A Study on Process Networks

by

Basu Vaidyanathan

Term Project

Embedded Software Systems

(EE382C)

The University of Texas at Austin

October 1999

ABSTRACT

Algorithms for digital signal processing are often described by directed graphs in which the nodes represent the computational units, which are interconnected by arcs that represent sequences of data values. This kind of formal modeling helps to resolve the hard to find problems like, deadlock and determinate execution. A process network model is one model of computation where concurrent processes interact through one-way first-in first-out (FIFO) queues. This is a natural model for signal processing systems that deal with infinite streams of data values, to realize concurrent processing of functional parallelism. Though true real-time measurements are difficult in the simulation of digital signal processing applications, process networks computation model helps develop a prototype on workstations that significantly reduces cost and development time over an equivalent hardware implementation. A real-time sonar beamformer[4] has been successfully simulated in software using process network model on multi-processor Unix workstations interfacing with POSIX light-weight threads library. Ptolemy is another simulation and prototyping environment that uses process network model. In this study, we investigate the definition of process network model and its various restricted versions in detail.

PROCESS NETWORKS

A process network can be thought of as a set of Turing machines connected by a unidirectional tapes, where each machine has its own working tape. It is well known that there is no algorithm that runs in finite time that can determine whether a turing machine will halt or not. Unlike scientific applications, signal processing applications that well fit into process network model, execute forever consuming infinite amount of data samples. Hence the question whether a process network will run forever or terminate because of a bad scheduling decision, is undecidable since halting problem is undecidable. Even the question of executing process network forever with bounded buffering on FIFO queues is undecidable..

KAHN PROCESS NETWORKS

Kahn process networks is a computation model where each process produce data elements called tokens on FIFO queues of infinite length. These tokens are consumed by a waiting destination process. In this model, the execution of the process is suspended if it attempts to consume data from an empty queue. A process may not test for presence or absence of data. At any point, a process is either enabled or blocked waiting for data. A process cannot wait for data from one queue or another.

We list some basic definitions before we present various restricted computation models of process networks.

Execution Order

The execution order is the order of read and write operations in a process network which can either be sequential or concurrent. In general, there will be many execution orders possible that clearly satisfy the restrictions imposed by the processes and the communication layer. The FIFO semantics and write-before-read are some restrictions that are known before execution where some others imposed by processes are not known.

Termination

Termination is completely determined by the definition of the process network and does not depend on the execution order. For a process network program that has finite number of tokens on all the queues, the program must terminate. If at-least one process produces infinite amount of tokens on any queue, then the program never terminates.

Boundedness

Depending on the choice of the execution order, the number of unconsumed tokens on the FIFO queue can accumulate for a process network program running forever. A process network is strictly bounded if the number of unconsumed tokens on *every* FIFO queue is bounded for *any* complete execution of the program. A process network is bounded if the number of unconsumed tokens on *every* FIFO queue is bounded for *at-least* one complete execution of the program.

In Kahn process network, the order in which tokens are produced on the FIFO queues do not depend on the execution order. Systems that follow Kahn's

model are determinate. This allows a process network program to be executed sequentially or concurrently with the same outcome.

RESTRICTED MODELS OF PROCESS NETWORKS

Dataflow is a model of computation that has close correspondence to process networks. The arcs of the graph in dataflow model correspond to FIFO queues and the nodes correspond to actors. Instead of using Kahn's model of blocking read semantics, dataflow actors have firing rules that specify what tokens must be available on arcs for the actor to fire. A process can be formed from a repeated firings of a dataflow actor so that infinite stream of tokens can be operated on.

Computation graphs[2] are a model of parallel computation similar to process networks developed by Karp and Miller. This determinate model is represented by directed graph containing finite set of nodes and arcs. The model defines four non-negative integers A, U, W and T associated with each arc: A is the number of data tokens initially present on the arc, U is the number of data tokens produced on the arc on each firing, W is the number of data tokens consumed from the arc on each firing, and T is the threshold minimum number of data tokens that must be present on the arc before the consumer actor is fired, where T is greater than or equal to W. Providing an upper bound T on the number of tokens on an arc, before a consumer can fire, determine the possible execution sequences. Due to the restrictions placed on the computation model, Karp and Miller are able to give necessary and sufficient conditions to decide on the questions of termination and boundedness for this model.

Synchronous Dataflow[1] is a special case of computation graphs where T is equal to W for all arcs in the graph. Because the number for tokens consumed and produced by an actor is constant for each firing, constructing a static finite schedule that can then be periodically repeated is possible for infinite stream of input data tokens. A balance equation for a synchronous dataflow program is a finite firings of all the actors (without considering initial tokens on the arcs) for which the net production and consumption of tokens are equal. A *complete cycle* is a sequence of actor firings that returns the program to its original state and the accumulation of tokens on any queue is bounded in a complete cycle. So, it is now possible to repeat the execution of the complete cycle indefinitely with only a bounded number of unconsumed data tokens accumulated on the arcs. Depending on the number of initial tokens on each arc, even if we have a balance equations, tokens may accumulate indefinitely if the program is executed forever. It is also possible for the program to deadlock even if we have a balance equations, but not having enough initial tokens on the arcs in the directed cycle. So, the existence of a complete cycle allows a program to execute forever with bounded buffer sizes. The balance equations specify the number of actor firings in a complete cycle. Finding a non-trivial solution to balance equations is necessary but not sufficient for a complete cycle to exist and we need initial tokens on the arcs to determine the existence of a complete cycle. The existence of a complete cycle is only a sufficient condition to determine if the program has a deadlock situation.

Boolean Dataflow is an extension of synchronous dataflow that allows conditional token consumption and production. By adding two simple control actors called switch and select (de-multiplexer and multiplexer) the boolean dataflow program can add conditional constructs such as if-then-else and do-while loops. Copying of data token from appropriate input to the appropriate output arc is controlled by the boolean value of the control token read by the switch or select control actor. As in synchronous dataflow, find a solution to balance equation is necessary but not sufficient to determine the existence of a complete cycle. Because the boolean values of the control token can only be determined at run-time, it is impossible to decide on the questions of termination and boundedness of a boolean dataflow program. However, a quasi-static scheduling is possible by clustering the nodes of a boolean dataflow graph and transforming it to a synchronous dataflow graph.

Dynamic Dataflow model is an extension of Boolean data flow model that obeys boolean dataflow semantics with one additional variation. The control actors can read multiple token values and the data actors can be fired conditionally based on the control token values read. So, dynamic scheduling is a must for this model that determines firing of actors at run-time. A dynamic scheduler for process networks must satisfy two requirements:

Complete Execution: If the program based on Kahn's model is non-terminating, then it should be executed forever without terminating.

Bounded Execution: If possible, the schedule must execute a program so that only a bounded number of tokens ever accumulate on any of the FIFO queues.

When there is a conflict, the first requirement is always preferred. This means that for unbounded programs that require unbounded buffering or tokens, complete unbounded execution is preferred to a partial unbounded execution.

Dynamic scheduling can be classified as data-driven(eager execution), demand-driver(lazy execution) or some combination of the two. An eager execution is one where a process is activated as soon as enough data is available which satisfies the first requirement and therefore this policy results in complete execution. For strictly bounded programs, the eager execution scheduling always performs complete execution with bounded memory. For bounded programs and unbounded programs, the second requirement is violated and so complete unbounded execution is preferred. A lazy execution model developed by Kahn and MacQueen, is one where a process is activated only if the consumer process does not already has sufficient tokens available on the queue. Thus in this model, unnecessary production of tokens is avoided. Also, there is never more than one active process at any time. Unbounded token accumulation is still possible in this model due to the presence of multiple data sinks. For example, if an unequal number of demands arrive for the branches of a fork, then tokens will accumulate at the input of one of the gate operators.

BOUNDED SCHEDULING OF PROCESS NETWORKS

Termination and boundedness are undecidable for process network programs. The process network programs can be classified as terminating, non-terminating, strictly bounded, bounded and unbounded programs. A scheduling policy must have a reasonable behavior for all types of programs. Thomas Parks

devised a scheduling policy[3] that simultaneously satisfies both requirements of a dynamic scheduler and provides a desired behavior for all types of programs. The model he proposed for bounded scheduling of process networks has 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 and c) on artificial deadlock, increase the capacity of the smallest full queue until its producer can fire. Only the read and write operations of Khan process networks program have to be modified to satisfy all the properties mentioned above.

If a program is bounded, then there exists a finite least upper bound and an execution order such that the size of each queue size never exceeds that upper bound. We start the program with an initial estimate on the queue size less than upper bound, such that the program is strictly bounded by the initial bound. We then execute the program satisfying the first requirement using some dynamic scheduling policy. There can be two outcomes: a) the execution of the program stops if and only all the processes are suspended reading from empty queues resulting in a *true deadlock*. b) the execution may also stop because some processes are suspended attempting to write to full queues resulting in an *artificial deadlock*. If it is true deadlock situation, then the program has terminated. If it is artificial deadlock situation, then we increase the initial bound to a new bound less than the upper bound, and continue execution. This is repeated for a steady forward progress of the process network program. If the program is bounded, the new bound will exceed the upper bound and we will be able to execute the program forever with bounded queue length, satisfying both requirements

simultaneously. If the program is unbounded, execution repeatedly stops due to artificial deadlock, and we increase the queue size repeatedly without limit. This is the case where we prefer the first requirement to be satisfied and avoid having partial bounded execution.

FUTURE WORK

The objective of future work is to realize an implementation of dynamic artificial deadlock detection and correction in T. Parks's bounded scheduling of process networks model. The detection and correction of artificial deadlock will be an enhancement to the existing C++ implementation of process networks framework that interfaces with pthread library and runs on SUN Solaris OS. This existing code was developed by Greg Allen of ARL at the University of Texas at Austin.

REFERENCES

[1] Shuvra S. Bhattacharyya, Praveen. K. Murthy, Edward A. Lee, Software Synthesis from Dataflow Graphs, Kluwer Academic Publications, ISBN 0-7923-9722-3, 1996.

[2] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," SIAM Journal of Applied Math, vol. 14, No. 6, November, 1966.

[3] T. M. Parks, "Bounded Scheduling of Process Networks," Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California, Berkeley, CA 94720, December 1995.

[4] G. Allen, B. Evans, and D. Schanbacher, "Real-time Sonar Beamforming on a Unix Workstation Using Process Networks and POSIX Threads," Proceedings of the 32nd Asilomar Conference on Signals, Systems & Computers, pp. 1725-1729, November, 1998.