Final Report

On

IMPLEMENTATION OF PROCESS NETWORK IN JAVA

Arnab Basu
And
Hampapur P. Vijay Kishen

For
EE382C
Embedded Software Systems

May 2000

ABSTRACT

Process networks are networks of sequential processes connected by channels behaving like FIFO queues. These are used in signal and image processing applications that need to run in bounded memory for infinitely long periods of time dealing with possibly infinite streams of data. The requirement to run for long times with limited memory raises concerns about deadlocking and memory requirements. T. Parks suggests an algorithm that provides a way of executing these models in bounded memory, whenever a bounded memory schedule exists. We implemented this algorithm in Java and devised methods to detect and resolve deadlocks.

Introduction

A Process Network (PN) is a directed graph, comprising of a set of nodes (processes) connected by a set of directed arcs (FIFO queues). Each process executes a sequential program. At any given moment this process may read a data token from one of its input queues, or it may write a data token to one of its output queues. The nodes can be viewed as concurrent processes that run concurrently and exchange data over the arcs. This model is well suited for signal processing and image processing applications. Managing the concurrency in these applications becomes a significant part of these applications.

In the first section a brief introduction to the different Process Network models is presented followed by prior work done in this field. In the second section we define our objectives, discuss the design and implementation issues followed by a brief discussion of why Java was chosen and in the last section we tabulate the results obtained and a brief conclusion.

Process Networks

Kahn Process Networks:

In a process network, concurrent processes communicate only through one-way FIFO channels with unbounded capacity. A process in the Kahn model is a mapping from one or more input sequences to one or more output sequences, all processes in this model are monotonic. Each channel carries a possible infinite sequence (a stream) of atomic data objects (tokens). Each token is written (produced) exactly once, and read (consumed) exactly once. Writes to the channels are non-blocking, but reads are blocking. This means that a process that attempts to read from an empty input channel

stalls until the buffer has sufficient token to satisfy the read. The “History” of a channel is defined as the sequence of tokens that have been written to and read from the channel. This model can be proved to be determinate, if the histories of the internal and output channels in the system solely depend on the history of the input channel [1,2]. Even though this model usually does not require infinite queues it does not guarantee bounded memory. In a Kahn PN it is not possible to predict whether the model will terminate in finite time and will be able to execute in finite memory (Turing complete). A deadlock occurs when all processes are (read) blocked trying to read from their respective channels.

Parks Model:

Thomas Parks devised a variation to the Kahn PN called Dataflow process networks [3]. Parks’ model circumvents the problem of infinite queue sizes between nodes by adding certain rules to Kahn’s model. The model he proposed had three properties: a) a process is suspended when attempting to read from an empty queue. b) A process is suspended when attempting to write to a full queue (write block) and c) on artificial deadlock, increase the capacity of the smallest full queue until its producer can fire. If we execute a PN program satisfying the above requirements using dynamic scheduling, there could be two kinds of deadlocks: a) the execution of the PN program stops if all the processes are suspended reading from empty queue resulting in a true deadlock. b) The execution stops because some processes are suspended attempting to write to full queues resulting in an artificial deadlock. Artificial deadlocks can be resolved by the third rule Parks added to Kahn PN. Parks' model guarantees bounded execution of a PN program if one exists [3,5].

Prior Work

The process networks domain in Ptolemy II 0.4 [10] models a system as a network of sequential processes, implemented as Java threads, which communicate using one-way FIFO queues. It implements Parks bounded scheduling and handles detection of artificial deadlock. However, it employs a separate Java thread that handles the deadlocks and resolves them when they arise according to Parks scheduling policies. It differs from our implementation in that this dedicated Java thread checks at regular intervals for artificial deadlock in all the threads.

Real time sonar beamforming [4] using Process Networks and POSIX threads implemented in C++ was also studied for its thread synchronization and deadlock resolution methods.

Objectives

The objective of this project is to thoroughly study the existing implementations (both Ptolemy II and computational PN) of PN. Realize the limitations of the current implementations. Design and implement a PN framework and two different policies for deadlock detection in Java keeping in mind the limitations of the existing implementations. Profile our implementation to understand the advantages and shortcomings of this model.

Design and Implementation

The design of the Process Network framework was done so as to enable other users to simulate their network in the PN model quickly and effectively. The design is object oriented and event driven. The main classes are PN, PNQueue, PNGlobals, and

PNBlockedEvent and the interfaces are PNActor and PNBlockedEventListener. The Class diagram is as shown in figure 1.

The PN class is the main class that creates and initializes the actors and their associated queues. This class also implements the PNBlockedEventListener interface that informs it whenever any actor is blocked. This can be used for deadlock detection (when all the actors are read/write blocked) and also for resolving the deadlock. The main innovations in this are the two methods of resolving write blocks, which are, 1) Generous – Here the size of the queue is increased as soon as any actor is write blocked. The size by which the queue is increased can be varied. 2) Parks – Here the size of the queue needing the least amount is incremented by that amount when all the actors are blocked (provided at least one of them is write blocked). The one that is chosen depends on the flag set in the PNGlobals class.

The PNActor interface is derived from the *runnable* interface, which allows it to be run in a Java thread. It also has methods, which allows the PN class to control the actors. The *init()* method allows the PN class to specify to the actor its input and output queues. The *getInputQueue()* and *getOutputQueue()* methods of the PNActor interface allow the PN class access to the actors input and output queues.

The PNQueue class is the class that enables the actors to interact with each other. This has functions like *enqueue()* and *dequeue()* which allow the actors to insert and remove data from the queue. If the capacity of the queue is reached, while enqueueing the queue throws a “Write Blocked Event” which the PN class handles. The *increaseSize()* method of the PNQueue class allows the main class to increase the size of the queue by the required size.

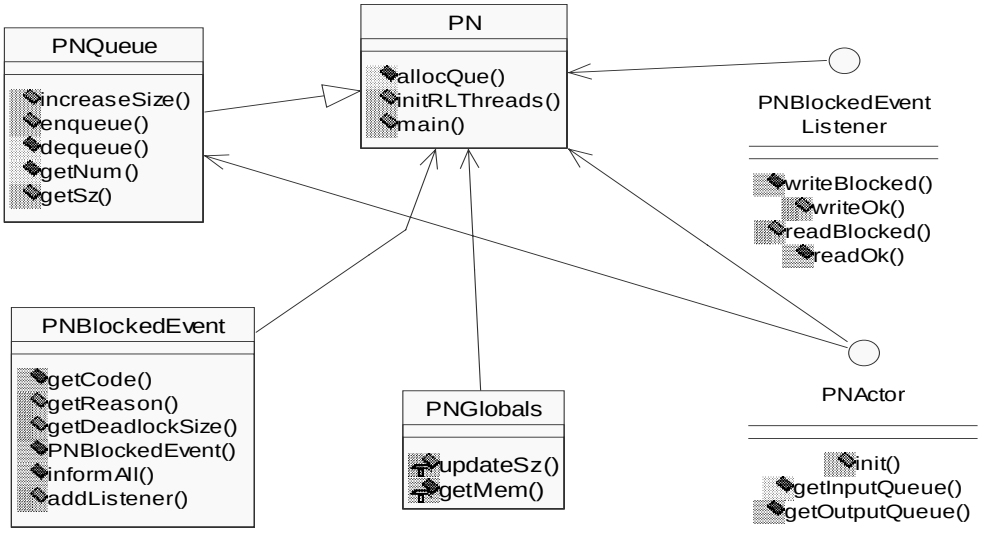

Figure 1: Class Diagram for the PN Framework

The PNGlobals class is the class that contains all the constants and flags required by the framework. The boolean *PARKS* allows the framework to either use the deadlock resolution technique of Parks or the generous method where there are no upper limits on the capacity of the queues. The *updateSz()* method of this class allows the queue objects to log their sizes on initiation and completion and the *getMem()* method allows the main class to print the initial and final sizes.

The PN class implements the PNBlockedEventListener interface in order to be able to receive the write, read blocked and also the write and read ok events. This is useful for implementing Parks' algorithm of increasing the size of the smallest queue when all the processes are blocked without the overhead of checking the threads at regular intervals for blocks.

Java

The implementation of process networks is helped considerably by a language that allows us to map processes into threads and also has well defined stream functionality. Java [7] was chosen to implement this because

- It is object-oriented, strongly typed.
- It supports multiple threads and is relatively easy to use.
- Its source code is platform independent (behavior on different platforms depends on the implementation of the language).
- It provides Exception handling, runtime checking and automatic garbage collection.

The Embedded Java [8] edition allows developers to code embedded applications using the language. Developers can then configure the core classes in order to come up with a smaller footprint JDK core to be shipped with the application. This helps in reducing both the development costs and the time to market.

Results

The Framework was tested with actors taken from the Computational Process Network source [9] and also the Ptolemy II PN demos [10].

Extensive tests were carried out with the run length encoding and decoding example (figure 2) of Ptolemy II (version 0.2) since it was more computationally intensive compared to the injector, copier and verifier actors from the CPN source [9]. The framework was executed using both Parks' Algorithm and also the generous method on an NT machine[1]. The results of the executions are as shown in figure 3 below.

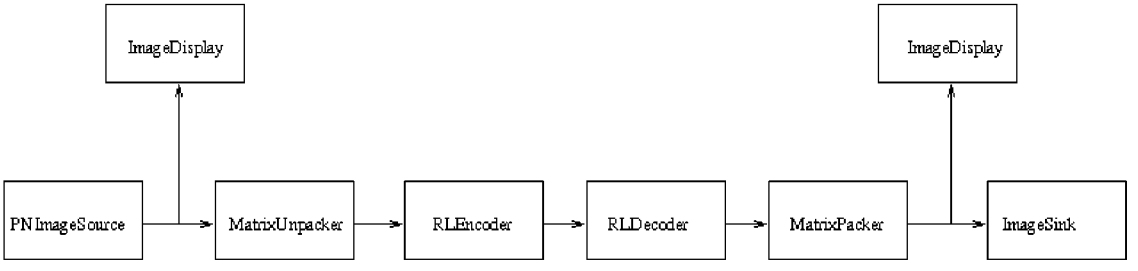

Figure 2: Actors used in the test framework.

[1] Pentium III with Windows NT 4.0, 128 MB RAM.

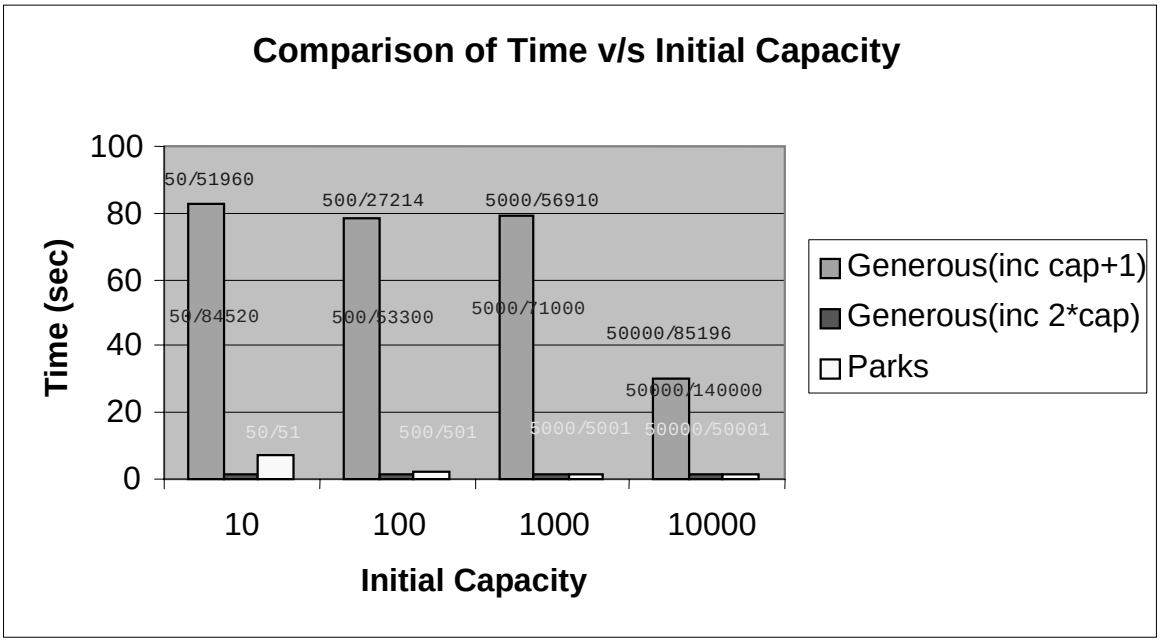

Figure 3: Comparison of Time v/s Initial Capacities of Deadlock resolution Algorithms.

The initial and final queue sizes are also shown, to help us understand the memory tradeoff between the two algorithms. From the figure it is evident that if the queue sizes are incremented in small sizes then the overhead to do this affects the execution time drastically. The execution time of the two methods is comparable when the capacities of the queue, in the generous method, are increased to twice their original size. The memory consumption on the queues, though, is huge in the generous method as compared to the Parks method.

Another factor contributing to the speedup with the Parks algorithm is maybe because there is a lot of blocking going on between the actors which allows the processor to schedule the other threads and thereby increase the overall speed. For this particular setup the occurrence of artificial deadlock was rare but still occurred once or twice per execution as is evident by the growth of the queues in the figure.

Conclusion

We designed and implemented a process networks framework in Java. We implement a policy that uses multiple threads with blocking reads and works correctly regardless of the scheduling algorithms used for the threads. To detect and resolve deadlocks (both artificial and true) we implemented two different policies. We tested the implementation of the framework and deadlock detection for different open source actors and on different platforms and profiled the performance of the policies.

References

[1] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", *Proceedings of International Federation for Information Processing Congress 74*, pp. 471-475, North Holland Publishing Co., Aug 1974.

[2] R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", *SIAM Journal*, Vol. 14, pp. 1390 – 1411, Nov 1966.

[3] E.A. Lee and T.M. Parks, "Dataflow Process Networks", *Proceedings of the IEEE*, Vol. 83 No. 5, pp. 773-801, May 1995.

[4] G. E. Allen and B. L. Evans, "Real-Time Sonar Beamforming on Workstations Using Process Networks and POSIX Threads", *IEEE Transactions on Signal Processing*, pp. 921-926, March 2000 CA.

[5] T. Parks, "*Bounded Scheduling of Process Networks*", Ph.D Thesis, University of California, Berkeley, CA 94720, Dec 1995.

[6] Richard S, Stevens, Marlene Wan, Peggy Laramie, T M. Parks and E.A. Lee, "Implementation of Process Networks in Java", draft 10 July 1997. http://www.cs.adelaide.edu.au/users/darren/research/literature/

[7] G. Cornell and C. Horstmann, "*Core Java*", 1996, Prentice Hall.

[8] Embedded Java, http://java.sun.com/products/embeddedjava/overview.html

[9] Computational Process Networks, http://www.ece.utexas.edu/~allen/CPNSourceCode/index.html

[10] Ptolemy II, http://ptolemy.berkeley.edu/ptolemyII/index.htm