP desktops are becoming extremely expensive. Currently, multiple single-CPU workstations may be purchased for the price of a single SMP-workstation, and they offer greater combined performance. Thus, a *distributed* process network framework (DPN) allows for greater performance at a reduced economic cost over PN implementations running on SMP workstations. Unfortunately, few DPN frameworks exist.

Distributing the process network model incurs challenges as channels take on non-zero latencies. Furthermore, the lack of a global clock and shared memory make correct deadlock detection more difficult. The primary objective of this paper is to research and design a high-performance DPN framework that performs distributed deadlock detection. This framework will provide a basis for the future research of dynamic process migration, dynamic load balancing, and other challenges. Deadlock detection is required for process network scheduling algorithms, since ‘artificial deadlocks’ may be introduced by placing bounds on memory usage [3]. An

original contribution of this paper is an algorithm that detects *both* local and global deadlocks in distributed process networks. Furthermore, this algorithm is also applicable to non-distributed process networks.

II. The Process Network Model

A. 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 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). Determinism is guaranteed with blocking reads. A process is blocked if it attempts to read more tokens from a channel than are available. Furthermore, a process is not allowed to test a channel for the presence of tokens. Additionally, it may only attempt to read or write to only one channel at a time. This model is determinate in that the *sequence* of tokens output from any process is only a function of the *sequence* of tokens arriving at its input(s). Thus, the 'correctness' of a computation under this model is not affected by the rates or order in which processes execute.

B. Parks

Kahn's assumption of 'unbounded channel capacities' translates into 'unbounded memory' usage when the process network model is implemented on a computer. Since the process network model is Turing complete, one cannot predict memory requirements statically [4]. In 1995, Parks proposed [3] an algorithm for scheduling process networks 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. These deadlocks are artificial in that sense they arise only from placing bounds on channel capacities. These deadlocks would not have occurred in the original Kahn model. Park's algorithm 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.

C. Geilen and Basten

Geilen and Basten show [5] 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.

D. Allen and Evans

Allen and Evans combined [2] the process network model with Karp and Miller computation graphs. In the resulting model, known as *computational* process networks, a process may *consume* fewer tokens than it *reads*. For example, if a process attempts to read 32 tokens, and 20 are available, it will be blocked until the channel holds 32 tokens. However, it may consume just 1 token upon reading 32. The remaining 31 tokens will be read again on the next read of 32 tokens. This model is highly useful for many computations, such as FIR filters. It allows memory bandwidth to be halved as this model often eliminates the need to copy tokens from the

channel buffer into an application-specific buffer. This model also makes many types of processes memoryless, as their only memory lies in the channel itself. In modern desktop computers, processors run at several times the speed of the memory subsystem. This model is significant as the speed of some computations is bounded by memory bandwidth rather than CPU speed. Artificial deadlock is still possible and either Park's or Geilen and Basten's scheduling algorithms may be applied.

III. Deadlock Detection

A. Previous Work

Since the process network model is Turing complete, deadlock is only detectable at run-time. Deadlock detection has been implemented in a few non-distributed PN implementations [6], [7], [8], [3]. All of these detect only global deadlock. Few distributed PN (DPN) implementations exist: [9], [10], [11]. Of these, none detect deadlocks. What follows in the next section is an original application of an existing deadlock detection algorithm to distributed process networks.

B. Mitchell and Merritt's Algorithm

Kahn's specifications for his process network model include that a process may be blocked on at most one other process at a time. Of the many distributed deadlock detection algorithms, we need only consider the set commonly known as 'single-resource' algorithms. This class of algorithms assumes that a process is waiting on at most one other process. 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 [12]. Although they developed this algorithm for distributed databases, this paper shows a novel application of their algorithm to the distributed process network model.

In their algorithm, each process contains two labels: a public label and a private label. Here, 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. The frequency at which the polling takes place is not important. 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.

IV. Distributed Process Network Framework Design

One goal of this project was to create a high-performance distributed process network implementation. This implementation will provide a basis for future exploration of capabilities such as dynamic process migration (relocating processes at run-time) and dynamic load balancing (instantiating/relocating processes at run-time to equalize server load). For performance reasons the C++ language was chosen. Multi-platform source compatibility is achieved through the use of POSIX functions.

In my design, each process contains two threads. The first is a *computation* thread that performs the desired computation, such as an FIR filter, etc. The second thread is a *communication*

thread. This design is similar to the decoupled access/execute processor architecture [13]. The communication thread is responsible for sending/receiving tokens and performing deadlock detection. Relative to each process, two types of channels are defined: *incoming* channels and *outgoing* channels. Incoming channels are those from which a process reads tokens. Likewise, outgoing channels are those to which a process writes tokens. For each incoming or outgoing channel on which a process communicates, a shared circular queue exists between the two threads. For incoming channels, this queue represents Kahn's channel queue. For outgoing channels, the outgoing channel queue helps to aggregate small tokens into larger groups before transmission.

The TCP protocol is used by each channel for communication as this protocol provides reliable and FIFO transmission of data over networks. In transmitting a single TCP/IP packet over an Ethernet network, there is at least 40 bytes of overhead per packet, due to Ethernet frame headers, IP headers, and TCP headers. Although the outgoing channel queue could be eliminated, this would reduce network bandwidth efficiency to less than 10% if 32 bit integers were used as tokens. This elimination would also increase CPU usage. The Nagle algorithm, present in most TCP implementations, does not provide any performance benefit here.

Instead of operating at the token level, all communications is performed at the byte level. Conceptually, a token is a single byte and processes communicate by sending/receiving multiple tokens at a time. When an outgoing channel establishes a connection to a remote process, it first obtains the number of bytes available in the remote process's incoming channel queue and stores this in a variable named $remoteAvail$. When an outgoing channel transmits x bytes of token data, it decrements $remoteAvail$ by x. After a process consumes data from its incoming channel queue, it sends a *Bytes Consumed* message over the TCP link that contains the number of bytes

consumed. Upon receiving such a message, the outgoing channel increments its $remoteAvail$ by this amount. Thus, $remoteAvail$ represents a lower bound on the number of number of bytes available in the queue of the remote process. The computation thread is blocked if it attempts to read more data than is present in an incoming channel's queue or if it attempts to write more than $remoteAvail$ bytes on an outgoing channel. The communication thread of each process runs once per 5ms. Thus, if a process consumes 100 bytes in 4ms, one *Bytes Consumed* message may be sent. This same mechanism also helps to aggregate tokens before transmission.

For the transmission of data, a lightweight serialization mechanism has also been implemented. The interface is similar to that of Java, but it is more efficient in terms of CPU usage and bandwidth. C++ classes can be serialized by inheriting from a $Serializable$ class and then defining a method for serialization and one for de-serialization. Platform-dependent byte re-ordering is also performed on all basic C/C++ data-types.

To implement deadlock detection, Mitchell and Merritt's algorithm is used. Since this deadlock detection algorithm does not require any arrays or a list of all processes, as in the case of other deadlock detection algorithms [14], this algorithm is compatible with dynamic process creation/deletion/migration. In this implementation, the 'label' is an ordered pair $(count, p)$ of two integers. Initially each the labels of each process are initialized to (pid, pid) where pid is a globally unique identifier for that process. When the computation thread of process X is blocked waiting on process Y, the label of process X becomes $max(X.count, Y.count), X.p)$. For comparison purposes, the label of process X is greater than the label of process Y if $(X.count > Y.count)\ OR\ (X.count = Y.count\ AND\ X.p > Y.p)$. To reduce network bandwidth, the deadlock detection algorithm is only activated if a process's computation thread has been (continuously) blocked for a sufficient period of time, currently set at 1 second. Likewise,

the same interval is also used for polling in the *transmit* step of the algorithm. Deadlock detection data and process token data are sent through the same TCP link. When a process becomes blocked on a read or write, it sends a *Label Request* message. If a process becomes blocked on a read while the needed token data is in transit, the FIFO property of the TCP protocol ensures that the token data will arrive before the label of the remote process. Thus the blocked process will never perform the blocking step of the deadlock detection algorithm if there is data already in transit. Likewise, the same guarantee holds for processes blocked on writes and the arrival of *Bytes Consumed* messages.

An extra optimization that has been implemented is that for an outgoing channel, token data packets are only sent if there is at least one kilobyte of data in the queue. Upon the receipt of a *Label Request* message, any remaining data is sent before the process's public label. This optimization helps to minimize communication protocol overhead and CPU usage.

Experimental tests confirm this DPN implementation detects deadlocks involving all processes blocked on reads, all processes blocked on writes, and artificial deadlocks involving some process blocked on writes and others blocked on reads. In terms of the actual interface to reading and writing tokens, both the 'regular' and 'computational' process network models are supported. The interface for the computational process network style is similar to the implementation of Allen and Evans [2].

V. Performance

Experimental tests show that this framework is capable of handling up to two million tokens per second on an Athlon 1.6 GHz PC. For single byte tokens, the use of queues in the outgoing channels increased performance by a factor of 20 over the same implementation sending tokens directly. Furthermore, the processing time per token appears to be independent of the token size.

Thus the run-time overhead of this framework is negligible if a computation spends significantly more than $0.5\mu s$ CPU time per token. In addition, the token aggregation methods described cause the communication protocol overhead to be negligible.

VI. Conclusion

In conclusion, distributing the process network model offers greater performance at reduced economic cost over a non-distributed model. An original deadlock detection scheme for process networks has been presented that detects both local and global deadlock. This scheme is applicable to both distributed as well as non-distributed process network implementations. Lastly, this paper presents a design of a high-performance distributed process network implementation that performs deadlock detection.

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

[4] 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

[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] 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/

[7] B. Vaidyanathan, "Artificial deadlock detection and correction in bounded scheduling of process networks," Oct. 1999. [Online]. Available: http://www.ece.utexas.edu/˜bevans/courses/ee382c/projects/fall99/index.html

[8] 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

[9] 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

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

[11] 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

[12] 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.

[13] J. Smith, "Decoupled access/execute computer architectures," in *9th Annual Symposium on Computer Architecture*, May 1982, pp. 112–119.

[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.