An Implementation of Process Networks in Java

by

Arnab Basu

and

H.P. Vijay Kishen

Literature Survey

Embedded Software Systems (EE382C)

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. In general, the termination and boundedness of process networks are not decidable in finite time. Parks suggests an algorithm that provides a way of executing these models in bounded memory, if a bounded memory schedule exists. The objectives of this literature survey are to study the various Process Network models in existence and to implement a Process Network model in Java, including Parks' bounded memory scheduler.

Introduction

Process Networks is a determinate, concurrent computational model. The PN model, a network of nodes interconnected via arcs, is well suited for implementing computation intensive, real time applications typically present in signal and image processing systems. This class of applications involves computing number of partial results on infinite streams of data in near real time. Managing concurrency in these applications becomes a significant part of these applications.

The Java language provides threads, which are concurrent sequential programs that can share data, precisely to deal with such applications. However, programming with threads in their raw form can easily lead to errors that are difficult to diagnose. In particular, the Java language does not define precisely how threads are scheduled. This is dependent on the implementation. Consequently, writing multithreaded applications that behave identically across multiple implementations requires painstaking care and attention to detail.

In the first section a brief introduction to the different Process Network (PN) models is given, in the second section reasons for choosing Java as the implementation language are discussed and in the last section a brief discussion of the problems that this implementation will face are presented.

Process Networks

The concurrent processes in a PN communicate using unidirectional FIFO queues. Each FIFO queue provides storage for a node's output and input for the downstream node. Using the syntax of directed multigraphs, each node represents a process and each edge represents a FIFO queue. When a process attempts to read an empty queue, the

process blocks. This guarantees determinacy [1], provided that the processes are inherently determinate and do not have any shared or global variables that are being accessed by other processes that might introduce indeterminacy into the model. Schedulability and boundedness of Process Networks are undecidable.

The first PN model by Kahn [1] has a possibility of infinite FIFO queue sizes between arcs. In this model, the functions are treated as functions that map every input sequence onto an output sequence. The processes in this model are *Monotonic*, functions which move in only one direction as the input increases. As a consequence, the nodes do not need to have all of the input ready before they start computing the output. 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 proven to be determinate, if the histories of the internal and output channels in the system solely depend on the history of the input channel. 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. When deadlock occurs i.e. when all nodes are blocked while trying to read from their respective channels, it happens because of the PN program and is not related to any schedule chosen to execute the program.

Another variation of process networks called Dataflow process networks [2], circumvents the problem of infinite queue sizes between nodes by adding certain rules to Kahn's model. The following rules when added will yield a bounded schedule if one exists:

- Block when attempting to write to a full queue.
- Block when attempting to read from an empty queue.

- When deadlock (due to blocking writes) occurs increase the size of the smallest size queue so that the deadlock is removed, i.e. the producer associated with this arc can fire.

The first constraint translates into an initial capacity being assigned to every FIFO queue in the graph. This has the possibility of introducing *Artificial Deadlock* into the model. This occurs when the nodes in the graph are blocked due to writes to the full queues. The third rule is to take care of such scenarios whereby the size of the smallest queue is increased so that deadlock is removed. The two requirements added by Parks for the dynamic scheduler associated with this kind of dataflow model follow:

a) *Complete Execution* – the scheduler should implement a complete execution of the process network program. If the program is non-terminating, then it should be executed forever without terminating. and

b) *Bounded Execution* – the scheduler should if possible execute the process network program so that only a bounded number of tokens ever accumulate on any of the communication channels. Whenever there is a conflict between the two constraints the first rule takes precedence over the second.

These constraints ensure that this model will always find a bounded memory schedule if one exists for the given graph. By restricting the type of dataflow actors to those that have predictable token consumption and production patterns, it becomes possible to perform static, off-line scheduling and to bound the memory required to implement the communication channel buffers.

Computation graphs [3] are a model of parallel computation similar to process networks in which the model is depicted by a directed graph consisting of nodes and arcs

between these nodes. Each node has a single valued function associated with it and each arc has a single source and sink node at each end, which populate or remove data from the arc correspondingly. The source and sink functions are also termed as *Producer* and *Consumer* nodes. In this graph, each arc has a set of parameters associated with it, which have the following interpretation,

a) A_P – The number of tokens that are initially present on the arc.

b) U_p – The number of tokens that are added to the arc when the source function executes.

c) W_p – The number of tokens removed from the arc when the sink function associated with this arc executes.

d) T_p – A threshold that specifies the minimum number of tokens that must be present on the arc before the sink function can be fired. Clearly $T_p \geq W_p$.

Due to the restrictions placed on the computation model Karp and Miller are able to prove that under certain conditions strongly connected subgraphs in the graph are self-terminating and also that the queue lengths for nodes in this subgraph are bounded. It is also proved that irrespective of the sequence of firing of the nodes the data tokens produced on the internal and output queues are the same for every valid sequence of execution provided that the initial data remains the same.

Other types of models are variations on the models described above. The Synchronous dataflow (SDF) [4] models are a variation of the Computation Graph model where the threshold is equal to the number of tokens removed from the arc when the sink actor is executed ($T_p = W_p$). In this model, a static schedule can be developed for executing the graph over and over again even for an infinite stream of tokens. Boolean

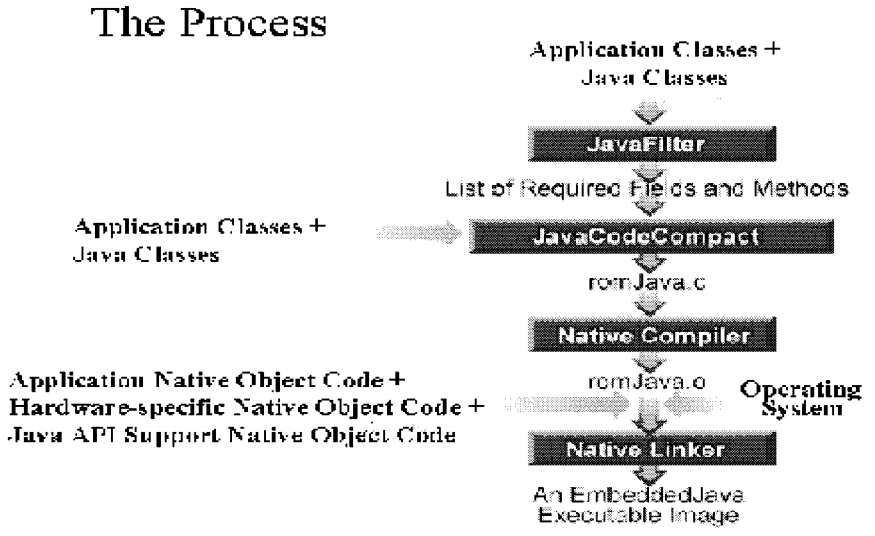
dataflow (BDF) is a modified kind of synchronous dataflow graph where the node can have an additional input termed as the control input which allows the node the liberty of conditional token consumption and production. Because of this addition the ability to analyze and determine a schedule statically is lost and a runtime scheduler is needed to sequence the firings for this graph. Dynamic dataflow is a more generic form of BDF where each node can have many control inputs based on which the node can vary its firing rule. This also requires the use of a dynamic scheduler for sequencing the firings of the graph.

Java

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

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

The Embedded Java 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 time to market.



Another advantage is that there are also quite a lot of tools that help the developer in configuring the classes and also in developing the intermediate C code which can then be compiled by the Cross Compiler that comes along with the chip that the application is targeting. The picture (taken from the Java website [10]) best describes the process.

Implementation

The basic structure of the implementation would involve a main thread that initiates the other threads and then takes on the role of a deadlock resolving thread, either running on low priority or being event driven. The number of threads could be read either from file or via user input.

The problem of bounded memory scheduling of graphs is a problem that most schedulers have to address, since memory is very limited on embedded systems. Parks' method of assigning capacities works well after an initial period where the model is readjusting the various capacities on its queues as deadlocks occur and is continuously

handled. This is a feasible solution as is shown by the development of a number of signal processing applications based on this like the “Real-time sonar Beamformer” [11].

In this implementation Parks’ method is used for scheduling the entire model. The scheduler chosen for the model can be either demand-driven or data-driven. Whenever deadlocks occur the capacities are increased by a main thread, which is activated by deadlock. The exact time at which the deadlock resolving thread is activated can also be varied between activating it when the first blocking operation is reached or when all the processes are blocked. Another factor here is that the capacities of queues are increased only if that queue is making a source actor block. That is causing a write block. With this there can be three different scenarios,

a) The process network can execute indefinitely in bounded memory. This scheduling approach then achieves this schedule within bounded memory
b) The PN cannot execute in bounded memory, hence there will be intermittent artificial deadlocks occurring that are resolved by increasing the queue sizes. The PN will run forever in unbounded memory
c) The PN terminates either because of true deadlock or because it is a terminating PN.

The scheduler in the Java language will handle the scheduling of all the threads in this implementation. No static analysis of the processes will be performed.

Another problem that is to be resolved is that of detecting deadlocks in the model. The deadlock occurs when all the processes are blocked. This can be detected in two ways, by either using an event, handled in the deadlock resolving thread, thrown by the last blocked process or by allowing this thread to run in a low priority and allowing it to

check the state of all the threads when it is executed by the scheduler. Another issue is that in the first solution above how will the process be able to detect that it is the last executing process and appropriately throw the event. This obviously requires more overhead to store the number of executing and blocked processes.

The implications of which method is to be chosen will depend on the cost of the operations and also the program and data memory overhead of both.

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

[3] R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", *SIAM Journal*, vol. 14, no. 6, 1390 – 1411, Nov 1966.

[4] S.S Bhattacharya, P.K.Murthy and E.A Lee, "*Software Synthesis from Dataflow Graphs*", Kluwer academic press, Norwell, MA, ISBN 0-7923-9722-3,1996.

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

[7] K. Arnold and J. Gosling, "*The Java Programming Language*", 1996, Addison Wesley.

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

[9] 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/

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

[11] 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.