The Dissertation Committee for Gregory Eugene Allen
certifies that this is the approved version of the following dissertation:

Computational Process Networks: A Model and Framework for High-Throughput Signal Processing

Committee:

Brian L. Evans, Supervisor

James C. Browne

Craig M. Chase

Lizy K. John

Charles M. Loeffler

Computational Process Networks: A Model and Framework for High-Throughput Signal Processing

by

Gregory Eugene Allen, B.S.E.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2011

Dedicated to my wife, Mariana, and my children:

Sabrina, Valerie, Samantha, and Nathan.

Acknowledgments

I would like to acknowledge the Independent Research and Development program at Applied Research Laboratories: The University of Texas at Austin. I also acknowledge the staff at ARL for the encouragement and support I have received throughout this project.

I would like to acknowledge my parents and my extended family for instilling in me the value of education.

I also would like to acknowledge Brian Evans for his patience, mentorship, and friendship.

Computational Process Networks: A Model and Framework for High-Throughput Signal Processing

Publication No. ____________

Gregory Eugene Allen, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Brian L. Evans

Many signal and image processing systems for high-throughput, high-performance applications require concurrent implementations in order to realize desired performance. Developing software for concurrent systems is widely acknowledged to be difficult, with common industry practice leaving the burden of preventing concurrency problems on the programmer.

The Kahn Process Network model provides the mathematically provable property of determinism of a program result regardless of the execution order of its processes, including concurrent execution. This model is also natural for describing streams of data samples in a signal processing system, where processes transform streams from one data type to another. However, a Kahn Process Network may require infinite memory to execute.

I present the dynamic distributed deadlock detection and resolution (D4R) algorithm, which permits execution of Process Networks in bounded memory if it is possible. It detects local deadlocks in a Process Network, determines whether the deadlock can be resolved and, if so, identifies the process that must take action to resolve the deadlock.

I propose the Computational Process Network (CPN) model which is based on the formalisms of Kahn's PN model, but with enhancements that are designed to make it efficiently implementable. These enhancements include multi-token transactions to reduce execution overhead, multi-channel queues for multi-dimensional synchronous data, zero-copy semantics, and consumer and producer firing thresholds for queues. Firing thresholds enable memoryless computation of sliding window algorithms, which are common in signal processing systems. I show that the Computational Process Network model preserves the formal properties of Process Networks, while reducing the operations required to implement sliding window algorithms on continuous streams of data.

I also present a high-throughput software framework that implements the Computational Process Network model using C++, and which maps naturally onto distributed targets. This framework uses POSIX threads, and can exploit parallelism in both multi-core and distributed systems.

Finally, I present case studies to exercise this framework and demonstrate its performance and utility. The final case study is a three-dimensional circular convolution sonar beamformer and replica correlator, which demonstrates the high throughput and scalability of a real-time signal processing algorithm using the CPN model and framework.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In high-performance computing, concurrent implementations are necessary in order to meet desired execution performance targets. Two common forms of concurrency are found in multi-core processors and clusters of computers. Significant parallel hardware is often available in modern multi-core computers, but effectively programming with threads to take advantage of this concurrency is notoriously difficult [3][4]. Another common approach for achieving parallelism is distributed systems, such as a cluster of computers. These clusters can achieve considerably increased computing performance, but require copying shared data across a network between the individual compute nodes. Programming these systems is also difficult, as they suffer many of the same concurrency problems as multi-core systems.

One of the difficulties of developing for concurrent systems is avoiding deadlock. Deadlock occurs when multiple processes are waiting for each other, but never proceed. This is a commonly encountered problem in concurrent programming models that use mutual exclusion, such as threads. A desirable property of a concurrent model of computation is the ability to predict, prevent, and detect deadlock.

Other desirable properties of a concurrent model of computation include determinism, scalability, and boundedness. Determinism in this context means that a modeled program will produce the same results every time that it is executed. One of the difficulties of the thread programming model is that insufficient locking of resources shared between threads leads to race conditions and can violate determinism. A model that can take full advantage of increasingly available parallel hardware is said to be scalable. The commonly used barrier synchronization model, for example, has limits to its scalability because there is no concurrency at each barrier event. In thread programming, too much locking of shared resources reduces scalability and exploitable concurrency. Boundedness is the ability of the model to execute within a finite amount of memory, and is obviously important for implementation.

Composability is another desirable property of a concurrent model of computation. Composable components are modular and self-contained, and their behavior can be analyzed and tested in isolation. Composable components can be combined with clustering and hierarchy to create larger, more complex systems.

A potential solution for effectively developing software for hard-to-program parallel systems is the application of formal models of concurrent computation. A *formal* model is one that can be mathematically proven to have certain properties. Not every model will have all of the desirable properties, but some properties are guaranteed to be true. Application of formal models of computation with the listed desirable properties can allow rapid development of determinate, scalable systems built from composable components. This can allow the developer to focus on the task at hand, with less concern about the complications of concurrent and distributed systems.

This dissertation proposes a new formal model of computation to represent common signal processing algorithms. It shows how this new model is an improvement over existing models for this type of algorithm. This dissertation also provides an implementation of this new model, including case studies. It also provides a dynamic distributed deadlock detection and resolution algorithm for the new model as well as previously existing models.

Section 1.1 provides a very high-level view of modern computer system architectures. Section 1.2 discusses common practices for programming of concurrent systems. Section 1.3 discusses common problems encountered with these approaches for programming concurrent systems. Section 1.4 provides background on formal models for concurrency that can be used to address these difficulties. Section 1.5 introduces the proposed Computational Process Networks model. Section 1.6 contains the thesis statement and expected contributions. Section 1.7 provides a list of acronym definitions and concludes this introductory chapter.

1.1 Computer System Architecture

This section describes modern computer system architectures at a high level in order to provide a foundation for concepts that will be discussed in later sections and chapters.

In a modern operating system, a *process* contains information for a single running program, which includes executable code, private data, and processor state, as well as other resources provided by the operating system. Modern operating systems are multi-tasking; i.e. they can run multiple processes. These processes are typically given a period of time, or *time slice*, to execute and then preempted so that another processes may run. Switching execution from one process to another is called a context switch. A scheduler algorithmically decides which processes will execute and in which order.

Systems with multiple identical processors commonly use *symmetric multiprocessing* [5] (SMP) architectures in which the processors are connected to a common main memory. This permits multiple processes to be executed concurrently, and the ability to balance a computing workload between the available processors. It also permits rapid and efficient exchange of data between the processors.

Modern operating systems also have protected memory, in that a process can access only the memory that the operating system allocates to it. Even though processes (and sometimes multiple processors) share the same physical memory, this prevents the processors from interfering with each other or with the operating system. These systems also commonly have virtual memory. This gives processes the appearance of a particular memory organization that may not reflect the actual arrangement in memory, and can even permit a system to use more memory than is available by reading and writing excess memory to disk. The virtual memory manager is a component of a processor and an operating system that manages both protected and virtual memory.

It is often useful for multiple processes on the same system to communicate so that they can exchange information, synchronize, or speed up computation. This set of techniques is called *Inter-Process Communication* (IPC), and are resources provided by the operating system. Examples of IPC include files, signals, queues, pipes, shared memory, and semaphores.

A process may contain multiple *threads* of execution. The threads in a process all share the same process space, in that they have the same view of virtual memory and operating system resources. Each thread has its own copy of the processor state. Threads can be executed preemptively or concurrently in the same manner as processes. Threads are also sometimes referred to as lightweight processes.

A number of computers can be connected together to form a *cluster*, working together to solve a common problem. A cluster of computers is also commonly referred to as a *distributed* system, where each computer in the cluster has a private local memory and the computers communicate by *message passing*. The computers are often interconnected via a high-speed local network to facilitate rapid exchange of these messages. Communication via message passing is typically slower than communication via shared memory, but is capable of scaling to larger systems.

1.2 Concurrent Programming

Concurrent programming is the process of writing software as collections of computational processes that could be executed in parallel. A program developed for concurrency, along with parallel hardware, may execute much faster than a program on a single processor. For a large class of computational problems, desired execution performance can be met only with parallel hardware and concurrent programs that take advantage of the parallelism.

Concurrent programming has become increasingly relevant as systems with multiple independent processors have become commonplace. Multiprocessing systems have been available for several decades in high-end systems, but have become increasingly commoditized and now occur frequently in consumer-grade computers. Processors with multiple cores in the same physical package, or multi-core processors, have driven this trend even further.

The two major classes of explicit communication between concurrent components are shared memory communication and message passing communication. For shared memory communication, processes (or threads) communicate by altering data in shared memory locations. Typically some kind of locking will be required for coordination between them, such as semaphores or mutual exclusion (mutex), to prevent corruption of the data due to multiple actors simultaneously modifying a shared resource. Obviously shared memory communication for concurrent programming requires a shared memory architecture, such as a single SMP system.

For concurrent programming of shared memory systems, the use of threads is widespread [3]. Threads can exploit the parallelism of multiple processors, just as processes can. Shared memory communication between threads within a process is considerably easier than communication between different processes because the threads share the same memory space. Context switching between threads within a process is also typically less expensive than context switching between processes.

POSIX [6], the **P**ortable **O**perating **S**ystem **I**nterface (for Uni**x**), is a family of standards specified by the IEEE to define the application programming interface (API) for Unix systems. It also describes other standards for these systems, such as shells, utilities, and filesystem layout. POSIX threads, or Pthreads [7], is the POSIX standard API for creating and manipulating threads.

OpenMP [8] is an API for shared memory parallelism that allows the addition of simple statements to software that can automatically parallelize some sequential code. OpenMP is fairly easy to use and built into many current compilers. At a typical construct such as loop parallelization, multiple threads can execute concurrently until the parallel work is complete. Once complete, the threads join together and the program continues sequentially. This use of OpenMP has fundamentally sequential underpinnings, which will lead limited scalability. Additionally, OpenMP does not target distributed memory systems.

There exist several projects to make clusters with distributed memory systems appear as a single computer, or a single-system image (SSI). Examples include MOSIX [9][10], OpenSSI [11], DIPC [12][13], and LinuxPMI [14]. These projects maintain the appearance of a single shared memory system on a distributed system, and are sometimes called *distributed shared memory* (DSM) systems. These systems can offer ease of programming and portability, but tightly coupled memory accesses between distributed nodes make them particularly prone to thrashing [15]. Although the communication mechanism is hidden from the programmer, the latency of the communication is still present. For increased performance, it is useful to capture some knowledge about the actual memory layout.

Explicit message passing is commonly used on distributed systems in the absence of shared memory. In this case, concurrent components explicitly communicate by exchanging messages. Although BSD Sockets [16] provide a fairly universal way for remote computers to communicate, software applications designed for clusters are typically built upon a communication library. MapReduce [17] has received considerable attention for certain classes of problems, but largely for applications outside of scientific computing. The de facto standard communication library for scientific cluster computing is the Message Passing Interface (MPI) [18].

MPI is an application programming interface for message passing proposed as a standard by a broadly based committee of vendors, implementers, and users. Although there is a single specification, there are several competing implementations. Examples include MPICH [19], LAM/MPI [20][21], and

OpenMPI [22]. MPI has been called the “assembly language of parallel computing” [23] because it is an explicit, low-level interface for message passing.

The use of message passing (including MPI) is not limited to distributed memory systems. Message passing can also be efficiently implemented on a shared memory system. The Process Network model, introduced in Section 1.4, is a form a message passing.

The following section discusses common problems than can be encountered when developing software that executes concurrently.

1.3 Pitfalls of Parallel Programming

It is widely acknowledged that parallel computer programs are more difficult to develop and debug than sequential ones [24]. Concurrency introduces new classes of potential errors, such as *race conditions*. In a race condition, the result of a program unexpectedly depends on timing or the sequence of execution of concurrent components. This violates our first stated goal of determinism: the program does not produce the same results every time that it is executed.

1.3.1 Shared Memory Model

For the shared memory model, preventing this nondeterminism is simple in concept: concurrent components must prevent race conditions by preventing simultaneous modification of any shared resource, such as a shared variable in memory. A common approach is to have concurrent components use synchronization, such as acquiring a mutex or semaphore before modifying a shared resource. The portion of software that accesses a shared resource is commonly referred to as a *critical section*.

Threads are the dominant model for programming shared memory systems (and systems with the abstraction of having shared memory). Lee [3] specifically addresses determinism in the context of thread programming.

> The core abstraction of computation ..., on which all widely-used programming languages are built, emphasizes deterministic composition of deterministic components. The actions are deterministic and their sequential composition is deterministic. Sequential execution is, semantically, function composition, a neat, simple model where deterministic components compose into deterministic results.
>
> Threads, on the other hand, are wildly nondeterministic. The job of the programmer is to prune away that nondeterminism. We have, of course, developed tools to assist in the pruning. Semaphores, monitors, and more modern overlays on threads ... offer the programmer ever more effective pruning. But pruning a wild mass of brambles rarely yields a satisfactory hedge.

In practice, correctly determining all of the necessary critical sections can be challenging. Any that are missed can lead to nondeterminism in the program. Even if all of the critical sections are properly identified and locked, a second class of potential concurrency errors arises: *deadlock*.

Deadlock is a situation in which multiple processes are unable to proceed because each is waiting on one of the others in order to continue. There are very simple rules for avoiding deadlock, such as to always acquire locks in the same order [25]. However, for programs with large numbers of locks, this can be extremely difficult to enforce.

Another method for deadlock prevention is the *try-lock*. Rather than blocking while waiting to obtain a lock, the attempting process is simply notified that the lock is already being held. A similar method is the *timed-lock*, which will block up until a specified timeout while attempting to obtain a lock [7]. These approaches are intended to avoid deadlock by breaking the cycle of multiple processes waiting on each other. If a process cannot obtain a needed lock, it can simply try again later. However, this approach can lead to *live lock*, where the state of the processes involved may be constantly changing, but one or more processes are not actually progressing.

There is a tension between concurrency and deadlock. Using coarse-grained locking may simplify the task of acquiring locks in order, but reduces concurrency. Fine-grained locks permit more concurrency, but increase the difficulty of preventing deadlock. A notorious example of coarse-grained locking is Linux's Big Kernel Lock, introduced in the 2.0 series in 1996 when SMP systems were first being supported. This is a single global lock that is held whenever a thread enters kernel space. Although simple, this eliminated any concurrency in kernel space, making it a bottleneck. Although increasingly fine-grained locking has since been implemented throughout the kernel, the removal of the big lock is still an ongoing task [26].

1.3.2 Message Passing Model

Message passing is the other major class of explicit communication between concurrent components, and eliminates shared memory race conditions. However, unintended nondeterminism can still occur with races among messages, where their order of arrival at a process is not guaranteed and is affected nondeterministically by things such as scheduling variations and network latencies [27]. There are several variations in how a message passing system could behave, such as whether messages are reliable or unreliable, whether they are guaranteed to be delivered in order, and whether communication between processes is synchronous or asynchronous.

In a synchronous system, a message sender will not proceed until the receiver has received the message. This provides synchronization points between concurrent components at communication boundaries, which can simplify reasoning about the program. However, it also reduces concurrency because the communicating programs must wait on each other to proceed. By buffering messages and using asynchronous messaging, processes can overlap computation and communication. Deadlock can still occur in message passing programs, such as when processes are waiting for messages that will never arrive, or when writing to full buffers.

MPI, the de facto standard library for writing message passing programs for computer clusters, provides a flexible and wide variety of function calls for exchanging messages between concurrent components. However, detection and prevention of the mentioned pitfalls of parallel programming is left entirely in the hands of the programmer and widely acknowledged as difficult.

1.3.3 Formal Models

To address these difficulties, theoretical models of concurrent computation have been developed in attempts to formally reason about concurrent systems, and have generally been based upon a message passing system. Examples include the Actor Model [28] and different Process Calculi such as Communicating Sequential Processes (CSP) [29], both developed in the 1970's. Also developed in the 1970's is the Process Network model, which has been successfully used to model computations in streaming data applications such as signal processing systems [30][31][32]. "G," the programming language that powers LabVIEW from National Instruments, is said to be similar to Kahn Process Networks [33]. Ptolemy, a tool for modeling real-time embedded systems, also includes Process Networks as one of its models of computation [34]. The Process Network model is introduced in the next section and thoroughly discussed in Chapter 2.

1.4 Kahn Process Networks

Kahn Process Networks (KPN or simply PN) is a formal model of concurrent computation in which a set of deterministic processes communicates via a series of unbounded first-in first-out (FIFO) queues [35]. Programs can

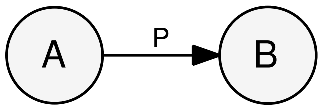


Figure 1.1: A simple Process Network program.

be represented as a directed graph, in which nodes represent processes and edges represent queues. Fig. 1.1 shows a simple Process Network program in which processes A and B execute concurrently, and A sends data to B through P, a unidirectional channel (FIFO queue).

Each node in a Process Network is a Turing equivalent computing process, and these processes are networked together while following simple dynamic firing rules at each node: blocking reads and non-blocking writes. That is, execution of a process is suspended when trying to consume data from an empty queue, but processes are never suspended for producing data (so queues can grow without bound). A process cannot detect the presence of data on an incoming queue, but attempt to read (and potentially block).

PN naturally models functional parallelism in a system, such as a pipeline. It is also natural for describing streams of data samples in a signal processing system, where nodes may transform streams from one data type to another. However, PN modeling is not limited to functional parallelism. By using scattering and gathering nodes, data parallelism can also be achieved. Fig. 1.2 shows a Process Network program that contains data parallelism, where process A scatters data to both processes $B1$ and $B2$, and

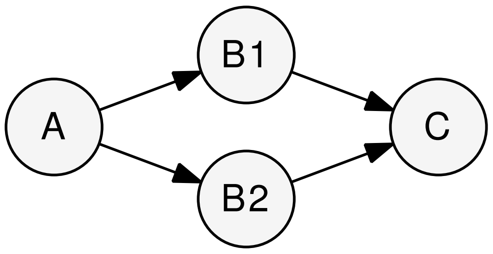


Figure 1.2: A Process Network program with data parallelism.

process C gathers the results.

Kahn's model provides the mathematically provable property of determinism of the program result regardless of the execution order of the processes in the program, which includes concurrent execution. Termination of a general PN program is undecidable in finite time, as is boundedness of the queues. The unbounded property of the queues makes an actual implementation of the PN model infeasible in bounded memory. Chapter 2 details the theory underlying Kahn's model.

1.4.1 Scheduling of Process Networks

Parks [36][37] shows that clever scheduling of the processes allows execution in bounded memory, if it is possible. By transforming the PN graph to have a feedback queue for every original queue, the transformed graph puts a limit on the queue sizes. If these queue bounds cause execution of the program to halt earlier than the original unbounded execution would have, it is an *artificial deadlock*. Parks addresses this issue by requiring online dynamic deadlock detection and resolution. He argues that lengthening the shortest deadlocked full queue will resolve the artificial deadlock and allow execution to continue. If the program can be executed in bounded memory, then the bound will eventually be found. Parks's original paper is a seminal work on the bounded scheduling of PN, but contains some important mistakes. Still, he sparked further work into the bounded scheduling of PN.

Parks states that *global deadlock* of the network, in which no process is able to execute, is required before detecting and resolving an artificial deadlock. However, not all artificial deadlocks result in a global deadlock. Geilen and Basten show [38] an example where Parks' algorithm produces an incomplete execution, which arguably compromises the determinacy of Kahn's model. They also point out that Parks' algorithm cannot schedule a PN graph composed of disjoint chains. They present an improved scheduling algorithm using local deadlock detection. They also argue that bounded scheduling will not yield a complete execution unless the PN is *effective*. They define an effective PN to mean that all produced tokens will eventually be consumed. In Section 2.3, I show a counterexample of a non-effective PN graph that can be scheduled correctly. This demonstrates that there is a larger set of PN programs that can be completely executed in bounded memory than Geilen and Basten propose.

Although Parks, Geilen, and Basten all describe the use of a dynamic deadlock detection and resolution algorithm, none of them provide a detailed algorithm for use. In Section 3.3 I develop and prove a distributed dynamic deadlock detection and resolution algorithm (D4R) for use with bounded execution of process networks [39]. This algorithm is based on a previous distributed algorithm by Mitchell & Merritt [40]. The D4R algorithm detects local deadlocks and determines whether a deadlock is real or artificial. For artificial deadlocks, the algorithm locates the blocked process that is key to resolving the deadlock: the process that is blocked writing to a queue that must grow in order for the network to proceed. The D4R algorithm is also distributed and scalable, which makes it appropriate for a distributed PN implementation.

1.4.2 Synchronous Dataflow

Synchronous Dataflow (SDF) [41][42] is another model of computation similar to Process Networks that represents a system as a directed graph of FIFO queues connecting processing elements. In dataflow, the nodes are referred to as *actors*, and the data samples flowing over the queues are referred to as *tokens*. Each actor *fires* or executes according to firing rules that specify what tokens must be available at its inputs. When firing, an actor consumes some number of input tokens and produces some number of output tokens.

In Synchronous Dataflow, whenever an actor fires it always consumes and produces the same number of tokens. Each different actor may consume or produce a different number of tokens, but the number for each individual actor is invariant. Because of the restrictive model and the predictable behavior of each actor, it is possible to compute a static, sequential execution order of actors, and also to know the flow of control and memory requirement for each queue. An SDF schedule compiler can construct a static schedule that is repeatedly executed. The asymptotic complexity to determine a sequential schedule for SDF is $O(N^3)$ for a system with N nodes [43].

SDF is well suited for modeling many digital signal processing (DSP) systems and subsystems, in which the fixed schedule is repeatedly executed and the overhead of a dynamic scheduler is eliminated. The SDF model has been used in some commercial electronic design automation (EDA) tools [44], such as the HP/Agilent EEsof Advanced Design System [45] and the Coware Signal Processing Worksystem.

1.4.3 Computation Graphs

Computation Graphs (CG) by Karp & Miller [46] are another model of computation similar to Process Networks. As in SDF, the firing behavior of each actor is fixed: each time a node executes, it will consume and produce the same number of tokens on each of its inputs and outputs. However, Computation Graphs bring an important concept that is leveraged in this dissertation: firing thresholds. Before an actor can fire, there is a threshold minimum number of tokens that must be present on the input. Clearly the threshold number of tokens required for firing has to be equal to or greater than the number of tokens that will be consumed. Computation Graphs have necessary and sufficient conditions for both termination and boundedness, and a static execution schedule can be determined at compile time. Synchronous

Dataflow is a special case of Computation Graphs where the number of tokens consumed upon firing of an actor is always equal to the firing threshold (for every actor input in the system).

The concept of a firing threshold has applications in high-throughput DSP systems, where it is common to have algorithms that operate on continuous, overlapping streams of data, or *sliding window* algorithms. Examples include filters, and sliding window fast Fourier transform (FFT) algorithms. For example, a finite impulse response (FIR) filter of order N requires $N + 1$ input samples to compute a single output sample. Computing the next output sample would require reuse of the N previous samples and one new sample. If such a filter were modeled as a Kahn Process Network or as Synchronous Dataflow, the node implementing the filter would be required to store these N previous samples internally as node (process) state.

With firing thresholds in Computation Graphs, a node can require more tokens to be present on a queue than it will actually consume upon execution. For example, a filter node could require that $N + 1$ samples be present on a queue before executing, and then consume only a single input sample. The N newest samples are still present in the queue, so it is not necessary to store their state in the filter node.

This separation of consumption and threshold counts permits efficient modeling of algorithms on overlapping continuous streams of data, and permits implementation with memoryless nodes. This ability to model these DSP algorithms as memoryless is an additional desirable property of a model of computation. This simplifies implementation of composable components that compute sliding window algorithms, because all of the overlap state is retained in the queues. Such a model can also enable more efficient execution of sliding window algorithms, which are very common in embedded real-time signal processing systems.

Table 1.1 summarizes some desirable properties of the models of computation that have been discussed in this section. The following section introduces, Computational Process Networks, a new model of computation.

Table 1.1: Properties of Models of Computation.

	Model		
Property	SDF	CG	KPN
Determinism	✓	✓	✓
Boundedness	✓	✓	
Scalability			✓
Composability			✓
Firing Thresholds		✓	

1.5 Computational Process Networks

In this dissertation, I propose the Computational Process Network (CPN) model which is based on the formalisms of Kahn's PN model, but with enhancements that are designed to make it efficiently implementable. These enhancements include multi-token transactions to reduce execution overhead, multi-channel queues for multi-dimensional synchronous data, and both consumer and producer firing thresholds for queues. Section 4.2 describes these enhancements in more detail.

Multi-token transactions are a simple extension to the Kahn Process Network model, but can provide a significant performance enhancement as the overhead for a queue transaction is amortized. Multi-channel queues permit straightforward modeling of systems with multiple channels, such as audio streams in a surround sound system, or beams in a sonar beamforming system.

Firing thresholds for consumers, a concept borrowed from Computation Graphs, enable memoryless computation of sliding window algorithms. The benefits of this model to signal processing systems are discussed in the previous subsection, 1.4.3.

CPN also provides the dual to consumer firing thresholds, which is firing thresholds for producers. A node can access more free space than it will fill, thereby allowing variable-rate outputs without requiring data copies. These thresholds allow the decoupling of computation from communication when developing process nodes, and permit a zero-copy queue implementation. By eliminating unnecessary data copying in high-throughput systems, the computer is free for additional computation tasks. CPN is useful for modeling signal processing systems of streaming data, and naturally captures concurrency in these systems.

1.6 Thesis Statement and Expected Contributions

In this dissertation, I defend the following thesis statement:

The Computational Process Network model preserves the formal properties of Kahn Process Networks while reducing the operations required to implement sliding window algorithms, which are common in digital signal processing. The CPN model is also efficiently implementable.

First I present an algorithm for distributed, dynamic deadlock detection and resolution, dubbed the D4R algorithm. Dynamic deadlock resolution is required for complete execution of Process Networks in bounded memory, and the D4R algorithm is suitable for use in distributed implementations of both KPN and CPN.

Second, I show by graph transformation that the CPN model preserves the formal properties of KPN: determinism, scalability, and composability. I show the reduction in operations of CPN versus KPN for sliding window algorithms, including an example of measured execution. The CPN semantics also provide multi-token firings, multi-channel queues, and a zero-copy interface, for a general reduction in operations compared to KPN for multi-dimensional signal processing.

Third, I present a high-throughput CPN framework implementation in C++, which maps naturally onto distributed targets. This framework uses POSIX threads, and can exploit parallelism in both multi-core and distributed systems. Using a simple coordination language to describe a CPN program graph, developers can build systems from deterministic, composable components.

I also present several case studies to illustrate the capabilities and performance of the CPN model and framework. I present the Sieve of Eratosthenes, a simple algorithm for finding prime numbers. This algorithm has been used in previous PN literature, and I compare execution performance using the enhancements present in the CPN model. I present a case study of randomly generated graphs with dynamically changing connections to demonstrate the robustness and stability of the CPN framework. I also present a 3D circular convolution sonar beamformer and replica correlator. This final case study demonstrates the high throughput and scalability of a real-time multidimensional signal processing system using the CPN model and framework.

1.7 Conclusion

Table 1.2 provides a list of definitions for acronyms that are used in this dissertation.

Many signal and image processing systems for high-throughput, high-performance applications require concurrent implementations in order to realize desired performance. Examples may include sonar beamforming, synthetic aperture radar processing, or high definition video processing. Multi-core processors [47] and distributed systems are attractive targets for implementing these types of applications.

Developing software for concurrent systems is widely acknowledged to be difficult, with common industry practice leaving the burden of preventing concurrency problems on the programmer. Formal models have been devel-

Table 1.2: Acronyms used in this dissertation

API	application programming interface
BPN	Bounded Process Network
BSD	Berkeley Software Distribution
CG	Computation Graphs
CPN	Computational Process Network
CSP	Communicating Sequential Processes
D4R	Distributed Dynamic Deadlock Detection and Resolution
DSM	distributed shared memory
DSP	digital signal processing
EDA	electronic design automation
FFT	fast Fourier transform
FFTW	the Fastest Fourier Transform in the West
FIFO	first-in first-out
FIR	finite impulse response (filter)
FLOPS	floating point operations per second
IEEE	Institute of Electrical and Electronic Engineers
IPC	inter-process communication
JSON	JavaScript Object Notation
KPN	Kahn Process Network
LFM	linear frequency modulated (pulse)
LFSR	linear feedback shift register
MPI	Message Passing Interface
PN	(Kahn) Process Network
POSIX	Portable Operating System Interface for Unix
RDMA	remote direct memory access
SDF	Synchronous Dataflow
SIMD	single instruction multiple data
SMP	symmetric multiprocessing
SSE	Streaming SIMD Extensions
SSI	single-system image
STL	Standard Template Library
TCP/IP	transmission control protocol / internet protocol
VMM	virtual memory manager
XML	Extensible Markup Language

oped to aid reasoning about concurrent systems. A model that has been successfully used for signal and image processing systems is Process Networks.

Chapter 2 discusses Kahn's formal Process Network model. It details the theory underlying Kahn's model, and discusses the approaches that have been taken toward scheduling PN in bounded memory. This chapter also addresses Computation Graphs, which introduce the concept of firing thresholds.

Chapter 3 discusses deadlock detection, which is required for bounded scheduling of PN. It discusses previous work in deadlock detection, and then presents the D4R algorithm: Distributed Dynamic Deadlock Detection and Resolution. This algorithm detects deadlocks in a PN graph and determines whether a deadlock is real or artificial. Proofs and case studies are presented in support of the D4R algorithm. The D4R algorithm is suitable for use in distributed implementations of both KPN and CPN

Chapter 4 proposes the Computational Process Network model which is based on the formalisms of Kahn's PN model, but with enhancements that are designed to make it efficiently implementable. I show by graph transformation that the formal properties of KPN are preserved, and also that CPN reduces the number of operations required to implement sliding window algorithms as compared to KPN.

Chapter 5 describes details of the reference CPN framework implementation. CPN Nodes map to POSIX threads to exploit concurrent hardware, and queues use the virtual memory manager to provide apparent data circularity. The CPN framework can allow rapid development of determinate, scalable concurrent systems built from composable components.

Chapter 6 describes case studies used to illustrate the utility of the CPN model and implementation. First is the Sieve of Eratosthenes, a simple algorithm for finding prime numbers. Second is a case study of random graphs with dynamic connections to demonstrate the robustness and stability of the CPN framework. Third is a 3D circular convolution beamformer and replica correlator that is representative of a real-time signal processing algorithm that can be implemented within the provided model and framework.

Chapter 7 concludes this dissertation.

Chapter 2

Kahn Process Networks

Developing concurrent computer programs is widely acknowledged to be difficult, so efforts have been undertaken to reason formally about concurrent systems. Kahn Process Networks is a formal model of concurrent computation first introduced by French computer scientist Gilles Kahn in 1974 [35]. Kahn introduces a language with simple semantics and formally studies the results, with the goal of applying mathematical approaches to programming languages and system design.

Kahn's model can naturally describe signal processing systems in which infinite streams of data samples are incrementally transformed by a collection of processes executing in sequence or in parallel [36]. As such, Process Networks have been used in a number of concurrent applications, including digital signal processing systems [30].

Section 2.1 introduces Kahn's simple semantics for communication. Section 2.2 details Kahn's illustrative sample program and discusses the formal consequences. Section 2.3 discusses progress toward scheduling of process networks in bounded memory. Section 2.4 introduces Computation Graphs, a similar model from which this work leverages the concept of firing thresholds.

Section 2.5 concludes this chapter.

2.1 Introduction

In Kahn's model, a set of processes communicates with each other via a network of unbounded FIFO queues [35]. He describes it as a set of Turing machines connected via one-way tapes, in which each machine can use its own separate working tape. These processes may have any number of input or output queues, but they can communicate only via the queues. Queues carry a particular type of data, such as the primitive type *int*.

The semantics for communication are the blocking read data function *get* and the non-blocking write data function *put*. When a processes wishes to receive a data value (token) from an input queue, it calls *get* on the queue. If no data is available in the queue, then the process will be suspended until a data value arrives. When data is present in the queue, *get* returns the first data element in the queue. A process cannot determine whether or not data is present in an input queue, and it can only call *get* on one queue at a time. When a process wishes to send a data element to an output queue, it calls *put* with two arguments: the data element to send, and the queue. Because queues can be unbounded in length, calling *put* can never cause a process to be suspended. (Kahn's original semantics were *wait* and *send*, which he later updated to *get* and *put* [48]).

At any point in time, a process is either enabled or it is suspended. A suspended process is blocked waiting on (only) one of its inputs. Kahn shows that programs following these semantics are *determinate*: the history of tokens in the queues is the same regardless of the execution order of the processes.

2.2 Kahn Process Networks

Kahn [35] presents an elementary concurrent sample program built upon his simple semantics of *get* and *put*, using a language similar to Algol. He then examines the behavior of this program from a formal, mathematical viewpoint, and draws conclusions about provable properties for general programs obeying these semantics. Here I recreate Kahn's example, describing it with the more current object-oriented C++ programming language [49].

2.2.1 Kahn's Simple Example

Figure 2.1 shows a diagram of Kahn's example. Four processes are connected by one-way FIFO queues. There are single instances of processes f and g, and two instances of process h each taking different parameters. The following three figures define the functions which implement these processes, and which communicate only via *get* and *put* over the associated queues.

Figure 2.2 defines the function f, which takes two queues of integers as inputs, and a single queue of integers as an output. It eternally interleaves the two input queues (U and V) into the single output queue (W), and never returns.

The code `iqueue<int>` represents a C++ template input queue class that carries tokens of type *int*. The class `iqueue` provides only the method *get*,

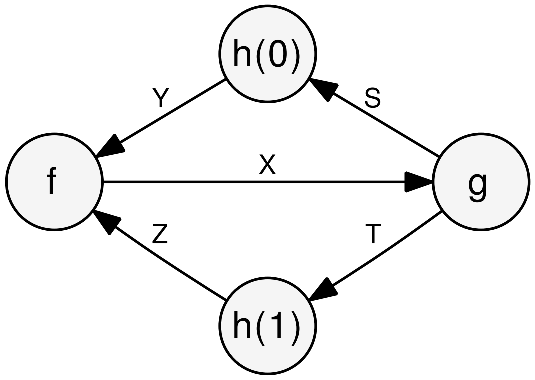


Figure 2.1: Kahn's example of a simple Process Network program.

```
void f(iqueue<int> U, iqueue<int> V, oqueue<int> W)
{
  while (true) {
    W.put( U.get() );
    W.put( V.get() );
  }
}
```

Figure 2.2: A function, f, that interleaves two input queues into one output.

```
void g(iqueue<int> U, oqueue<int> V, oqueue<int> W)
{
  while (true) {
    V.put( U.get() );
    W.put( U.get() );
  }
}
```

Figure 2.3: A function, g, that distributes one input queue into two outputs.

which takes no argument and returns a single token of type *int* (the template argument type). According to Kahn's semantics, *get* will block until a token is available, remove it from the queue, and return it. The code `oqueue<int>` represents a C++ template output queue class also carrying tokens of type *int*. The class `oqueue` provides only the method *put*, which takes an argument of *int* (the template argument type) and has no return value. According to Kahn's semantics, *put* will immediately insert the token into the queue and return nothing to the caller. Recall that in Kahn's model, queues may be infinitely long and *put* will never block.

Figure 2.3 defines the function g, which takes a single queue of integers as input, and two queues of integers as outputs. It eternally distributes tokens from the input queue (U) into the two output queues (V and W), and never returns.

Figure 2.4 defines the function h, which takes a single integer (x), and input and output queues of integers (U and V). Function h inserts a token at the head of a queue by first sending the single integer parameter to the output

```
void h(int x, iqueue<int> U, oqueue<int> V)
{
  V.put( x );
  while (true) {
    V.put( U.get() );
  }
}
```

Figure 2.4: A function, h, that inserts an element at the head of a queue.

```
main()
{
  queue<int> X, Y, Z, S, T;
  process<f>( Y, Z, X );
  process<g>( X, S, T );
  process<h>( 0, S, Y );
  process<h>( 1, T, Z );
}
```

Figure 2.5: A main program implementing Kahn's example in Figure 2.1.

queue, and then eternally copying integer tokens from the input queue to the output queue.

Figure 2.5 contains a main function that implements Kahn's full example from Figure 2.1, by instantiating the five queues and spawning the four processes. The template class `queue` multiply inherits from both classes `iqueue` and `oqueue`, and is specified to carry integer tokens. For now, we will conveniently ignore the fact that Kahn's model requires queues of possibly unbounded size. All of the queues are empty upon instantiation, and have the same names as shown in Figure 2.1.

A template class `process` could be implemented by using template meta-programming techniques and taking a function as a template parameter. (It may require somewhat more syntax than shown, but adding this would not further the discussion.) The class constructor could create a new thread, which calls the specified (template parameter) function using the provided arguments. With such a `process` class, the main function in Figure 2.5 will spawn four thread processes, one executing function f, one executing function g, and two executing function h but with different integer parameters.

Initially, processes f and g are blocked waiting on their inputs, but processes h(0) and h(1) can write tokens 0 and 1 to their respective outputs. The ambitious reader can follow these tokens around the loops. The 0 token will pass around the upper loop from queues Y to X to S, and then again back to queue Y and eternally repeating the cycle. Similarly, the 1 token will cycle around the bottom loop. With some examination, one can see that the sequence of tokens on queue X will be alternating 0's and 1's that repeat forever. For this simple example, Kahn [35] sets out to formally prove three assertions:

1. All processes in the program will execute forever,
2. Queue X will pass an alternating sequence of 0's and 1's forever, and
3. If any process were to (erroneously) stop, the whole system would stop.

We proceed with Kahn's formal, mathematical representation of process networks. For clarity, I borrow heavily from the notation later used by Parks [36].

2.2.2 Mathematical Representation

In Kahn's representation, the queues (or channels) are *streams*, and the processes are mathematical functions that map streams into other streams. With these definitions, a process network can be described by a set of mathematical equations, and one can solve for the least fixed point of these equations. Kahn showed that the least fixed point is unique. First we proceed with some definitions.

2.2.2.1 Streams

A stream is defined as a sequence of data elements (or tokens), e.g. $X = [x_0, x_1, x_2, \ldots]$. In the context of a process network, the contents of a stream X represent the tokens that have been inserted into a queue. Sequences can be ordered in a *prefix order*, such that if X is a prefix of Y (or they are equal) it is written as $X \sqsubseteq Y$. For example, $[0] \sqsubseteq [0, 1] \sqsubseteq [0, 1, 2]$. The empty sequence $\perp$ is a prefix of all sequences, i.e. $\forall X, \perp \sqsubseteq X$.

A sequence of prefix-ordered streams, $\vec{X} = (X_0, X_1, \ldots)$ where $X_0 \sqsubseteq X_1 \sqsubseteq \cdots$, is an *increasing chain* of streams. An increasing chain $\vec{X}$ represents the entire history of tokens that have been inserted into a PN queue as execution progresses; each stream element of $\vec{X}$ contains the history of the queue at some point in the execution. The supremum (or least upper bound) of an increasing chain of streams $\vec{X}$ is the shortest stream such that all streams in $\vec{X}$ are a prefix of the supremum $\sup(\vec{X})$, i.e. $\forall X_i \in \vec{X}, X_i \sqsubseteq \sup(\vec{X})$. For any increasing chain of streams $\vec{X}$, the supremum exists and is unique: $\sup(\vec{X}) \equiv \lim_{n\to\infty} X_n$. The order is *complete* because every increasing chain of streams has a supremum that is itself a stream.

The prefix order binary relation $\sqsubseteq$ is a partial order on the set of all streams because it is reflexive, antisymmetric, and transitive. The set of all streams using the prefix order $\sqsubseteq$ is therefore a *complete partial order.* Tuples of streams are also a complete partial order. For example, $(X_0, X_1) \sqsubseteq (Y_0, Y_1) \leftrightarrow X_0 \sqsubseteq Y_0, X_1 \sqsubseteq Y_1$. The fact that streams form a complete partial order will aid in solving for the least fixed point of a set of equations that describe a process network in Section 2.2.2.3.

2.2.2.2 Processes

A process is a functional mapping between input streams and output streams. For each process, a mathematical equation can be written to describe the mapping. For example, the C++ function in Figure 2.2 can be described by the equation $W = f(U, V)$. Similarly, the function in Figure 2.3 can be described by the equation $(V, W) = g(U)$.

A functional mapping $f : X \to X$ between partially ordered sets is defined to be *continuous* (for the increasing chain $X_0 \sqsubseteq X_1 \sqsubseteq \cdots$) if and only if

$$f(\lim_{n\to\infty} X_n) = \lim_{n\to\infty} f(X_n) \tag{2.1}$$

All such continuous functions are also *monotonic*, i.e. for the function f:

$$X_n \sqsubseteq X_m \Rightarrow f(X_n) \sqsubseteq f(X_m) \tag{2.2}$$

Table 2.1 describes three examples of continuous functions that are commonly used in functional programming: first, rest, and cons. Continuous, monotonic functions of a complete partial order have a least fixed point that can be computed.

Table 2.1: Examples of continuous functions.

function	behavior	behavior for $\perp$
first(U)	returns first token of stream U	first$(\perp) = \perp$
rest(U)	returns stream U with first token removed	rest$(\perp) = \perp$
cons(x, U)	inserts token x at beginning of stream U	cons$(\perp, U) = \perp$ cons$(x, \perp) = [x]$

2.2.2.3 Least Fixed Point

The Kleene fixed-point theorem [50] states that every continuous functional mapping $f : X \to X$ between partially ordered sets has a unique least fixed point. The solution is computed by iterating the function f on the least element $\perp$ of X, which gives the increasing chain

$$\vec{X} = [\perp, f(\perp), f(f(\perp)), \cdots, f^n(\perp), \cdots] \tag{2.3}$$

The unique least fixed point of the function f is simply the supremum of the increasing chain sup$(\vec{X})$ with $\vec{X}$ as defined in (2.3). This increasing chain $\vec{X}$

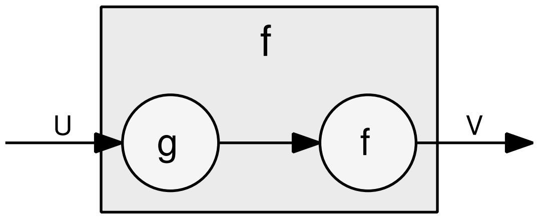


Figure 2.6: Recursive definition of a process f.

may grow to be of infinite length (just as queues in a process network may be unbounded). By defining a function f recursively (where the process is defined in terms of itself) and using mathematical induction, the least fixed point may still be computed. Figure 2.6 shows a graphical representation of a sample process f that is recursively defined. Written as equations, it is recursively defined as $f(U) = f(g(U))$.

By applying this approach to a set of equations (describing processes in a process network) and solving for the least fixed point for the system of equations, Kahn [35] proves that the history of the streams (and queues) in the network can be analytically computed. Further, this solution is a property of the system of equations and does not vary with execution order or concurrent operation of processes in the system. We now return to Kahn's example.

2.2.3 Proving Kahn's Assertions for a Simple Example

The processes in Kahn's simple example (Figures 2.2, 2.3, and 2.4) can be defined recursively in terms of the continuous, monotonic functions in

Table 2.1. Process g, with two outputs, can be separated into two continuous functions, g_1 and g_2.

$$f(U,V) = \mathrm{cons}(\mathrm{first}(U), \mathrm{cons}(\mathrm{first}(V), f(\mathrm{rest}(U), \mathrm{rest}(V)))) \tag{2.4}$$

$$g(U) = (g_1(U), g_2(U)) \tag{2.5}$$

$$g_1(U) = \mathrm{cons}(\mathrm{first}(U), g_1(\mathrm{rest}(\mathrm{rest}(U)))) \tag{2.6}$$

$$g_2(U) = \mathrm{cons}(\mathrm{first}(\mathrm{rest}(U)), g_2(\mathrm{rest}(\mathrm{rest}(U)))) \tag{2.7}$$

$$h(x,U) = \mathrm{cons}(x,U) \tag{2.8}$$

The process f (described by function f) is represented graphically in Figure 2.7. All of the functions describing the simple example from (2.4) to (2.8) are therefore also continuous and monotonic. Similarly, any process that uses the simple blocking *get* and non-blocking *put* semantics can be defined recursively and has a corresponding set of continuous, monotonic functions [35].

The set of functions that describe such a system is a set of fixed point equations over complete partial orders, in which the operators are continuous. This set of equations has a unique least fixed point, and that solution corresponds to the history of tokens produced on the queues [35].

The system that is Kahn's simple example (Figures 2.1 – 2.5) can be

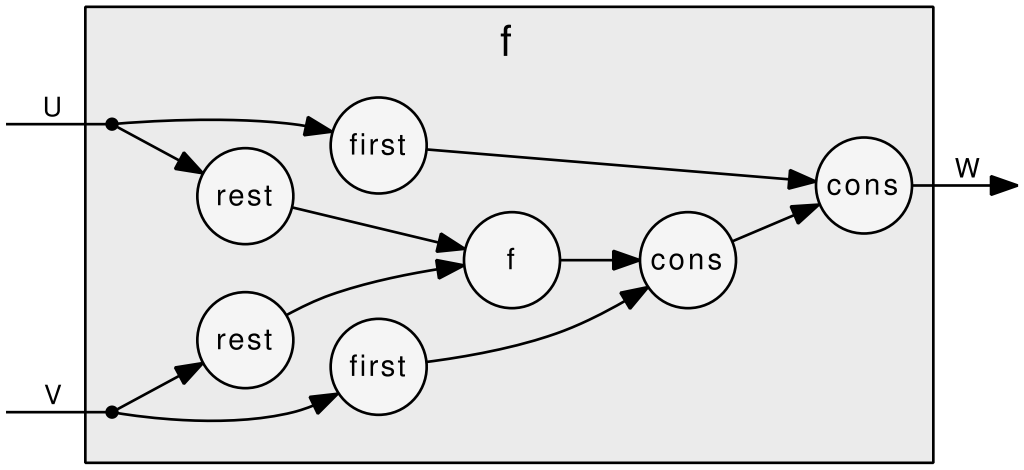


Figure 2.7: Recursive definition of process f from Figure 2.2.

described by the following system of equations:

$$X = f(Y, Z) \tag{2.9}$$

$$Y = h(0, S) \tag{2.10}$$

$$Z = h(1, T) \tag{2.11}$$

$$(S, T) = g(X) \tag{2.12}$$

By solving for X and splitting g as before in (2.5), this system of equations can be reduced to a single equation:

$$X = f(h(0, g_1(X)), h(1, g_2(X))) \tag{2.13}$$

Kahn [35] proves by structural induction that the history of stream X is an infinite alternating sequence of zeros and ones.

Parks [35] alternatively solves for (S, T) and illustrates the iterative procedure to solve for the least fixed point. Here, the single system equation reduces to:

$$(S, T) = g(f(h(0, S), h(1, T))) \tag{2.14}$$

Iterating (2.14) by beginning with the empty sequence for each stream gives the following history:

$$(S_0, T_0) \qquad = (\perp, \perp) \tag{2.15}$$

$$(S_1, T_1) = g(f(h(0, \perp), h(1, \perp))) \qquad = ([0], [1]) \tag{2.16}$$

$$(S_2, T_2) = g(f(h(0, [0]), h(1, [1]))) \qquad = ([0, 0], [1, 1]) \tag{2.17}$$

$$(S_3, T_3) = g(f(h(0, [0, 0]), h(1, [1, 1]))) \qquad = ([0, 0, 0], [1, 1, 1]) \tag{2.18}$$

$$(S_n, T_n) = g(f(h(0, S_{n-1}), h(1, T_{n-1}))) \quad = ([0, 0, 0, \cdots], [1, 1, 1, \cdots]) \tag{2.19}$$

Proving by induction, $\sup(\vec{S}) = [0, 0, 0, \cdots]$ and $\sup(\vec{T}) = [1, 1, 1, \cdots]$. Using these results with (2.9) to (2.11) we conclude that

$$\sup(\vec{Y}) = h(0, \sup(\vec{S})) \qquad = [0, 0, 0, \cdots] \tag{2.20}$$

$$\sup(\vec{Z}) = h(1, \sup(\vec{T})) \qquad = [1, 1, 1, \cdots] \tag{2.21}$$

$$\sup(\vec{X}) = f(\sup(\vec{Y}), \sup(\vec{Z})) \qquad = [1, 0, 1, 0, \cdots] \tag{2.22}$$

Equation (2.22) matches the result from Kahn [35], thereby analytically solving for the history of queue X and answering the first two questions raised about this simple example in Section 2.2.1. Kahn states the following:

> The simplicity of the program and the proof produced should not induce the reader into believing that only very simple minded proofs are feasible.

All programs comprised of processes that use the simple blocking *get* and non-blocking *put* semantics can be described as a system of continuous equations on streams. The unique fixed point solution is a function of the values of the streams [35]. The fact that the fixed point solution of the network equations corresponds to the behavior of the executing processes is sometimes called the *Kahn Principle* [38][51].

More concretely, the Process Network model of computation is both composable and determinate. Processes written to obey this model can be combined with hierarchy and clustering to create larger, more complex systems. Individual process behavior can be analyzed and tested in isolation, because processes are modular and self-contained. As long as the externally visible behavior of a process is unchanged, differing internal details will not introduce perturbations in the remainder of the system. The value of the history of tokens on all of the queues depends only on the solution to the fixed point equation, not on execution order of the processes in the system.

The only exception to this solution is an *incomplete execution*, in which at least one process does not execute to completion. In an incomplete execution, the program will produce less output than the solution to the fixed point equation. What is produced is correct. Kahn [35] mentions an unfair scheduler (which starves a ready process) as something that could cause an incomplete execution. An unfair scheduler addresses Kahn's final question raised about the simple example in Section 2.2.1. It is well known that the Kahn Principle hinges on fair scheduling of the processes, where fairness means that all processes that can make progress should make progress at some point [38][52].

Fixed point equations can also be used to describe recursive Process Networks, where continuous functions on streams are defined recursively. Such continuous mappings describe a complete partial order where $f \sqsubseteq g$ if and only if $\forall X, f(X) \sqsubseteq g(X)$. This leads to the potential for unbounded parallelism, in which an unbounded number of processes may compute in parallel. The Sieve of Eratosthenes case study in Section 6.2 is an example of such unbounded parallelism.

2.2.4 Termination

Just like determinism, whether a Process Network program will terminate is given only by its definition and is not affected by the execution order of the processes in the system [53]. The least fixed point of the system of equations describing the program determines the value (and therefore the length) of every stream in the program. If the length of every stream in the solution is finite, then the program will terminate. If one or more stream is of infinite length, the program will never terminate.

Kahn's simple example in Figure 2.1 produces an infinite number of tokens and is non-terminating. By removing a single process we change it to a terminating program, as in Figure 2.8. The modified program can be described

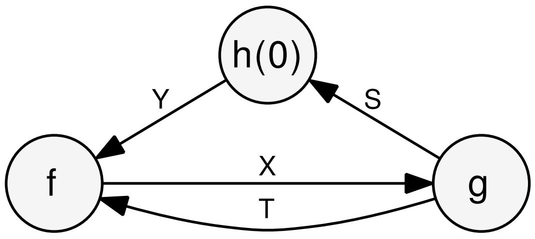


Figure 2.8: A terminating Process Network program.

by the single system equation:

$$(S, T) = g(f(h(0, S), T)) \tag{2.23}$$

Again finding the least fixed point by iterating (2.23) gives the following result:

$$(S_0, T_0) \qquad = (\perp, \perp) \tag{2.24}$$

$$(S_1, T_1) = g(f(h(0, \perp), \perp)) \qquad = ([0], \perp) \tag{2.25}$$

$$(S_2, T_2) = g(f(h(0, [0]), \perp)) \qquad = ([0], \perp) \tag{2.26}$$

Streams S and T (and all others in the network) have finite length, so the program terminates.

The least fixed point solution determines the content and length of every stream, including the ordering of tokens in each stream, but says nothing about the execution order in which tokens are produced. There are generally many possible execution orders that can lead to the least fixed point. In a *complete execution* of a Process Network, the result corresponds to the least fixed point solution and none of the streams can be extended. In a *partial* *execution*, one or more of the streams can still be extended and the least fixed point solution has not been reached.

Parks [36] defines a terminating Process Network program to be one where all complete executions have a finite number of operations, and a non-terminating program to be one where all complete executions have an infinite number of operations. He also points out that determining whether a general Process Network program will terminate is not possible in finite time. This is related to the halting problem for Turing machines, which is undecidable in finite time.

Parks's [36] main contribution is to take Kahn's unbounded queue model and attempt to implement Kahn Process Networks in bounded memory when possible, by using clever scheduling. Park's bounded scheduling approach is discussed in the next section.

2.3 Bounded Scheduling of Process Networks

The previous section shows that a Process Network is determinate. That is, the history of tokens on all of the queues depends only on the solution to the fixed point equation and not on the execution order of the processes in the system. However, the execution order (or schedule) can affect the boundedness of a Process Network program.

Boundedness is related to the number of unconsumed tokens in the queues of a Process Network during its execution. For some Process Network programs, the queues will grow to be of infinite length, and are therefore impossible to implement in finite memory. For *bounded* Process Network programs, the number of unconsumed tokens in queues may be finite and the program can be implemented in finite memory. A Process Network program with infinite length streams may still be bounded. For example, Kahn's simple example in Fig. 2.1 has infinite (and non-terminating) result streams in the least fixed point solution. However, it is bounded because there are only ever a finite number of unconsumed tokens in the queues.

This section discusses scheduling policies that have been developed toward implementing Process Networks in bounded memory. First we define execution order and boundedness more rigorously.

2.3.1 Execution Order

We define the execution order of a Process Network to be the order of the *get* and *put* operations in the system, as in Parks [36]. Given the stream $X = [x_0, x_1, x_2, \ldots]$, $t_{put}(x_0)$ is the time that token x_0 was written (with *put*) into the associated queue and $t_{get}(x_0)$ is the time that token x_0 was read (with *get*) from the associated queue. A token must be written before it can be read, so $t_{put}(x_i) \leq t_{get}(x_i)$ for every token and every queue in a Process Network. The fact that the queues have FIFO behavior also gives, for all i and j that are indices of tokens in a queue, that

$$t_{get}(x_i) \leq t_{get}(x_j) \longleftrightarrow t_{put}(x_i) \leq t_{put}(x_j) \tag{2.27}$$

Some operations can occur simultaneously; e.g. if a process produces two tokens x_i and x_{i+1} simultaneously, then $t_{put}(x_i) = t_{put}(x_{i+1})$. The processes can impose additional restrictions on ordering. The function f from Figure 2.2 interleaves two input streams into a single output stream, and is defined mathematically as $W = f(U, V)$. This process reads alternately from the two inputs and imposes an order on its input tokens,

$$t_{get}(u_i) \leq t_{get}(v_i) \leq t_{get}(u_{i+1}) \leq t_{get}(v_{i+1}) \leq \cdots \tag{2.28}$$

Because the process writes to its output W after reading from each input stream U and V, it also imposes the order

$$t_{get}(u_i) \leq t_{put}(w_{2i}) \leq t_{get}(v_i) \leq t_{out}(w_{2i+1}) \leq \cdots \tag{2.29}$$

A *sequential* execution of a Process Network is a *total ordering* of all *get* and *put* operations. For any pair of operations (on any of the queues) the execution schedule dictates an ordering, e.g. either $t_{get}(x_i) \leq t_{put}(y_j)$ or $t_{put}(y_j) \leq t_{get}(x_i)$. However, a *parallel* execution is not so ordered; it is only a *partial ordering* of the operations. For some operations the ordering is not given by a schedule and is unknown, e.g. $t_{put}(y_i)$ and $t_{put}(z_i)$ in Figure 2.1.

Any execution order must satisfy all of the ordering restrictions: write-before-read for each token, FIFO queues, and any orders imposed by the definitions of the processes in the network. Given that processes can be arbitrarily complex, the ordering restrictions imposed by the processes may not be fully characterized. In general, there will be many possible execution orders that satisfy all of the restrictions. I can now define boundedness of a Process Network.

2.3.2 Boundedness

Fig. 2.9 is a simple example that can illustrate the definitions of different types of boundedness for Process Networks, as previously defined by Parks [36]. If process A eternally produces single tokens and process B eternally consumes single tokens, then the number of unconsumed tokens that accumulate on queue P depends on the execution order of A and B.

A queue is defined to be *bounded by b* if there exists at least one complete execution of the Process Network in which the number of unconsumed tokens in the queue does not exceed b. A queue is said to be *bounded* if there exists a finite constant b such that the queue is bounded by b. A Process Network is defined to be *bounded by b* if every channel in the network is bounded by b. A Process Network is bounded if there exists a finite constant b such that it is bounded by b. If A and B in Fig. 2.9 are always scheduled to execute alternately, then P would only ever contain a single unconsumed token and is bounded by 1. If A executes b times before B executes, then P will have at most b unconsumed tokens. The queue P and the Process Network in Fig. 2.9 are bounded (when B consumes data as defined) because there exist complete executions with a finite number of unconsumed tokens. However, this example is not *strictly bounded* because some of the (infinite number of) execution orders are unbounded.

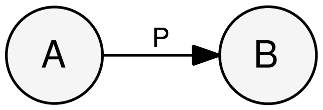


Figure 2.9: A simple Process Network program.

A queue is defined to be *strictly bounded by b* if the number of unconsumed tokens in the queue does not exceed b for *all* complete executions of the Process Network. A queue is said to be *strictly bounded* if there exists a finite constant b such that the queue is strictly bounded by b. A Process Network is defined to be *strictly bounded by b* if every channel in the network is strictly bounded by b. A Process Network is strictly bounded if there exists a finite constant b such that it is strictly bounded by b. Regardless of the choice of b, the Process Network in Fig. 2.9 always has an execution order in which more than b unconsumed tokens are present on P, so it is not strictly bounded. If we were to alter the behavior of process B in Fig. 2.9 such that it never consumes tokens, then P would increasingly fill with tokens and both P and this Process Network would be unbounded.

Kahn's simple example from Fig. 2.1 is strictly bounded because its queues are strictly bounded. For all execution orders, queues S, T, Y, and Z can never contain more than a single unconsumed token, and are therefore strictly bounded by 1. For all execution orders, queue X can never contain more than two unconsumed tokens, and is therefore strictly bounded by 2. Kahn's simple example from Fig. 2.1 is strictly bounded by 2.

With the definitions for bounded and strictly bounded as provided, a Process Network can be bounded even if it has an unbounded number of queues such as in a recursively defined Process Network. If a Process Network is not bounded, then it is *unbounded*. That is, at least one channel is not bounded for all complete executions of the Process Network.

Any arbitrary Process Network program can be transformed to be strictly bounded by adding a feedback queue for each existing queue and changing every process's interaction with queues. This transformation is the key to Parks's bounded scheduling of Process Networks.

2.3.3 Scheduling for Boundedness

For some restrictive subsets of Process Networks such as Synchronous Dataflow [41][42], it is possible to compile a finite schedule that can be repeatedly executed. However, for general Process Networks the questions of termination and boundedness are undecidable in finite time and so the program must be scheduled dynamically. If the program being executed is non-terminating, then the dynamic scheduler conveniently has infinite time to determine these properties. Parks [36] lists two requirements for such a dynamic scheduler:

Axiom 2.3.1. *(Complete Execution) The scheduler should implement a* complete execution *of the of the Kahn Process Network program. If the program is non-terminating, then it should be executed forever without terminating.*

Axiom 2.3.2. *(Bounded Execution) The scheduler should, if possible, execute the Kahn Process Network program so that only a bounded number of tokens* *ever accumulate on any of the queues.*

When these requirements conflict, Parks' intention was to prefer a complete, unbounded execution to a partial, bounded execution. For Process Network programs that are bounded (which is a property of the program, not the execution order), *some* execution orders lead to bounded queue sizes. Parks [36] presents a scheduling policy intending to always execute bounded programs in bounded memory. Parks claims that terminating programs terminate as expected when using this scheduling policy. He also claims that in the case of an unbounded program, the scheduling policy will execute forever (or as long as memory is available) and not introduce deadlock. Counterexamples to Parks' scheduling policy (where it leads to a partial, bounded execution) were later presented [38], but his general approach is a seminal step toward bounded scheduling of Process Networks.

Any arbitrary Process Network program can be transformed to be strictly bounded by adding a feedback queue for each existing queue as in Fig. 2.10 and modifying each process [36]. Before a process can write a token to an existing queue, it must read a token from the associated feedback queue. When a process reads a token from an existing queue it also writes a token to the associated feedback queue. With this transformation, the initial number of tokens on the feedback queue strictly bounds the number of tokens that can ever be present on the existing queue. For example, if P_f initially contains one token, then A and B must alternate and P is strictly bounded by one.

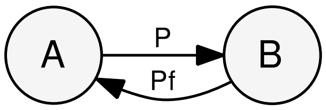


Figure 2.10: A feedback queue makes Fig. 2.9 strictly bounded.

However, this transformation may introduce the side effect of *artificial deadlock* and cause the program to terminate earlier than it would have without the transformation. This is in contrast to a *true deadlock* that a terminating program would reach naturally (and without the transformation).

Parks's approach is to transform the program graph G to produce the semantically equivalent feedback graph G^0 that is strictly bounded by b^0. If the program executes without stopping then a complete, bounded execution has been realized. If execution stops and the result is only a partial execution of the original program graph G, then we have reached artificial deadlock and the bound b^0 was too small. We must choose a new larger bound $b^1 > b^0$ and try again. If the program is bounded, we know that a bound b exists and is finite. This approach will eventually discover a bound $b^N \geq b$ where a complete execution of G^N corresponds to a complete execution of the original graph G. Parks [36] states:

> So we see that this bounded scheduling policy has the desired behavior for terminating and non-terminating programs, strictly bounded, bounded and unbounded programs. This is important because termination and boundedness are undecidable. There will

always be programs that we cannot classify, so our scheduling policy must have a reasonable behavior for all types of programs.

Parks points out that any scheduling policy can be used for the processes in the transformed graph, and any execution order will lead to bounded buffering on the queues if the original graph is bounded.

There is no reason to increase the capacity of every queue equally. If execution stops because of an artificial deadlock, then at least one process is blocked and attempting to write to a full queue. By increasing the capacity of one of these full queues, the execution can continue. Parks argues that it is sufficient to increase the full queue with the smallest capacity, and this will prevent queues from growing without bound. This approach can use significantly less memory than if the capacity of every queue is increased equally.

Rather than actually executing a transformed graph, this leads to a set of dynamic scheduling rules at each process in the system:

1. Block when attempting to read from an empty queue,
2. Block when attempting to write to a full queue, and
3. On artificial deadlock, increase the capacity of the smallest (deadlocked) full queue such that execution can continue.

Parks states that *global deadlock* of the network, in which no process is able to execute, is required for detecting and resolving an artificial deadlock. However, not all artificial deadlocks result in a global deadlock. Geilen and Basten

```
void A(int n, oqueue<int> P)
{
  while (n>0) {
    P.put( n );
    n -= 1;
  }
}
```

Figure 2.11: A function, A, that produces n output tokens and terminates.

show [38] an example where Parks' algorithm produces an incomplete execution, and also point out that it cannot schedule a PN graph composed of disjoint components.

Geilen and Basten argue that a dynamic deadlock detector for Process Networks must detect *local* deadlocks. If any cycle of processes is unable to proceed because of a queue of insufficient length in the cycle, then artificial deadlock should be declared and resolved by increasing the length of the smallest full queue in the cycle. This holds regardless of whether processes outside of the deadlocked cycle can execute.

Local deadlock detection increases the set of Process Network programs for which bounded scheduling will lead to a complete execution. However, Geilen and Basten argue that bounded scheduling cannot provide a complete execution for a large class of Process Network programs [38].

2.3.4 Complete Execution with Bounded Scheduling

Geilen and Basten argue that bounded scheduling will not yield a complete execution of a Process Network program unless all tokens that are produced in the program are also eventually consumed. They call such Process Network programs *effective*. This restriction can be particularly limiting for signal processing systems that use sliding window algorithms (and firing thresholds in general). Consider one way to implement an FIR filter of order N is to require $N + 1$ input samples to compute a single output sample. It would be expected that such a filter would terminate with N unconsumed tokens at its input. It seems reasonable that bounded scheduling with such a filter should not preclude a complete execution, despite not being effective.

I show that effectiveness is too strong of a restriction: there exist non-effective Process Network programs that will achieve complete execution when using bounded scheduling. In fact, there exist an infinite number of such programs. I produce a set of counterexamples to the requirement for effectiveness by defining processes A and B as shown in Figs. 2.11 and 2.12, and by using the trivial PN program in Fig. 2.9.

The Process Network program in which process A feeds process B (in Figs. 2.11 and 2.12) is not effective when $n > m$. When executed with (Kahn's original) unbounded scheduling rules, this program will terminate with $n - m$ unconsumed tokens. By using bounded scheduling and defining a queue bound b_P for queue P, the execution will be complete if $b_P \geq n - m$. In the case that $b_P < n - m$, process B will terminate before a complete execution has

```
void B(int m, iqueue<int> P)
{
  while (m>0) {
    P.get();
    m -= 1;
  }
}
```

Figure 2.12: A function, B, that consumes m input tokens and terminates.

been reached. Additionally, process A will be blocked and waiting on queue P when process B terminates. Parks's transformation to make a Process Network strictly bounded [36] does not discuss behavior for when a process terminates. When a consuming process terminates, it leaves a dangling queue without a consumer. It is reasonable to assume that a queue without a consumer is bounded by zero because it never needs to store a token. Also, a producer process need never block on a queue with no consumer. When process B in this example terminates, any tokens currently in the queue can be discarded. Process A can then execute to completion. Even though the queue bound is smaller than the number of unconsumed tokens, bounded scheduling has yielded a complete execution.

Clearly Parks' original specification for a (global) dynamic deadlock detector significantly reduces the set of Process Network programs for which bounded scheduling will lead to a complete execution. Even with a local dynamic deadlock detector, Geilen and Basten argue that to achieve a complete execution with bounded scheduling, a Process Network program must be effective. However, I have shown that there are an infinite number of non-effective programs that yield a complete execution. Geilen and Basten provide examples of Process Network programs for which bounded scheduling will not yield a complete execution [38]. In each of their examples, incomplete execution occurs because at least one process does not read from one of its inputs for an indefinite amount of time.

2.3.5 Fair Processes and Scheduler

I argue that to achieve a complete execution with bounded scheduling, a Process Network must be *fair*.

Definition 2.3.1. A Process Network is fair if it has a fair scheduler, and each of the processes in the network is fair with respect to its inputs and outputs.

Kahn [35] requires a (loosely) fair scheduler to achieve a complete execution even with unbounded queues:

> A parallel program can be safely simulated on a sequential machine, provided the scheduling algorithm is fair enough, i.e. it eventually attributes some more computing time to a process which wants it. If this algorithm is not fair however, the only thing that may happen is for the parallel program to produce less output than what could be expected. But what is produced is correct.

If the scheduler is fair, then eventually the network will converge to the unique least fixed point.

If each process in the network is fair with respect to its inputs and outputs, then bounded scheduling (with artificial deadlock resolution) will achieve a complete execution for a bounded program.

Definition 2.3.2. A process is fair with respect to its inputs and outputs if, when that process is making progress, it will eventually consume from each of its inputs and produce on each of its outputs.

Stated another way, a fair process cannot indefinitely neglect any of its inputs or outputs. A fair scheduler and fair processes will allow the network to proceed and eventually converge to the least fixed point.

This definition assumes that the process eventually has sufficient input tokens available and sufficient output free space available. If a process never progresses because it is blocked reading or writing a queue, that does not render it unfair. For example, a process that would access each of its queues in round-robin fashion is fair, even though it could be prevented from progressing by a deadlock condition. A process is also fair if it releases an input (or output) indicating that it will not read (or write) more data on that queue. A process that terminates is also fair because its queues can be released upon termination. A terminated process therefore does not impede progress of the network.

When a consumer indefinitely ignores an input, the associated queue may become full and indefinitely block the associated producer process. This blocked process prevents the Process Network from progressing toward a complete execution, as it would in the absence of bounded scheduling. When a producer indefinitely ignores an output, it can similarly impede progress toward a complete execution.

By requiring all processes to be fair, any full queues are eventually and periodically read from, and any such blocked producers will eventually make progress. As long as processes that are blocked on full queues can eventually make progress, the Process Network will eventually proceed and converge to the least fixed point. That is, given a fair Process Network program, a bounded scheduler will find a complete and bounded execution if one exists. For an unbounded Process Network, the required dynamic deadlock detector will repeatedly detect artificial deadlocks and grow queue bounds until memory is exhausted. Just as the Kahn Principle requires a fair scheduler [38] for a complete execution, bounded scheduling requires fair processes in the network for a complete execution.

An effective Process Network is typically also a fair one, so assertion of fairness for complete execution with a bounded scheduler is not in conflict with the conclusions of Geilen and Basten [38]. There exist pathological examples of Process Networks that are effective but not fair (e.g. no tokens are ever produced or consumed). However, an effective program typically must read from its inputs for it to consume all of the tokens that have been produced. There also exist Process Networks that are not fair, but can be executed to completion with a bounded scheduler. For example, a producer that never writes any outputs could be connected to a consumer that never reads any inputs. Even though both processes (and the Process Network) are unfair, no tokens will ever accumulate and execution progress will never be impeded.

It is sometimes easy to determine whether a process is fair or unfair. For example, a very large class of processes that implement common signal processing operations are fair because they read from their inputs, perform a computation, and write a result to their outputs. However, fairness is generally undecidable in finite time. Sometimes the behavior of a process depends on its input values (e.g. a selector), and the determination of fairness depends on the inputs to the process (and ultimately the entire network). Determining whether a Process Network is effective is also generally undecidable in finite time.

However, if the vision of Process Network is to model "infinite streams of data samples [that] are incrementally transformed by a collection of processes" [36], then a very large class of interesting problems can be solved using processes that are easily shown to be fair. Signal processing systems that include sliding window algorithms are likely to be fair but not effective. Having a complete execution with bounded scheduling for this large class of Process Network systems is an important result.

Even for fair Process Networks, bounded scheduling requires a correct and complete dynamic deadlock detector. Chapter 3 is dedicated to deadlock detection. Computation Graphs, discussed in the following section, are further evidence that non-effective programs can achieve complete execution with bounded (and even static) scheduling.

2.4 Computation Graphs

Computation Graphs is a model of computation developed by Karp and Miller [46], and Computational Process Networks (the subject of this dissertation) use the concept of firing thresholds from this model. The Computation Graphs model is similar to Process Networks in that it is a dataflow model and programs can be represented as a directed graph of compute nodes and communication channels. However, Computation Graphs are significantly less expressive than Process Networks. Because of restrictions in the model, Computation Graphs have necessary and sufficient conditions for both termination and boundedness, and a static execution schedule for a given program can be determined at compile time [46].

A Computation Graph consists of a finite set of nodes $n_1, \ldots, n_l$ and a set of queues $d_1, \ldots, d_t$. Each queue d_i is directed from one node n_p to another node n_c and behaves as a one-way FIFO. Each queue d_i has four non-negative constant integer parameters associated with it:

- A_i, the number of tokens initially present on queue d_i,
- U_i, the number of tokens inserted in queue d_i at each firing of the associated producer node n_p,
- W_i, the number of tokens removed from queue d_i at each firing of the associated consumer node n_c, and

- T_i, the number of tokens that must be present on queue d_i before the associated consumer node n_c can fire, where $T_i \geq W_i$.

This last parameter, T_i is called the firing threshold, and allows a node to require the presence of more tokens than it will consume upon firing. The concept of a firing threshold has applications in DSP systems, where it is common to have algorithms that operate on continuous, overlapping streams of data, or sliding window algorithms. Examples include filters and overlap-save FFT algorithms.

Consider one way to implement an FIR filter of order N is to require $N+1$ input samples to compute a single output sample. In this case, $T = N+1$ and $W = 1$ for the filter's input queue, and $U = 1$ for the filter's output queue. In this implementation, the filter node is memoryless; i.e. the N most recent samples need not be stored in the node state because they are present on the input queue. Synchronous Dataflow [41] is a special case of Computation Graphs in which the number of tokens consumed upon firing of an actor is always equal to the firing threshold ($W_i = T_i$).

2.5 Conclusion

This chapter covers Kahn's formal Process Network model [35]. It introduces his *get* and *put* semantics and recreates his illustrative sample program. It details Kahn's mathematical representation of Process Networks: functions that map streams onto streams. The formal consequences of this model are that the tokens in the queues are determinate and do not depend on execution order, including concurrent execution. The set of mathematical equations that describe a Process Network can be solved for the unique least fixed point, and this corresponds to the queue contents in an execution.

Kahn's model requires potentially infinite queues. This chapter also discusses execution of Process Networks in bounded memory by using clever scheduling and an online deadlock detector. Parks [36] introduces the general approach for bounded scheduling, and Geilen and Basten [38] refine it. Geilen and Basten also discuss which set of Process Network programs can be executed to completion using bounded scheduling techniques and call this set *effective*. I argue for a set of *fair* Process Network programs that can achieve complete execution with bounded scheduling. Fair programs are more likely to occur in signal processing systems that use sliding window algorithms.

Finally this chapter introduces Computation Graphs, a model similar to Process Networks, from which this dissertation leverages the concept of firing thresholds.

Chapter 3, which follows, is dedicated to deadlock detection. Deadlock detection is required for bounded scheduling of Process Networks.

Chapter 3

Deadlock Detection

In multi-tasking systems, multiple processes may compete for finite resources. If a process requests a resource that is not available (in use by a different process), it may be suspended and wait until the resource is available. Deadlock is a situation in which multiple processes are unable to proceed because each is waiting on one of the others in order to continue.

As discussed in Chapter 2, dynamic deadlock detection and resolution is required for a complete execution of Kahn Process Networks when using a bounded scheduler. Although deadlock detection is a widely studied subject, one must carefully select an appropriate distributed deadlock detection algorithm as it applies to bounded scheduling of Process Networks.

Section 3.1 introduces deadlock and wait-for graphs. Section 3.2 summarizes some previous work in distributed deadlock detection. Section 3.3 details the distributed dynamic deadlock detection and resolution (D4R) algorithm, suitable for use in a distributed Process Network implementation. Section 3.5 includes case studies that illustrate the effectiveness of the D4R algorithm. Section 3.6 concludes this chapter.

3.1 Introduction

A multi-tasking computer system consists of a finite number of resources to be shared among a number of competing processes [54]. Memory, CPU time, and I/O devices are all examples of resources. Resources can be partitioned into several types, each consisting of some number of identical instances (such as multiple CPUs in an SMP system). If a process requests a resource type, the allocation of any instance of the type will satisfy the request.

A process must request a resource before using it, and then release the resource when finished using it. During normal operation, a process may utilize a resource in only the following sequence:

1. *Request.* If the request cannot be granted immediately (e.g. the resource is in use by another process), then the requesting process must wait until it can acquire the resource.

2. *Use.* The process can operate on the resource.

3. *Release.* The process releases the resource, and it is now available for use by other processes.

In an operating system, these requests and releases of system resources are made through system calls.

A set of processes is in a deadlock state when every processes in the set is waiting for an event that can be caused only by another process in the set. Deadlock can arise if the four following conditions hold simultaneously in a system:

1. *Mutual exclusion.* At least one resource must be non-sharable; that is, only one process at a time can use the resource. If another process requests that resource, it must be wait until the resource has been released.

2. *Hold and wait.* A process must be holding at least one resource and waiting to acquire additional resources (that are held by other processes).

3. *No preemption.* Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it.

4. *Circular wait.* A set $\{P_0, P_1, \ldots, P_N\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

All four conditions must hold for a deadlock to occur [54].

There exist several models for processes requesting resources in a system [55]. The simplest model is *single-resource*, in which a process can have only a single outstanding resource request. In this model, a blocked process is waiting for only one other process. Other examples of resource request models include the AND and OR models. In the AND model, a process can simultaneously request multiple resources and the request is satisfied only after all of the

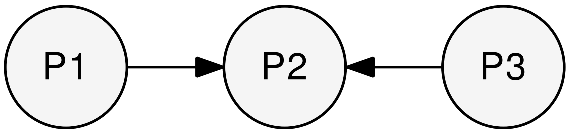


Figure 3.1: An example of a wait-for graph.

resources are acquired. In the OR model, a process can simultaneously request multiple resources and the request is satisfied if any one of the resources are acquired. The AND-OR model is a generalization of these models, in which a process may specify any combination of *and* and *or* for a resource request.

Given a set of processes that share resources, we can construct a *wait-for* graph, which is a directed graph that contains a node for every process. An edge indicates that a process is blocked and waiting for another process. The direction of the wait-for edges is from the waiting process to the process being waited on. Fig. 3.1 shows an example of a wait-for graph in which two processes P1 and P3 are waiting on process P2. In the single-resource model, the out-degree of a node in the wait-for graph can be at most one, and a cycle in the wait-for graph indicates a deadlock. In other models, the out-degree may be more than one, and a cycle may not always imply a deadlock.

In a Process Network, processes communicate only via the queues between them, and the resources that are being shared are the tokens that the processes are passing between them. When a process reads from an input queue, it is requesting a (token) resource and waits until it can acquire the resource. A process releases resources (for other processes to acquire) by writing tokens to an output queue. Although the set of resources is not fixed, processes still follow the cycle of request, use, and release for resources.

In the case of bounded scheduling (according to Parks' feedback transformation [36]), a process is requesting a resource when writing to an output queue: it is reading a token from the feedback queue that indicates space is available for writing. One could also consider a process to be waiting for the resource of free space in the output queue. Similarly, reading from a input queue releases the resource of a feedback token (or free space) in the queue.

The deadlock conditions hold for Process Networks. The mutual exclusion condition holds because queues have only a single endpoint from which tokens can be read and can therefore only be used by the single destination process. The hold and wait condition is also in effect for processes with both input and output queues; a processes, while waiting on tokens from an input queue, is holding tokens to be written to an output queue. In the case of bounded scheduling, hold and wait is in effect for a process with any two queues whether they are inputs or outputs. The non-preemption condition also holds because a process can only write to a queue voluntarily.

According to Kahn's definition [35], the Process Network model is a single-resource model. A process can only read from one input queue at a time and, if blocked, it is waiting for the producer process at the other end of the queue. In the case of bounded scheduling, a process that is blocked writing is waiting for the respective consumer process. Cycles in the wait-for graph imply a deadlock.

This indicates that we can apply general deadlock detection algorithms for the single-resource model to the problem of deadlock detection in a Process Network. The following section describes previous work in deadlock detection.

3.2 Previous Work

Deadlock prevention is inefficient and impractical in distributed systems, so deadlock detection in these systems is typically the best approach to handle deadlocks [55]. Deadlock detection is a very widely covered topic. Singhal [56] provides a survey of deadlock detection algorithms prior to 1990. The Process Network model is defined to be a single-resource model, so it is reasonable to focus on algorithms that fit this model.

Singhal [56] lists correctness criteria for deadlock detection algorithms:

1. *Progress (No undetected deadlocks):* the algorithm must detect all existing deadlocks in finite time.
2. *Safety (No false deadlocks):* the algorithm should not report deadlocks that do not exist.

Singhal states that it is difficult to design a correct distributed deadlock detection algorithm, and that many algorithms reported in the literature are incorrect [56].

Deadlock detection algorithms can generally be categorized as centralized or distributed. In a centralized algorithm, a designated control node is responsible for constructing the global state of the graph and searching it for cycles. The control node may maintain the global state at all times, or it may build it when deadlock detection is to be carried out. One method for constructing the global state is by taking a snapshot of the current state of the distributed system. Another method is for each node to send its local wait-for graph information to the control node, so that it can construct the global graph. Centralized algorithms are straightforward and easy to implement, because the control node contains the entire graph state. However, centralized algorithms suffer from bottlenecks at the control node (leading to limited scalability) and a single point of failure [56].

In a distributed deadlock detection algorithm, the responsibility of deadlock detection is shared equally among the nodes. The global state is not contained in any one place, but is spread over the many nodes. In such a distributed algorithm, there is no single point of failure and no bottleneck, so the algorithm is scalable. Potential disadvantages of distributed algorithms include deadlocks that are detected multiple times and in multiple nodes, and determining which of several distributed nodes should resolve a detected deadlock [56]. Proof of correctness of distributed deadlock detection algorithms is also said to be difficult [56].

Singhal [56] also mentions hierarchical algorithms, in which nodes are arranged hierarchically and a node detects deadlocks involving only its descendants. This has the advantages of distributed algorithms while attempting to reduce overhead and improve efficiency.

A common method used in distributed deadlock detection algorithms is *edge-chasing.* In edge-chasing algorithms, special messages called *probes* are sent along the edges of the wait-for graph. (In a Process Network program, the edges of the wait-for graph are coincident with edges in the Process Network). When a node blocks, the node will send a probe message along each outgoing edge. (In Process Networks and other single-resource models, the wait-for graph can have only one outgoing edge.) If a node is not blocked, it will discard any probe messages that it receives. When a node receives a probe message that it previously sent, the wait-for graph must contain a cycle. This condition indicates a deadlock. In edge-chasing algorithms, probe messages can typically be a very short fixed size.

Mitchell and Merritt [40] provide a simple example of an edge-chasing distributed deadlock detection algorithm for the single resource model. This algorithm was originally developed for use in distributed database systems. In the Mitchell and Merritt algorithm, each node has two labels: one public and one private. Probe messages sent along the wait-for graph are a fixed size and contain only the public and private labels. Probe messages may propagate upstream in the wait-for graph, until a cycle is detected. A useful feature of this algorithm is that only one process in the cycle will detect the deadlock. Mitchell and Merritt show that a deadlock cycle is detected in $N-1$ steps of the algorithm, where N is the number of nodes in the deadlock cycle.

Singhal [56] shows the Mitchell and Merritt algorithm to be one of several that requires a worst case of N^2 messages to detect a deadlock (again where N is the number of nodes in the deadlock cycle). In fact, Kshemkalyani and Singhal [57] show that any edge-chasing algorithm in which nodes are constrained to exchanging messages with immediate neighbors in the wait-for graph take a worst case of $O(N^2)$ messages to detect a deadlock. More sophisticated distributed deadlock detection algorithms than that of Mitchell and Merritt exist, such as Chandry-Misra-Haas [58] or Kshemkalyani-Singhal [57]. However, these algorithms are attempting to address deadlock detection in models that are more general than the single resource model. Given that their added complexity gives no asymptotic improvement in performance when used with the single resource model of Process Networks, the Mitchell and Merritt algorithm appears to be the more reasonable choice for deadlock detection in Process Networks.

An algorithm by Prieto, Willadangos, Farina, and Corboda [59] can detect a deadlock cycle with only $O(N)$ messages. However, this algorithm requires that every node in the deadlock cycle can communicate directly with any other node in the cycle. This approach is effectively reducing the number of message by requiring N^2 connections. Huang and Qi present [1] a deadlock detection and resolution algorithm based on the algorithm of Prieto *et al.* [59]. Generally, the nodes in a Process Network will not be fully connected and such an approach would require significantly more connections than an edge-chasing algorithm. Again, the Mitchell and Merritt algorithm appears to be a more appropriate choice for deadlock detection in Process Networks.

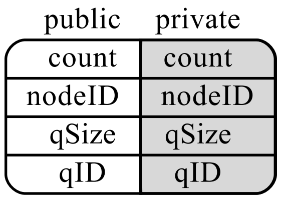
Olsen and Evans [60] are the first authors known to apply the Mitchell and Merritt algorithm [40] to deadlock detection in a Process Network. The application presented by Olson and Evans could detect whether a deadlock was present, but it did not specifically locate or resolve the deadlock.

Section 3.3 presents an algorithm based on a different Mitchell and Merritt algorithm that both detects and resolves deadlocks using process priorities [40]. I call this algorithm the distributed dynamic deadlock detection and resolution algorithm, or D4R algorithm.

3.3 D4R Algorithm

Mitchell and Merritt present a second algorithm [40] that both detects and resolves deadlocks. This second algorithm uses process priorities, and identifies the lowest priority process in the deadlock cycle so that it can be resolved. The D4R algorithm assigns the process priorities such that it can be determined whether a deadlock is real or artificial, and can identify the node that is blocked writing to the queue that must be lengthened in order to resolve the deadlock.

The wait-for graph of the D4R algorithm coincides with the nodes and edges of the Process Network graph being monitored for deadlock. In the wait-for graph, an edge indicates that a process is blocked and waiting on a single other process. The direction of the wait-for edges is from the waiting process to the process being waited on. For nodes that are blocked writing, the direction of the wait-for edge is the same as in the original PN graph. For nodes that are blocked reading, the direction of the wait-for edge is opposite



public	private
count	count
nodeID	nodeID
qSize	qSize
qID	qID

Figure 3.2: D4R algorithm state data at each node.

that in the original PN graph. State for the D4R algorithm is a small table present at each node.

3.3.1 Algorithm Description

The algorithmic state data used for D4R is shown in Fig. 3.2, consisting of public and private sets of four unsigned numbers: a non-decreasing counter *count*, a unique node identifier *nodeID*, a queue size variable *qSize*, and a second unique identifier *qID*. Each process is initialized with public and private sets equal. The public set changes as the algorithm progresses, but the private set remains unique to that node.

Note that *count* and *nodeID* are combined into a single variable in the algorithm by Mitchell and Merritt. The use of these variables in D4R is consistent with the suggestion of "keeping the low-order bits of the label constant and unique while increasing the high-order bits when desired [40]." The notation *count:nodeID* is used to show these variables concatenated into one. The variables *qSize* and *qID* are also combined in the algorithm by Mitchell and Merritt and serve the function of the priority variable for deadlock resolution.

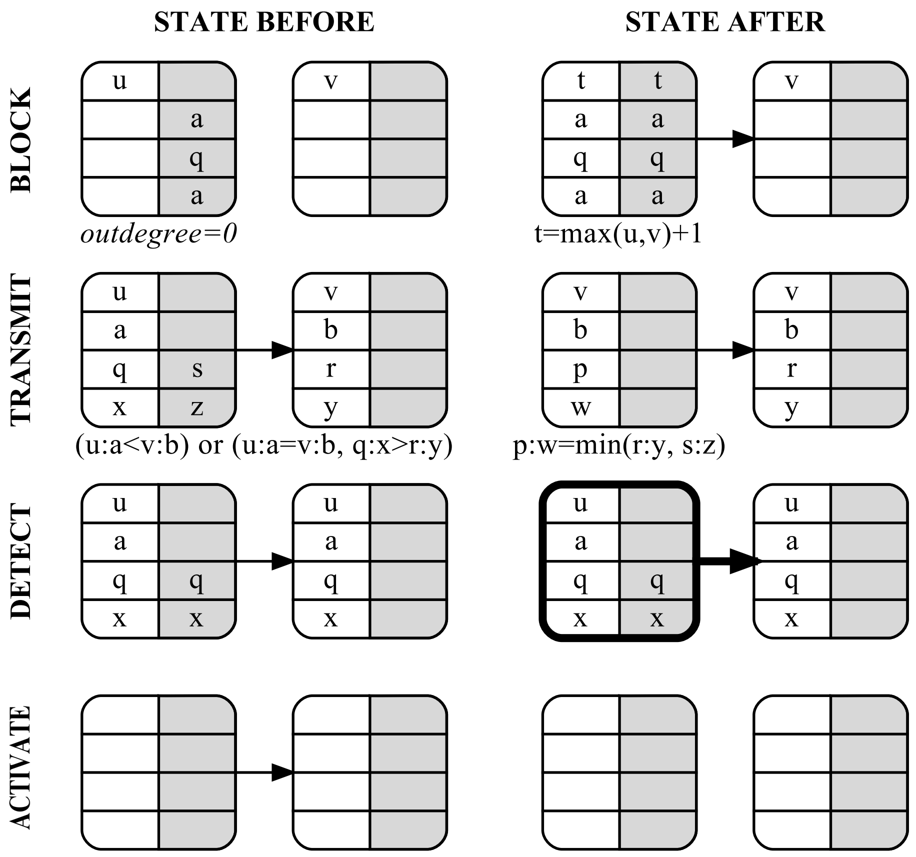


Figure 3.3: State transitions for the D4R algorithm.

The notation *qSize:qID* is used to show these variables concatenated into one.

The node state data and wait-for edges define the state of the D4R algorithm at any time. Fig. 3.3 shows the possible state transitions for this algorithm in the order that they occur: *Block*, *Transmit*, *Detect*, and *Activate*. State data that is unchanging or unused in a transition has been left blank.

The *Block* state transition occurs when a Process Network node blocks on a queue, creating an edge in the wait-for graph. The blocking node's state data is fully initialized, and its *count* variables are incremented to be greater than that of both nodes. The *qSize* variable is also initialized during this state transition. The *qSize* is what permits the D4R algorithm to determine whether a deadlock is artificial, and localize the smallest full blocked queue. The variable *qSize* is set as follows: when a process blocks on a write, set *qSize* to the size of the blocking (full) queue; when a process blocks on a read, set *qSize* = MAX_UINT. Note that this is MAX_UINT for the word size of the variable, the same as if the variable were set to -1: all bits are set. Clearly MAX_UINT must be larger than any actual queue size in the implementation, e.g. a 64-bit variable on a 64-bit architecture. The variables *nodeID* and *qID* are both initialized to the same unique number that identifies the node in the Process Network. Although *nodeID* and *qID* are initialized to the same value, they will change to different values as the algorithm progresses.

The *Transmit* state transition occurs when a waiting process detects a change in the public state of the node upon which it is waiting, and certain criteria are met: if the other node's public *count:nodeID* is larger than its own, or if they are equal and *qSize:qID* is smaller. If the criteria are met and the state transition occurs, then the waiting node replaces its public *count:nodeID* with the one it just read, and sets its public *qSize:qID* to the minimum of the two nodes. In implementation, each time a node's public state changes, it will notify any dependent nodes. The effect is that larger *counts* and smaller *qSizes* migrate along the edges of the wait-for graph, in the opposite direction.

The *Detect* state transition occurs when a waiting process sees that its entire public set matches that of the node upon which it is waiting, its public and private *qSize* match each other, and its public and private *qID* also match each other. It then knows that it is not only a part of a deadlock cycle, but that it also has the smallest *qSize* of any node in the cycle. (If multiple nodes have the same *qSize*, the node with the smallest *qID* will break the tie.) Only one process will detect the deadlock, and the value of *qSize* tells the type of deadlock: if *qSize* is MAX_UINT, this is a real program deadlock (and all nodes in the cycle are blocked on reads); otherwise this is an artificial deadlock, and the smallest, blocked, full queue has been identified. This is precisely the queue that must be lengthened for correct bounded scheduling of Process Networks [38].

The *Activate* state transition may occur after *Detect*. If the deadlock was real, the program has terminated. If it was artificial, the culpable queue has been lengthened so that the Process Network program may continue. Of course, *Activate* will also repeatedly occur after *Block* as the Process Network proceeds normally in the absence of any deadlock.

3.3.2 Algorithm Proof

Portions of the D4R algorithm have intentionally been made equivalent to the Mitchell and Merritt priority-based algorithm. The following theorem is therefore included from their paper [40] without further proof.

Theorem 3.3.1. *If a cycle of N nodes forms and persists long enough, the lowest priority process (with the smallest* qSize:qID*) in the cycle will execute* *the* Detect *step after at least* $N-1$ *and at most* $2N-2$ *consecutive* Transmit *steps.*

Mitchell and Merritt assign a fixed priority to each node, whereas the D4R algorithm dynamically sets *qSize* and *qID* each time the *Block* state transition is executed. It must therefore be shown that this dynamic updating in the D4R algorithm does not violate the rules of the original algorithm.

Lemma 3.3.2. *If a node has an outdegree of 0 in its wait-for graph, it can change the value of its private* qSize *and* qID*. That is, a node's private* qSize *and* qID *need only be fixed when it has non-zero outdegree in the wait-for graph.*

Proof. The private *qSize* and *qID* of a node is unused unless the node has a non-zero outdegree. No other node can access these private variables at any time. The deadlock detection algorithm will therefore proceed unaffected. □

Theorem 3.3.3. *A node can change its* qSize *and* qID *during its* Block *state transition (both public and private).*

Proof. Just prior to a *Block* state transition, a node must have an outdegree of 0 because this is a single-resource algorithm, and a process can only block on a single queue. By Lemma 3.3.2, the node can set its private *qSize* and *qID* at this time. Prior to executing the *Block* state transition, the public *qSize* and *qID* are unused. During the *Block* state transition, a node's public *qSize:qID* is initialized with its private *qSize:qID*. □

I have shown that I can correctly schedule bounded Process Networks using the D4R algorithm as described in Section 3.3. The D4R algorithm modifies that of Mitchell and Merritt [40] to set the *qSize* based on the size of the queue that a node is about to block on, and whether that node is blocking on a read or a write.

As a further discussion topic, a *count* that continuously increases is not implementable for a program that never terminates. I wish to examine the possibility of periodically resetting the *count* variables to prevent "rolling over".

Proposition 3.3.4. *If a node has an outdegree of 0 and an indegree of 0 in the wait-for graph, it can change its public and private* count *variables. The* nodeID *variable is still unique and unchanged.*

Proof. The *count* variables are only used when the node has non-zero outdegree or indegree. When both are zero, it is as if the node has never been in a wait-for graph. Adding an arc to the wait-for graph requires a *Block* step, which will increment the *count* variables as necessary for the algorithm. □

While it is easy to determine that outdegree is 0, it is not obvious how to inexpensively and easily determine that indegree is 0. For the current implementation, the *count* variables simply must be large enough to prevent rollover in any reasonable amount of time.

3.4 D4R Implementation

In addition to proving correctness, the presented D4R algorithm is implemented in the Computational Process Network framework described in Chapter 5. It is important to point out that this D4R algorithm can also be applied to bound scheduling of general (not "Computational") Process Networks. Goals for the CPN framework include high performance and very low overhead, with the ability to handle high-throughput streams of data for signal processing.

Performance is intentionally not a goal for the implementation of the D4R algorithm. Artificial deadlock in a program is an undesired state, and considered an exception to normal operation. In a real deadlock, as opposed to an artificial deadlock, that portion of the program has terminated. In any case where there is a performance tradeoff between the D4R algorithm and normal queue operation, faster normal queue operation is preferred. This makes the framework faster and lower overhead for programs where the minimum queue lengths have already been determined.

The CPN framework implementation permits disabling of the D4R algorithm in order to reduce overhead. This may yield an incomplete execution for some programs, but can be beneficial when the queue sizes are already known to be sufficiently large to prevent artificial deadlock.

Currently, when a node blocks on another node it immediately sends a D4R message to its peer to indicate that it is attempting to perform the *Block* state transition. Sometimes, this blocked state will simply resolve itself as the peer node makes more data (or free space) available in the adjoining queue. By waiting a small delay time before sending a block message, the D4R messaging overhead would be reduced and the latency of detecting deadlocks would be increased. The longer the delay, the lower the overhead but the greater the latency. By varying this delay time, it could be possible to adjust between the tradeoffs of overhead and latency in the D4R implementation. The approach of reducing overhead by delaying the start of the deadlock detection algorithm has been explored in the literature [61]. This is not currently implemented in D4R for the CPN framework. The delay time currently takes on two effective values: zero and infinity. When D4R is enabled, the delay time is zero. This yields maximum D4R messaging overhead and the minimum latency for deadlock detection. When D4R is disabled, the delay time is effectively infinity. This yields no D4R messaging overhead and infinite latency for deadlock detection.

The presented D4R algorithm can dynamically detect and resolve artificial deadlocks. It also detects real deadlocks, which indicate that some local portion of the program will make no further progress. The CPN framework therefore correctly schedules bounded, fair Process Networks in bounded memory. Because the presented D4R algorithm is based completely on local data between connected nodes (and do not require any global synchronization) implementation on a distributed system is straightforward and scalable.

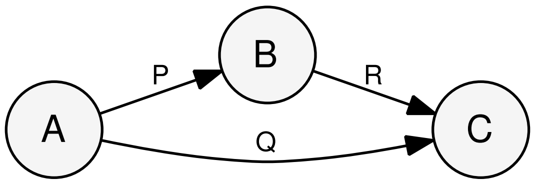


Figure 3.4: A simple artificial deadlock detection and resolution example.

3.5 D4R Case Studies

To demonstrate that the D4R algorithm successfully detects and resolves deadlocks, a series of case studies have been implemented within the CPN framework. These case studies include both artificial deadlocks (which can be resolved by growing the correct queue and continuing) and a real deadlock (which indicate that a portion of the program has terminated). Also included is an unbounded program.

The first case study is a simple 3-process artificial deadlock detection and resolution, as shown in Fig. 3.4. This example is also available online in animated form [62]. The processes A, B, and C are defined in Figs. 3.5 to 3.7. In this example, the queues P, Q, and R all begin empty and with a capacity of one sample. The nodes in this example could proceed in any of a large number of execution orders. However, in any of these execution orders the D4R algorithm detects that Q is preventing A from executing because Q is not sufficiently large. To resolve this artificial deadlock, Q must grow to size two, and then the network can proceed unhindered.

The following is a possible execution order for the steps of deadlock

```
void A(oqueue<int> Q, oqueue<int> P)
{
  while (true) {
    Q.put(1);
    Q.put(1);
    P.put(1);
  }
}
```

Figure 3.5: Implementation of node A from Fig. 3.4.

```
void B(iqueue<int> P, oqueue<int> R)
{
  while (true) {
    int x = P.get();
    R.put(x);
  }
}
```

Figure 3.6: Implementation of node B from Fig. 3.4.

```
void C(iqueue<int> Q, iqueue<int> R)
{
  while (true) {
    R.get();
    Q.get();
    Q.get();
  }
}
```

Figure 3.7: Implementation of node C from Fig. 3.4.

detection and resolution in this example.

1. *Initial conditions:* all queues are empty and have capacity of one.

2. Node A executes and produces a single token on queue Q.

3. A attempts to produce another token, but Q is full. A initializes its D4R state variables as it *Blocks* on C. *qSize* is 1 because A is blocked writing and Q has a capacity of 1.

4. B attempts to read from P, which is empty. B initializes its D4R state variables as it *Blocks* on A. *qSize* is MAX_UINT because B is blocked reading.

5. C attempts to read from R, which is empty. C initializes its D4R state variables as it *Blocks* on B. *qSize* is MAX_UINT. Note that there is now a cycle in the wait-for graph: $C \rightarrow B \rightarrow A \rightarrow C$.

6. C performs *Transmit* of the public D4R variables upstream (in the wait-for graph) to A. A now has the public D4R variables from C, except it keeps $qSize = 1$, the minimum of the queue sizes.

7. A performs *Transmit* of the public D4R variables upstream to B. B now has the public D4R variables from A.

8. B performs *Transmit* of the public D4R variables upstream to C. C now has the public D4R variables from B.

9. A *Detects* that its D4R variables match those of C. Because *qSize* is one, this is an artificial deadlock. A is blocked on the culpable queue, Q, which must grow in order to resolve the deadlock.

10. Q grows to size two so that A can proceed.

11. A *Activates* and proceeds by producing a token on Q and a token on P.

12. B *Activates* and proceeds by consuming from P and producing on R.

13. C *Activates* and proceeds by consuming from R and twice from Q.

14. Each queue is again empty, and each node has returned to its beginning.

The artificial deadlock has been detected and resolved. The system can now execute repeatedly without further deadlock.

A second case study is the detection of a real (as opposed to artificial) deadlock as shown in Fig. 3.8. In this example, each node attempts to read one token from its input and then writes that token to its output. However all of the queues are empty, so none of the nodes can ever proceed. As in the previous example, each of the nodes will execute the *Block* and *Transmit* stages of the D4R algorithm. Eventually one of the nodes will *Detect* its D4R variables in the peer that it is blocked on, and declare that a deadlock cycle has been detected. However in this example, the detected *qSize* is MAX_UINT. This indicates that all processes in the deadlock cycle are blocked reading, and that the deadlock cannot be resolved. This is a real deadlock, and the program has terminated.

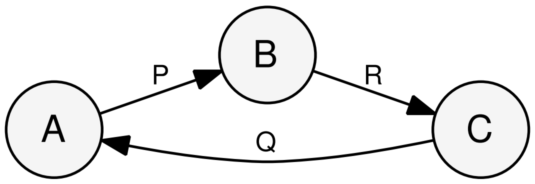


Figure 3.8: An example with real (as opposed to artificial) deadlock.

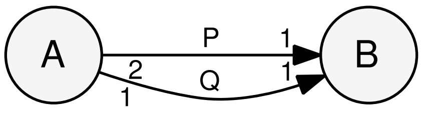


Figure 3.9: An unbounded example that will grow indefinitely.

A third case study is of an unbounded program as shown in Fig. 3.9. In this example A produces 2 tokens on P and 1 token on Q, and then repeats. B consumes 1 token from P and 1 token from Q, and then repeats. Clearly tokens will accumulate on P. In synchronous dataflow, this would be called an unbalanced program [41]. However, it is a valid construct in process networks, which can be unbounded.

As this example executes, the D4R algorithm will repeatedly detect an artificial deadlock a A, which is blocked writing to P. This artificial deadlock will be resolved by growing P so that A can continue. This cycle will repeat, and P will grow indefinitely, until memory is exhausted.

Two examples from the literature on bounded scheduling of process networks are also included as case studies. Fig. 3.10 from is a simple example of artificial deadlock from [1], but with 8 nodes. In this example, the queue

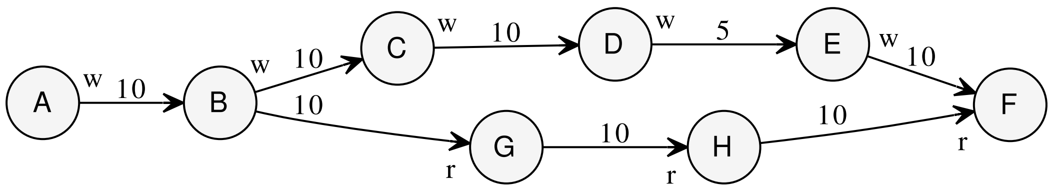


Figure 3.10: An example of artificial deadlock from [1].

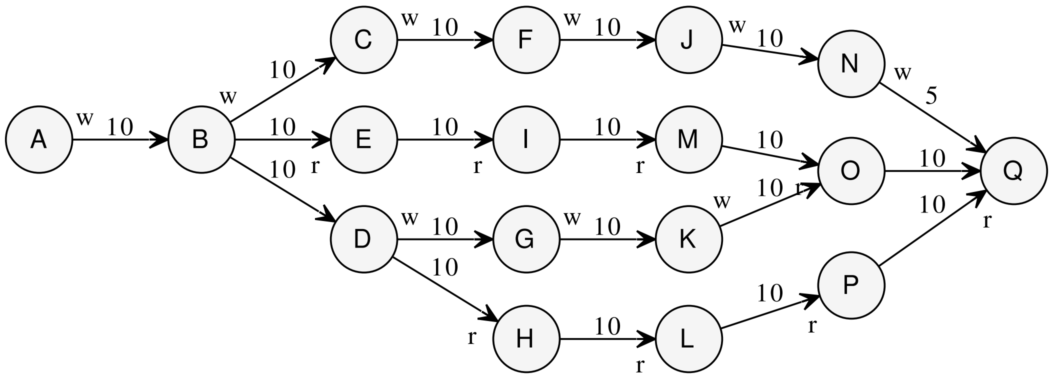


Figure 3.11: A seventeen-node deadlock resolution example from [2].

between nodes D and E will grow to resolve the artificial deadlock. Fig. 3.11 from is an example from [2] in which artificial deadlock must be detected and resolved multiple times. First, the queue between nodes N and Q must grow. Next, some queue between D and O must grow.

Basten and Hoogerbrugge [2] discuss how Parks' approach of increasing the smallest full queue may sometimes use more memory than necessary, because the smallest queue may not need to grow for the process network to generate more output. Kahn [35] defines the state of the process network as the state of all the tokens on all of the queues, so Parks' approach is making

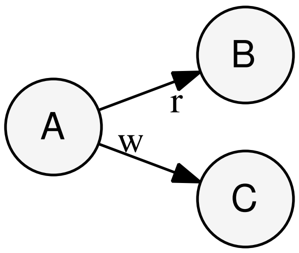


Figure 3.12: An example of artificial deadlock without a cycle.

progress according to Kahn's definition, even if not producing more output.

Basten and Hoogerbrugge [2] also show examples of artificial deadlock in a process network without a cycle in the wait-for graph. In Fig. 3.12, B cannot produce any more output until C consumes some input and allows A to progress. In the absence of bounded queues (as in Kahn's original model), B would be produce this output regardless of the behavior of C. In Section 2.3, I assert that a complete execution with bounded scheduling requires fair processes. If C is fair, then C will eventually read its inputs, and A and B will eventually produce the same outputs given by the Kahn Principle.

The D4R algorithm is based on cycle detection, a necessary condition for the classical definition of deadlock. As currently specified, the D4R algorithm does not detect artificial deadlocks that do not include a cycle. In the presented examples [2], there is a wait-for chain from an output (sink) node through at least one process that is blocked on a write. Much of the information needed to detect this condition is already included in the state variables maintained and exchanged by the D4R algorithm. If nodes have knowledge that they are an endpoint (source or a sink) of the process network, I believe that it is possible to extend the D4R algorithm to detect artificial deadlocks that do not include a cycle. If there exists a wait-for chain from a sink through a node blocked on a write, there could be an artificial deadlock that can be resolved. The smallest write-blocked queue in that wait-for chain could be increased so that the dependency is resolved and additional output can be produced. However, given that determining fairness is generally undecidable in finite time, this may also lead to unnecessary increases in queue size. Development of such an extension to the D4R algorithm would be future work.

3.6 Conclusion

This chapter covers deadlock detection both for general multiprocessing, and as it relates to complete executions with bounded scheduling of process networks. Deadlock can arise if four conditions hold simultaneously in a system: mutual exclusion of a resource, holding a resource while waiting to acquire other resources, no preemption of processes that are held, and a cycle of processes blocked waiting for each other to acquire a resources. Processes networks are an example of the single-resource model, in which a process can block waiting for only a single other process. In the single-resource model, a cycle in the wait-for dependency graph represents a deadlock.

Deadlock detection algorithms have been well-developed in the literature, with different algorithms suited for different models and using different approaches. One simple approach is edge-chasing, where processes send small probe messages along edges of the wait-for graph. When a message comes back to its origin, a deadlock cycle has been detected. Mitchell and Merritt [40] provide a simple example of an edge-chasing distributed deadlock detection algorithm for the single resource model.

I present the distributed dynamic deadlock detection and resolution algorithm, or D4R algorithm. This algorithm is based on the algorithm of Mitchell and Merritt [40], but is designed for use with a dynamic scheduler in Process Networks. Not only does this algorithm detect deadlocks, it can determine whether a deadlock is artificial and, if so, identify the queue that must be lengthened in order to resolve the detected deadlock. I describe the D4R algorithm in detail, and provide proofs of correctness. The D4R algorithm dynamically assigns node priorities based on queue sizes in a wait-for graph. In an artificial deadlock, the process with the lowest priority is located and known to be blocked on the culpable queue. I also discuss implementation of the D4R algorithm as included in the Computational Process Network framework described in Chapter 5. The D4R algorithm is suitable for use with Process Networks and Computational Process Networks.

I present case studies of the D4R algorithm that are used to demonstrate correct behavior. This includes artificial deadlock, real deadlock, an unbounded program, and examples from the literature.

Chapter 4, which follows, details the Computational Process Networks model, and how it relates to Kahn's model of Process Networks.

Chapter 4

Computational Process Networks

Kahn's Process Network model [35] provides determinism, scalability, and composability for a concurrent model of computation. Having these properties can allow rapid development of determinate, scalable concurrent systems built from composable components. This can allow the developer to focus on the task at hand, with less concern about the complications of concurrent and distributed systems.

I propose the Computational Process Network model, which is based on the formalisms of Kahn's PN model, but with enhancements that are designed to make it efficiently implementable. The Computational Process Network model preserves the formal properties of Process Networks, while reducing the operations required to implement algorithms operating on overlapping continuous streams of data commonly found in digital signal processing systems, as mentioned in Section 1.4.3. Using CPN, nodes implementing such algorithms can be memoryless because the state is retained on the queues. The PN model naturally models functional parallelism, but would typically use scattering and gathering process nodes to achieve data parallelism. CPN includes multi-channel queues as an aid for systems with data parallelism. CPN also uses bounded scheduling (in Section 2.3) with the D4R algorithm (in Section 3.3) so as to achieve complete execution in bounded memory where possible.

Section 4.1 introduces the enhancements that the CPN model provides to Kahn's PN model. Section 4.2 describes the communication semantics for the CPN model. Section 4.3 shows how the CPN semantics preserve the formal properties of the PN model. Section 4.4 shows how the CPN model semantics can improve performance over the KPN model. Section 4.5 concludes this chapter.

4.1 Introduction

CPN makes enhancements to the original KPN model to make it efficiently implementable while still preserving the formal properties of determinism, scalability, and composability. These enhancements include multi-token transactions to reduce execution overhead, multi-channel queues for multi-dimensional synchronous data, and firing thresholds for queues as both consumers and producers.

Multi-token transactions are a straightforward extension to Kahn's original semantics for the Process Network model, but can provide a significant performance enhancement as the overhead for a queue transaction is amortized. Multi-channel queues permit straightforward modeling of systems with multiple synchronous channels, such as audio streams in a surround sound system, or beams in a sonar beamforming system.

Consumer firing thresholds is a concept borrowed from Computation Graphs, discussed in Section 2.4. This approach has applications in high-throughput DSP systems, where it is common to have algorithms that operate on continuous, overlapping streams of data, or sliding window algorithms. Examples include filters, and sliding window fast Fourier transform (FFT) algorithms. For example, a finite impulse response (FIR) filter of order N requires $N + 1$ input samples to compute a single output sample. Computing the next output sample would require reuse of the N previous samples and one new sample.

If such a filter were modeled as a Kahn Process Network, the node implementing the filter would be required to store these N newest samples internally as process state. With firing thresholds, a node can require more tokens to be present on a queue than it will actually consume upon execution. For example, a filter node could require that $N + 1$ samples be present on a queue before executing, and then consume only a single input sample. The N previous samples are still present in the queue, so it is not necessary to store their state in the filter node.

Firing thresholds allow the decoupling of computation from communication when developing process nodes, and permit a zero-copy queue implementation. This decoupling also permits efficient modeling of algorithms on overlapping continuous streams of data, and permits implementation with memoryless nodes. This simplifies implementation of composable components that compute sliding window algorithms, because all of the overlap state is

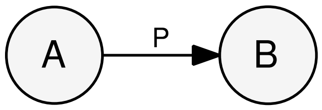


Figure 4.1: A simple Process Network program.

retained on the arcs. Such a model can also enable more efficient execution for these types of algorithms, which are very common in embedded real-time signal processing systems.

CPN also provides the dual to consumer firing thresholds, which is firing thresholds for producers. A node can access more free space than it will fill, thereby allowing variable-rate outputs without requiring data copies. These thresholds allow the decoupling of computation from communication when developing process nodes, and permit a zero-copy queue implementation. By eliminating unnecessary data copying in high-throughput systems, the computer is free for additional computation tasks. CPN is useful for modeling signal processing systems of streaming data, and naturally captures concurrency in these systems.

4.2 The Semantics of CPN

Recall from Section 2.2 that Kahn's semantics for communication are the read function *get* and the write function *put*. When a processes wishes to receive a token from an input queue, it calls *get* on the queue. If the queue is empty, *get* will block until data is available. When a process wishes to send

```
void A(oqueue<int> P) {
  while (true)
    P.put( 0 );
}

void B(iqueue<int> P) {
  while (true)
    P.get();
}

main() {
  queue<int> P;
  process<A>( P );
  process<B>( P );
}
```

Figure 4.2: A program that implements the Process Network of Figure 4.1.

a token to an output queue, it calls *put* on the queue with the argument of the value to send. In Kahn's original semantics, *put* could never block, but bounded scheduling (Section 2.3) allows for queues that can become full and force producers to block. Queues carry a particular type of data, such as the primitive type *int*.

In Section 2.2, I describe Kahn's semantics with object-oriented C++ using template data types. For example, template class `iqueue<T>` models the semantics of an input queue and provides only the method `get()` that returns a token. Template class `oqueue<T>` models the semantics of an output queue and provides only the method `put(T)` that sends a token. The class `queue<T>` inherits from both `iqueue<T>` and `oqueue<T>`, so that the declaration `queue<int> Q` creates a queue that carries integers. With these semantics, the Process Network program in Figure 4.1 can be completely implemented by the program in Figure 4.2. Process A produces a continuous stream of zero tokens and sends them over the queue P. Process B consumes the tokens from queue P. These semantics recreate Kahn's original semantics, and permit sending and receiving (only) a single token per queue transaction.

4.2.1 Extending Kahn's Semantics

It is trivial to extend these semantics to send multiple tokens per queue transaction. For class `iqueue<T>`, I define a new method `read(T*,unsigned)`. The method `read` takes a pointer and length, like a typed version of the traditional *read* system call in POSIX [6]. For class `oqueue<T>`, I define a new method `write(const T*,unsigned)`. The method `write` also takes a pointer and length, like the function in POSIX. These new methods are simply implemented as a *for* loop which repeatedly calls Kahn's original *get* or *put*. This is a simple extension to the communication semantics, but the performance consequences can be profound because the overhead for a queue transaction is amortized over many tokens.

Figure 4.3 shows the processes from Figure 4.1 using multi-token transactions with the semantics of *read* and *write* in place of *get* and *put*. Note that processes A and B need not operate on the same number of tokens per transaction. In fact, each transaction of each process could be with a different number of tokens.

```
void A(oqueue<int> P) {
  int output[4] = { 0, 0, 0, 0 };
  while (true)
    P.write( output, 4 );
}

void B(iqueue<int> P) {
  int input[5];
  while (true)
    P.read( input, 5 );
}
```

Figure 4.3: Process Network functions that use multi-token transactions.

Semantics for multi-channel queues is perhaps less obvious, having no direct parallel in POSIX. However it can provide a performance enhancement similar to multi-token transactions, and could be a common construct in multi-dimensional digital signal processing. Multi-channel queues carry multiple synchronous channels within a single queue, such as multiple audio channels in a surround sound system. A token in a multi-channel queue simply consists of multiple primitive values. For example, a multi-channel queue containing 8 channels of integers could be declared with `queue<int[8]>`, and be used to carry 7.1 surround sound audio samples. The semantics of *get*, *put*, *read*, and *write* could be used in this manner, although multi-token transactions on multi-channel queues must have the input or output data organized in a very specific manner.

Computation Graphs [46] (and CPN) have firing thresholds, where a process can access more tokens in a queue than it will consume. This could be implemented by extending the traditional POSIX *read* to have two separate length parameters: the threshold number of tokens to read into the user's buffer, and the number of tokens to discard from the stream. However, the CPN semantics avoid this approach in favor of zero-copy semantics.

4.2.2 Zero-copy Semantics

For high-throughput systems, *read* and *write* present an additional problem: they are based on copy semantics. When *read* is called, data must always be copied from where it currently resides into the buffer requested by the caller. For the program in Figure 4.1 to communicate using *read* and *write*, process A must copy data to queue P, and then process B must copy it out of queue P. For systems moving significant amounts of data, the copying overhead can be significant.

Consequently, zero-copy semantics have been explored in an effort to improve efficiency of I/O systems [63][64]. One approach has been for peers to exchange preallocated shared memory buffers, so that sending or receiving a buffer requires only a small pointer transaction instead of copying the entire buffer contents. This approach requires two steps for each zero-copy transaction. One example [64] implements *read* in the following two steps: one to request a full buffer (uf_read), and one to release that buffer (uf_deallocate) once its data is no longer needed. The implementation of *write* is also in two steps: one to request an empty buffer (uf_allocate), and one to send that buffer

```
template <typename T>
class cpn_iqueue {
  ...
  public:
  const T* GetDequeuePtr(uint thresh, uint chan=0);
  void Dequeue(uint count);
};

template <typename T>
class cpn_oqueue {
  ...
  public:
  T* GetEnqueuePtr(uint thresh, uint chan=0);
  void Enqueue(uint count);
};
```

Figure 4.4: Basic input and output semantics for CPN.

(uf_write) once filled.

4.2.3 The Semantics of CPN

CPN uses semantics that provide for both firing thresholds and a zero-copy interface, as well as multi-token and multi-channel transactions. As previously in [64], *read* and *write* are separated into two steps. To implement a *read* operation, CPN uses *GetDequeuePtr* followed by *Dequeue*. To implement a *write* operation, CPN uses *GetEnqueuePtr* followed by *Enqueue*. Figure 4.4 provides portions of the class declarations that describe the basic CPN queue interface semantics. The semantics of CPN are as follows:

`const T* cpn_iqueue<T>::GetDequeuePtr(uint thresh, uint chan=0)`

The parameter *thresh* is the (read) firing threshold in tokens. *GetDequeuePtr* blocks until at least *thresh* tokens are available for consumption in the referenced input queue. Once sufficient tokens are available, *GetDequeuePtr* returns a pointer to the tokens, which can then be treated as a sequential array. This pointer is *const* to prevent modification of tokens in an input queue. If the referenced queue has multiple channels, the parameter *chan* identifies which channel the return pointer references. If any channel in a queue has sufficient tokens, then all channels must.

`void cpn_iqueue<T>::Dequeue(uint count)`

The parameter *count* is the number of tokens that *Dequeue* will dequeue (and discard) from the referenced input queue. If insufficient tokens are available, *Dequeue* will block until *count* tokens are available. If the referenced queue has multiple channels, *count* tokens are discarded from each of the channels.

`T* cpn_oqueue<T>::GetEnqueuePtr(uint thresh, uint chan=0)`

The parameter *thresh* is the (write) firing threshold in tokens. *GetEnqueuePtr* blocks until free space is available for at least *thresh* tokens in the referenced output queue. Once sufficient free space is available, *GetEnqueuePtr* returns a pointer (which can be treated as a sequential array) to where the tokens should be written. If the referenced queue has multiple channels, the parameter *chan* identifies which channel the return pointer references. If any channel in a queue has sufficient free space, then all channels must.

`void cpn_iqueue<T>::Enqueue(uint count)`

The parameter *count* is the number of tokens that *Enqueue* will enqueue into the referenced output queue. If the referenced queue has multiple channels, *count* tokens are enqueued into each of the channels. Before calling *Enqueue*, the first *count* tokens of all channels should have been set by dereferencing pointers returned from *GetEnqueuePtr*. If any tokens were not set, then unknown data will be enqueued. The *GetEnqueuePtr* parameter *thresh* should be greater than or equal to *count*. If not, unknown data will be enqueued and *Enqueue* may have insufficient free space to enqueue tokens. If insufficient free space is available in the referenced queue, *Enqueue* will block until free space for *count* tokens is available.

The four class methods detailed above are the basic input and output semantics for Computational Process Networks. They provide both firing thresholds and a zero-copy interface, as well as multi-token and multi-channel transactions. Note that the output transactions are blocking, so these semantics implement bounded scheduling as described in Section 2.3. Bounded scheduling requires an online deadlock detection algorithm such as D4R, described in Section 3.3.

```
const unsigned Nfft = 1024;
const unsigned Nolap = Nfft/2;
typedef complex<float> T;
T filter[Nfft];
```

Figure 4.5: Definitions to be used in Figures 4.6 and 4.7.

4.2.4 Example: FIR Filter Using Overlap-Save FFT

The *overlap-save* method is commonly used to compute the convolution between a very long (possibly infinite) signal and an FIR filter [65]. An overlap in the input signal is required to produce the same output as a linear convolution, and overlap-save can often be implemented in fewer operations than linear convolution. In this section, I implement the same overlap-save FIR filter with two different semantics for comparison: extended KPN (based on traditional *read* and *write*) and CPN. Figure 4.5 provides definitions that will be used in both implementations, and shows that this filter example uses a 1024-point overlap-save FFT with 50% overlap and complex single-precision floating point. The variable `filter` contains the FIR filter that will be convolved with the signal carried by the queues. This filter must be in the frequency domain, and of order no greater than the number of points of overlap.

Figure 4.6 implements the filter using extended KPN semantics, as discussed in Section 4.2.1. These traditional (*read* and *write*) semantics require two temporary buffers (line 3): `ibuf` where the overlap of the input signal is managed, and `obuf` where the output results are temporarily stored. The buffer `ibuf` contains the filter history, so it must be initialized (to zero in this

```
 1 void fftfir_pn(iqueue<T> iQ, oqueue<T> oQ )
 2 {
 3   T obuf[Nfft], ibuf[Nfft] = { 0 };
 4   while (true) {
 5     // explicitly copy for overlap-save
 6     memcpy(ibuf, ibuf+Nfft-Nolap, Nolap*sizeof(T));
 7     // dequeue and copy in new tokens
 8     iQ.read(ibuf+Nolap, Nfft-Nolap);
 9
10     // perform filtering operations
11     fft(ibuf, obuf, Nfft);
12     cpx_multiply(filter, obuf, obuf, Nfft);
13     ifft(obuf, obuf, Nfft);
14
15     // copy out and enqueue results
16     oQ.write(obuf, Nfft-Nolap);
17   }
18 }
```

Figure 4.6: An FIR filter with overlap-save using extended KPN semantics.

case). The remaining steps in the filter implementation are repeated forever: copy in new data while managing the overlap, call functions that implement the filtering, and copy out the results.

To manage the overlap, the POSIX function *memcpy* is called (line 6) to copy the last `Nolap` (512) tokens from the end of the input buffer to the beginning of the input buffer. Next, the remainder of the input buffer is filled by calling *read* (line 8), which dequeues 512 tokens from the input queue and copies them to the end of the input buffer. The *read* function will block until sufficient tokens are available. These input buffer management steps require copying 1024 tokens, or 8192 bytes. Now the overlapping input stream is in a contiguous memory region so the functions that implement the filtering can be called.

Multiplication in the frequency domain is circular convolution in the time domain [65]. The function *fft* computes the FFT of `ibuf` and puts its frequency domain representation into `obuf` (line 11). Next, *cpx_multiply* computes the complex product of the frequency domain signal and filter (line 12). Finally, *ifft* computes the inverse FFT of `obuf` and puts the time domain result back into `obuf` (line 13). Functions such as these are commonly included in signal processing libraries. For example, the Fastest Fourier Transform in the West (FFTW) is a very widely used FFT library [66].

The remaining step is to enqueue the result (contained in `obuf`) to the output queue (line 16). Only a portion of the output buffer corresponds to linear convolution (all but the final `Nolap` tokens), so only the valid portion is sent to the output. With bounded scheduling, the *write* function will block until sufficient free space is available in the output queue. This output step requires copying 512 tokens, or 4096 bytes. One "firing" of the filter has completed, and the process loops back to the top to filter the next block of data.

Figure 4.7 implements this same FIR filter with the zero-copy and thresholding semantics of CPN. The call to *GetDequeuePtr* (line 5) will block until 1024 tokens are available for consumption in the input queue, and then return a pointer to those tokens in contiguous memory. The call to *GetEn-*

```
 1 void fftfir_cpn(cpn_iqueue<T> iQ, cpn_oqueue<T> oQ )
 2 {
 3   while (true) {
 4     // blocking calls to get in/out pointers
 5     const T* iPtr = iQ.GetDequeuePtr(Nfft);
 6     T*       oPtr = oQ.GetEnqueuePtr(Nfft);
 7
 8     // perform filtering operations
 9     fft(iPtr, oPtr, Nfft);
10     cpx_multiply(filter, oPtr, oPtr, Nfft);
11     ifft(oPtr, oPtr, Nfft);
12
13     // complete the queue transactions
14     iQ.Dequeue(Nfft-Nolap);
15     oQ.Enqueue(Nfft-Nolap);
16   }
17 }
```

Figure 4.7: An FIR filter with overlap-save using CPN semantics.

queuePtr (line 6) will block until free space for 1024 tokens is available in the output queue, and then return a pointer to space for those tokens in contiguous memory. Neither of these functions has copied any tokens, and no management of input overlap was required. We now have contiguous input and output buffers as required to call the filtering functions.

The functions that implement the filtering in Figure 4.7 (lines 9-11) are exactly the same as those in Figure 4.6. The only difference is that they are operating on the pointers returned from *GetDequeuePtr* and *GetEnqueuePtr* instead of temporary stack buffers. At the conclusion of the filtering calls, the results are pointed to by `oPtr`. Again, only portions of the results correspond to linear convolution and are sent as output.

The final steps are to complete the queue transactions. The call to *Dequeue* (line 14) tells the input queue to discard 512 tokens. The call to *Enqueue* (line 15) tells the output queue to insert the first 512 tokens pointed to by `oPtr`. Neither of these steps copies any tokens; they simply adjust indices in the queue. For Figure 4.7, one firing of the filter has completed, and the process loops back to the top to filter the next block of data.

The examples of Figure 4.6 and 4.7 perform the same filtering operation and compute the same results. However, the CPN semantics remove the need to manage the overlapping input buffer and also eliminate several copy operations. With the KPN approach, the filter state must be stored in a buffer (`ibuf`) that is local to the process. However with the CPN approach, the filter state is stored entirely in the queues and the state of the network can be described by the tokens in the queues. The following section examines how the CPN semantics preserve the formal properties that Kahn proved for the semantics of Process Networks.

4.3 Preservation of Formal KPN Properties

Section 2.2.2 details Kahn's mathematical representation of Process Networks. Each queue is a stream, and the contents of the stream represent the tokens that have been inserted into the queue. An increasing chain of streams represents the entire history of tokens that have been inserted into a given queue as execution progresses. This increasing chain is a complete partial order. A processes is a continuous, monotonic functional mapping between input streams and output streams. Continuous, monotonic functions of a complete partial order have a least fixed point that can be computed, and the fixed point solution of the network equations corresponds to the behavior of the executing processes [35]. As discussed in Section 2.2.3, this holds for all programs comprised of processes that use the simple blocking *get* and non-blocking *put* semantics.

Although CPN uses different semantics than Kahn's original PN to insert and remove tokens from queues, any arbitrary CPN program can be transformed into a KPN program that uses only *get* and *put* semantics. Because of this transformation, CPN has precisely the same mathematical representation as KPN and is similarly described by continuous, monotonic functions of a complete partial order. If this transformation can be shown, then the Kahn Principle applies, and all of the formal properties of KPN are preserved in CPN.

This transformation of CPN to KPN is accomplished by adding queues and modifying each process. Anywhere a queue enters or leaves a process, a self-loop queue is added to handle firing thresholds and multi-token transactions on contiguous buffers. Tokens entering or leaving a CPN process temporarily pass through one of these self-loop queues. Because CPN is defined to have blocking write semantics (via *GetEnqueuePtr* and *Enqueue*), this CPN transformation must also include feedback queues for boundedness, as proposed by Parks [36] and detailed in Section 2.3.

```
template <typename T>
T bpn_iqueue<T>::get()
{
  T result = iqueue<T>::get();
  feedbackQ.put(0); // feedback value unused
  return result;
}

template <typename T>
void bpn_oqueue<T>::put(T t)
{
  feedbackQ.get(); // feedback result discarded
  oqueue<T>::put(t);
}
```

Figure 4.8: Illustration of Parks' transformation for boundedness.

The following sub-sections illustrate the transformation of CPN as transformed to use *get* and *put*. Before addressing CPN, I revisit Parks' transformation for boundedness.

4.3.1 Parks' Transformation for Boundedness

Boundedness, including Parks' transformation, is covered in Section 2.3. Any arbitrary KPN program can be transformed to be strictly bounded by adding a feedback queue for each existing queue and modifying each process. When a process reads a token from an existing queue it also writes a token to the associated feedback queue. Before a process can write a token to an existing queue, it must read a token from the associated feedback queue.

Parks merely describes his transformation for boundedness. For clarity, Figure 4.8 illustrates Parks' transformation with code. In the style of `iqueue<T>` and `oqueue<T>` from Section 2.2, I define bounded queue classes `bpn_iqueue<T>` and `bpn_oqueue<T>` where the `bpn` prefix indicates that these are bounded KPN. Each bounded queue class contains a feedback queue as described by Parks, and named `feedbackQ`. The bounded version of *get* calls Kahn's original *get*, and then calls *put* on the feedback queue. The bounded version of *put* must *get* a token from the feedback queue before it calls Kahn's original *put*.

An implementation of bounded scheduling does not actually add feedback queues. Parks argues that bounded queues that block when becoming full (with proper artificial deadlock detection) are equivalent to the feedback queue transformation he describes. The transformation from CPN to KPN will be similarly described.

4.3.2 Transforming CPN to KPN

Figure 4.9 illustrates the transformation of a single CPN process (with input P and output Q) to KPN. The feedback queues for boundedness are P_f and Q_f, and the self-loop queues for managing firing thresholds are P_t and Q_t. Queue that are grayed carry only feedback tokens, in which the value is unimportant.

In the transformation from CPN to KPN, all tokens entering a CPN process (including feedback tokens for boundedness) pass through a self-loop queue. For incoming queues, a call to *GetDequeuePtr* ensures that the associ-

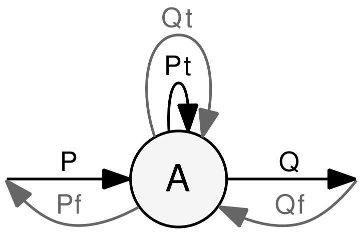


Figure 4.9: Transformation of a CPN node to KPN.

ated loop-back queue contains at least the number of tokens requested by the threshold parameter. If there are fewer tokens than requested, then additional tokens are read from the incoming queue and placed in the self-loop queue. Using this approach, a process calling *GetDequeuePtr* will block until it has the requested threshold number of tokens readily available to be accessed. A call to *Dequeue* simply discards the requested number of tokens from the self-loop queue and places the same number of (valueless) tokens in the associated feedback queue.

For outgoing queues, a call to *GetEnqueuePtr* also ensures that the associated self-loop queue contains at least the number of tokens requested by the threshold parameter. In this case, the outgoing self-loop queue contains feedback tokens for preservation of boundedness of the outgoing queue. If there are fewer feedback tokens than requested, the additional feedback tokens are read from the associated feedback queue. This way, a process calling *GetEnqueuePtr* will block until it has the requested threshold amount of free space on its output queue. When the process later calls *Enqueue*, it will do so without blocking because the requested space was made available. A call to *Enqueue* inserts the requested number of tokens into the output queue, and discards the same number of (valueless) feedback tokens from the associated self-loop queue.

This use of self-loop queues is out of a desire for memoryless processes, stated to be a desirable property in Section 1.4.3. This transformation can also be shown with persistent memory buffers associated with each queue, similar to `ibuf` and `obuf` in Figure 4.6. For clarity, the following subsections illustrate this transformation with code, similar to what was done in Figure 4.8, and use persistent memory buffers.

4.3.3 CPN Dequeue Semantics

Continuing the illustration with the template class `cpn_iqueue<T>` from Figure 4.4, I show how the CPN semantics *GetDequeuePtr* and *Dequeue* can be implemented using only *get* of new data from an input queue and *put* to the feedback queue for boundedness.

The method *GetDequeuePtr* must form a buffer of contiguous tokens of the requested threshold length, and return a pointer to that buffer. If insufficient tokens are available in the buffer, *GetDequeuePtr* should *get* more tokens from the incoming queue and add them to the buffer. The method *Dequeue* must remove the number of instructed tokens from the buffer, and *put* this same number of tokens into the boundedness feedback queue. The method *Dequeue* must also manage any overlap needed in the input buffer.

```
const T* cpn_iqueue<T>::GetDequeuePtr(uint thresh, uint)
{
  // ibuf must contain >= thresh tokens
  while (ibuf.size() < thresh)
    ibuf.push_back( iqueue<T>::get() );
  return &(ibuf[0]);
}

void cpn_iqueue<T>::Dequeue(uint count)
{
  // handle (abnormal) count>thresh
  while (count > ibuf.size()) {
    iqueue<T>::get(); // overage discarded
    feedbackQ.put(0); // feedback value unused
    count -= 1;
  }
  // manage the overlap
  uint overlap = ibuf.size() - count;
  for (int i=0; i<overlap; i++)
    ibuf[i] = ibuf[i+count];
  // discard and send feedback
  while (count) {
    ibuf.pop_back();
    feedbackQ.put(0); // feedback value unused
    count -= 1;
  }
}
```

Figure 4.10: Code transforming CPN read semantics to KPN.

This working input buffer is a private class member of `cpn_iqueue<T>` called `ibuf`. This member `ibuf` is shown to be using the *vector* template class from the C++ standard template library (STL) [67], which effectively implements a variable-length array. The member `feedbackQ` is also used as before in Figure 4.8. Figure 4.10 provides code for *GetDequeuePtr* and *Dequeue*, and these implementations use only *get* and *put* according to the stated goal. This code is to illustrate the transformation from CPN to KPN, not an actual implementation.

4.3.4 CPN Enqueue Semantics

CPN semantics *GetEnqueuePtr* and *Enqueue* can also be described as a transformation from CPN to KPN. These form the semantics for a process writing to a queue, as given by template class `cpn_oqueue<T>` in Figure 4.4, and must use only *put* of produced data and *get* from the feedback queue for boundedness.

The method *GetEnqueuePtr* must provide an empty buffer of the requested threshold length, and return a pointer to that buffer so that the client can fill it with tokens to be sent. If insufficient space is available in the buffer, *GetEnqueuePtr* should *get* additional tokens from the feedback queue until the requested space is present. The method *Enqueue* must *get* boundedness feedback tokens for any output tokens it will send, and then *put* the first tokens from the output buffer into the outgoing queue. This working output buffer is a private class member of `cpn_oqueue<T>` called `obuf`, and again uses the STL

```
T* cpn_oqueue<T>::GetEnqueuePtr(uint thresh, uint)
{
  // obuf must have space for >=thresh tokens
  while (obuf.size() < thresh) {
    feedbackQ.get(); // feedback result discarded
    obuf.push_back(0); // free space placeholder
  }
  return &(obuf[0]);
}

void cpn_oqueue<T>::Enqueue(uint count)
{
  // put tokens from head of obuf
  uint ntok = obuf.size();
  if (ntok>count) ntok = count;
  for (uint i=0; i<ntok; i++) {
    feedbackQ.get(); // feedback result discarded
    oqueue<T>::put( obuf[i] );
  }
  // remove sent tokens from obuf
  // (no need to manage overlap)
  obuf.resize( obuf.size()-ntok );
  count -= ntok;
  // handle (abnormal) count>thresh
  while (count) {
    feedbackQ.get(); // feedback result discarded
    oqueue<T>::put( 0 ); // garbage inserted!
    count -= 1;
  }
}
```

Figure 4.11: Code transforming CPN write semantics to KPN.

vector class. The member `feedbackQ` is also used as before. Figure 4.11 provides code for *GetEnqueuePtr* and *Enqueue*. Again, this code is to illustrate the transformation from CPN to KPN, not an actual implementation.

I have shown the CPN semantics operations as using buffers (`ibuf` and `obuf`) that reside as part of process memory, meaning that the processes are no longer memoryless. However this use of process memory is only for clarity, and this state information can be put onto self-loop queues for each process. With that approach, the overlap state would again be contained in the queue memory.

4.3.5 Multi-Channel Queues

A queue containing multiple channels and using Kahn's PN semantics can simply send and receive tokens consisting of tuples with one value for each channel. For example, to carry stereo audio data one could declare `queue<int[2]>`, where the two integers represent one sample each for the left and right channels. In this example, *get* would return a 2-tuple (pair) of integers, and *put* would take a pair of integers as an argument. Tuples of values are sent and received synchronously by *get* and *put* as a single token. Although not specifically mentioned by Kahn, this use of tuples for tokens follows his mathematical model and preserves the formal properties that he asserts.

Parks used feedback queues for each data-carrying queue to achieve bounded scheduling when possible [36]. These feedback queues need not carry tuples, as the values in the feedback queues are unimportant and are simply a placeholder for space in the data-carrying queue. Parks' transformation is not affected by multi-channel queues.

In Sections 4.3.3 and 4.3.4, details on multi-channel queues is omitted for clarity. For the methods *GetDequeuePtr* and *GetEnqueuePtr*, the final parameter is *channel*. A default parameter of zero gives the first channel unless otherwise specified.

For the case of multi-channels queues in the transformation from CPN to KPN, the members `ibuf` and `obuf` simply become a vector of vectors: one vector for each channel. Each channel is operated on identically with the tuple tokens being exchanged via *get* and *put*: if a sample is inserted or removed from one channel of the buffer, all channels must perform the same operation. When *get* returns a tuple of samples (one per channel), one sample is inserted into each channel of the input buffer `ibuf`. When calling *put* to send a tuple of samples, one sample is removed from each channel of the output buffer `obuf`. The methods *GetDequeuePtr* and *GetEnqueuePtr* again verify that the input or output buffer is of the requested threshold length, and return a pointer to the buffer containing the requested channel.

I assert that the Computational Process Networks can be transformed to Kahn Process Networks as shown, and that the CPN semantics as defined can be implemented using only Kahn's simple *get* and *put* semantics. Therefore, CPN indeed preserves all of the formal properties of KPN: determinism, scalability, and composability. The next section addresses the performance advantages of CPN over KPN.

4.4 Performance Improvements of CPN Over KPN

Kahn [35] points out that Process Networks are Turing complete. By transformation, CPN is also Turing complete. Although it is therefore true that KPN can eventually compute anything that CPN can compute, CPN can compute a particular important class of problems in fewer operations than KPN. Specifically, CPN efficiently models algorithms operating on overlapping continuous streams of data commonly found in digital signal processing systems, and CPN also uses zero-copy semantics.

Although transforming CPN to KPN is fairly complicated, implementing the semantics of KPN in terms of the semantics of CPN is trivial, and illustrates the additional operations implicit in the copy semantics of KPN. In Section 4.2.1 I extended Kahn's single-token *get* and *put* semantics with multi-token transactions *read* and *write*, based on the traditional POSIX functions of the same name. This is a simple extension to the Kahn's semantics, but the performance consequences can be profound because the overhead for a queue transaction is amortized over multiple tokens.

It is also illustrative to express these traditional *read* and *write* semantics using the semantics of CPN, as shown in Figure 4.12. Note that *get* and *put* are special cases of *read* and *write* where count is one. Deconstructing *read* and *write* in this manner makes obvious their implicit copy. In both functions, a pointer to the source or destination data is obtained, the copy is performed, and the queue state is updated. Clearly using zero-copy semantics can eliminate the implicit copy in *read* and *write*. In Figure 4.12, the threshold

```
int iqueue<T>::read(T* ptr, unsigned count) {
  const T* p = GetDequeuePtr(count);
  for (unsigned i=0; i<count; i++)
    ptr[i] = p[i]; // copy count tokens
  Dequeue(count);
  return count;
}

int oqueue<T>::write(const T* ptr, unsigned count) {
  T* p = GetEnqueuePtr(count);
  for (unsigned i=0; i<count; i++)
    p[i] = ptr[i]; // copy count tokens
  Enqueue(count);
  return count;
}
```

Figure 4.12: Implementing traditional read and write with CPN semantics.

(number of tokens or free space pointed to) and the count (number of tokens dequeued or enqueued) are necessarily equal. However, the semantics of CPN also support thresholds that are larger than the count.

Sliding window algorithms, which operate on continuous, overlapping streams of data, are common in DSP systems. Examples include filters and overlap-save FFTs. Firing thresholds and zero-copy semantics efficiently model sliding window algorithms, and serve to reduce copying that is typically required for data management. The performance advantage of firing thresholds is readily apparent in the overlap-save FFT example from Section 4.2.4 which compares traditional *read* and *write* semantics to the semantics of CPN.

When using DSP algorithm implementations (such as FFTs) that are written to assume operation on contiguous input or output buffers, the user must manage the overlap if a sliding window is to be used. Some DSP processors provide modulo addressing hardware to aid this problem, but this is not typically available on general purpose processors. Figure 4.6 shows the overlap-save FFT example using traditional *read* and *write* semantics. Generally, the steps for executing one iteration of a sliding window algorithm using these traditional semantics are as follows:

1. Copy overlap data from the tail to the head of the input buffer (line 6)
2. Copy new incoming data after the overlap data (implicit in *read*, line 8)
3. Perform the desired operations on the contiguous buffer (lines 11-13)
4. Copy result data to the output (implicit in *write*, line 16)

Step 1 above is necessary to manage the contiguous input buffer, but is not required with the semantics of CPN (as in Figure 4.7). The implicit copies performed by *read* and *write* in steps 2 and 4 are eliminated with the semantics of CPN, even if the algorithm does not require a sliding window. While the semantics of CPN clearly require fewer operations than the traditional approach, the magnitude of performance improvement depends on the relative execution times of these steps. For workloads where step 3 is relatively dominant, the elimination of the copies in the other steps may appear negligible. However, memory latency is frequently a bottleneck in computer systems [68], and eliminating these copies (and the associated memory access operations) can remove unnecessary overhead. Section 5.4.2 presents simulation results for the FFT example in Section 4.2.4.

The CPN semantics also offer multi-channel queues, which can help to capture data parallelism. By using multi-channel queues instead of multiple queues, the number of queue transactions is reduced, which can additionally reduce runtime overhead.

4.5 Conclusion

This chapter proposes the Computational Process Network model. The CPN model is based on the formalisms of Kahn's Process Network model, but with enhancements that are designed to make it efficiently implementable. The enhancements include multi-token transactions to reduce execution overhead, multi-channel queues for multi-dimensional synchronous data, firing thresholds for queues as both consumers and producers, and zero-copy queue communication semantics.

This chapter details the semantics of CPN: *GetDequeuePtr*, *Dequeue*, *GetEnqueuePtr*, and *Enqueue*, and compares them to the traditional *read* and *write* semantics present in POSIX. An example of an FIR filter using an overlap-save FFT is provided for both sets of semantics.

To prove that the formal properties of Kahn's PN model are preserved, I describe how any CPN program can be transformed into a KPN program by adding queues and modifying each process. A CPN program is therefore a set of continuous, monotonic functions of a complete partial order with a least fixed point that corresponds to the behavior of the executing processes [35]. I also provide code that illustrates this transformation from CPN to KPN by implementing the CPN semantics only in terms of Kahn's simple *get* and *put* operations. I argue that the semantics of CPN can reduce the number of operations performed when implementing sliding-window algorithms, as compared to the traditional *read* and *write* semantics.

Chapter 5, which follows, details my high-performance implementation framework for Computational Process Networks, and targeting parallel and distributed POSIX systems.

Chapter 5

Implementation of CPN Framework

The Computational Process Network model, detailed in Chapter 4, is designed to be efficiently implementable while preserving the formal properties of Kahn's PN model. Additionally, the CPN model reduces the operations required to implement algorithms operating on common signal processing algorithms. In this chapter, I present details about the CPN framework implementation, which is targets high-throughput computationally intensive algorithms being implemented on workstations and workstation clusters.

Section 5.1 provides an introduction. Section 5.2 describes the CPN kernel. Section 5.3 describes the implementation of nodes in CPN. Section 5.4 describes the implementation of queues in CPN. Section 5.5 describes the distribution of CPN programs across multiple computers. Section 5.6 describes building CPN programs from a coordination language. Section 5.7 discusses the use of CPN as an embedded library. Section 5.8 concludes this chapter.

5.1 Introduction

The CPN framework is a high-performance implementation of the CPN model targeting POSIX (Unix) systems. This scalable software framework is written in C++ and uses a layered inheritance approach to build interfaces and functionality. It can exploit multi-core parallelism because it is built on POSIX threads, and it can also exploit distributed parallelism via a network. The queues implement the CPN semantics (Section 4.2), which include firing thresholds and a zero-copy interface. The queues use C++ templates so that they can carry various different data types. The queue implementation uses a novel technique with the virtual memory manager (VMM) to achieve apparent circularity and reduce overhead for high-throughput systems.

5.1.1 Development History and Public Release

Portions of this framework have been implemented and in use for many years. Early versions targeted only large SMP servers [69], and required writing C++ code to build a CPN system by manually instantiating nodes and connecting them with queues. However, recent versions can create a distributed parallel system from descriptions in a simple coordination language and dynamically loadable libraries of nodes. The CPN Kernel helps nodes to coordinate with each other, so that nodes connect and communicate using the same interface regardless of whether their peer is local or remote. Recent versions also contain D4R (Section 3.3), to yield complete, bounded execution. Using the CPN framework, developers can build high-performance, high-throughput, distributed systems from deterministic, composable components.

The CPN framework is released as open source software under the GNU Library General Public License (LGPL) [70]. As of the last release, it contains about 26000 lines of source code in about 330 source code files (as estimated by the line counting tool `cloc` [71]). It contains extensive unit tests, aiming for robustness and stability. It fully executes on both Linux and MacOS X, and would probably work on other POSIX systems with minimal effort. The CPN framework is the product of several man-years worth of effort, supported by the Independent Research and Development program at Applied Research Laboratories: The University of Texas at Austin.

5.1.2 Describing a CPN System

As a practicing electrical engineer with experience in circuit design, I naturally draw parallels between Process Networks and electrical circuits. A circuit consists of some number of parts that are connected by some number of wires, or *nets*. A circuit may have many different types of parts, including multiple instances of the same type of part. A net is described as going from a pin (or port) of one part to a pin of another part. The list of all the parts (including part type) in a circuit is called the *part list*, and a list of all the nets is called a *netlist*. We commonly draw schematic diagrams of circuits, but the salient information contained in a schematic can generally be boiled down to a part list and a netlist.

One can also describe a Process Network with a part list (of node designators and types) and a netlist (of connections between nodes). Kahn's original example from Section 2.2.1 is reproduced here in Figure 5.1, and can be described by the part list in Table 5.1 and the netlist in Table 5.2. There

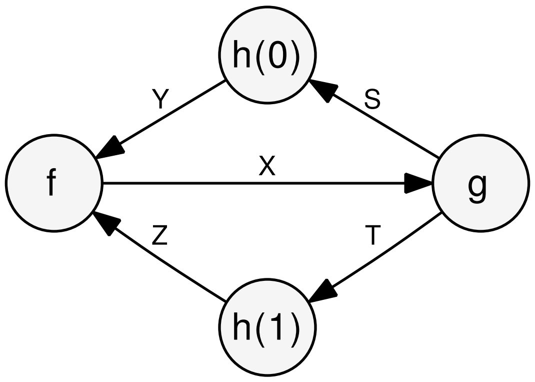


Figure 5.1: Kahn's example of a simple Process Network program.

are two instantiations of the node type h, each taking a different parameter. The are no parameters in this part list (parameters will later be shown to be attributes of an instance), but designator `h0` represents the node $h(0)$.

It is easy to describe an arbitrarily complex CPN system with these two simple lists. The level of granularity in the description is an important consideration. Like CPN, many signal processing algorithms are modeled using directed graphs. Often, each node represents very fine-grain computations such as addition and multiplication [65]. Very fine granularity is inappropriate for

Table 5.1: Part list for the example in Figure 5.1

designator	node type
f	Kahn_f
g	Kahn_g
h0	Kahn_h
h1	Kahn_h

Table 5.2: Netlist for the example in Figure 5.1

	from		to	
queue name	node	port	node	port
X	f	out	g	in
S	g	out0	h0	in
Y	h0	out	f	in0
T	g	out1	h1	in
Z	h1	out	f	in1

this implementation, because the overhead of the dynamic scheduler in the OS will dominate the overall execution time. This framework is intended for use with nodes of larger granularity, such as an FFT node, a filter node, or a beamformer node. However, if node granularity is too large, scalability is limited.

CPN graphs can be thought of as system block diagrams, and a part list and netlist can describe a large software system that can be built from composable components. The portion of the CPN framework that maintains these lists and builds the network is the CPN Kernel.

5.2 The CPN Kernel

In programs that use the CPN framework, the CPN Kernel serves as the coordinator for building the network. Every node in the Process Network program is spawned by a kernel (as a POSIX thread), and any modifications to the system graph are done with the help of a kernel. The kernel provides interfaces so that nodes can connect to each other without knowing higher-level details about the system, therefore keeping the nodes modular and self-contained. Nodes and queues can be dynamically created in the CPN model, so the kernel also provides these interfaces. Runtime parameters to CPN nodes are also provided by an interface from the kernel.

The intention is that one CPN Kernel is running on a shared-memory compute host, within a single (Unix) process on that host. Because this one process contains a number of threads (one per CPN node), it can exploit parallel hardware. If there are multiple kernels (as in the case of cluster computing), the kernels communicate and coordinate with each other, so that the CPN nodes in the system are unaware that they may be distributed.

A `CPN::KernelAttr` instantiation is created so that various global parameters and options can be set. A CPN Kernel is created when the class `CPN::Kernel` is instantiated, taking an attribute object as an argument. The newly created CPN Kernel spawns a new thread in which its duties will be performed, and then returns control to the calling thread. This way, the CPN framework does not take complete control of a (Unix) process, and can be used as a library that is embedded within other applications.

Because there is a single CPN Kernel coordinating a collection of CPN nodes, one could be concerned that the kernel is a bottleneck for a parallel system. Indeed, the kernel contains locking and critical sections for nodes that interact with it. However, the only required kernel interaction for a node is when a node is trying to establish its queue connections at initialization time, or if a node is requesting the creation of new nodes or queues (and therefore modifying the system graph). Nodes that start up and execute without requesting graph modifications are autonomous and need no further interaction with the kernel.

More details about the CPN Kernel will be discussed in upcoming sections, in the context of those other components of the CPN framework.

5.3 CPN Nodes

Each node in a CPN program corresponds to a POSIX thread, or Pthread. Multiple threads can run concurrently when there is parallelism, and thus can take advantage of multiple processors on an SMP workstation. Pthreads are intended to provide high performance with low overhead, and can optionally be given fixed-priority real-time scheduling priority.

By realizing the CPN framework with POSIX Pthreads, it can be run on many different Unix platforms. Because of the hierarchical design of the framework, it could be possible to port it to additional thread implementations by changing some key base classes. There is also a project that claims to provide a high-quality implementation of pthreads on Microsoft Windows [72].

In Section 1.3.1 I discussed the problems with threads and echoed Lee's assertion [3] that threads are "wildly nondeterministic." However, the complication of thread programming is not exposed to users of the CPN framework. Users need not be concerned with managing critical sections and locking shared resources, only following the implementation of the formal model using the CPN semantics. Nodes can interact with one another only via queues, so queues are the only shared resources that need to be locked. This locking occurs transparently inside the CPN framework without any effort by the user. This limited scope of resource sharing (at queues) also helps with verification of the CPN framework implementation.

Because the formal model provides determinacy for any execution order, any thread scheduler will yield a correct answer. The nodes in the system will execute as the flow of data permits. Generally the standard system scheduler is used, although the real-time scheduler can optionally be used.

An overly fine level of node granularity will yield significant overhead due to dynamic scheduling. The cost of executing a node should be much larger than the cost of a thread context switch (on the order of microseconds). This framework is intended for use with nodes of larger granularity. However, if the computation of a node is too costly, then the node may need to be divided into smaller pieces to increase scalability or to achieve real-time performance. Generally, a trade-off exists between overhead, latency, and parallelism.

5.3.1 Creating a New Type of Node

In the CPN framework, a new type of node is created by subclassing `CPN::NodeBase`. The user must override the method `NodeBase::Process()`, which is pure virtual. When a node is instantiated, it is this member function that executes in a new separate thread.

To communicate with the outside world, a newly created node queries the CPN Kernel for its input or output queues, and any parameters it may need. The kernel maintains a netlist of queue connections, so it can resolve and return the queue to which the node should communicate. The `NodeBase` class provides methods for performing these queries. For example, for the node to request a means to communicate with an input queue, it will call the method `GetIQueue(portName)`. The parameter is a string that names the particular port being requested. The node will call `GetOQueue(portName)` to get an output queue. These calls will block until the kernel returns an object that the node can use to communicate with the queue that is connected to the specified port. The returned object provides the CPN queue interfaces described in Section 4.2.3 and below in Section 5.4 on CPN Queues. For brevity and clarity where the zero-copy semantics may be overkill (such as when sending only a single token), the CPN framework also provides convenience queue access methods in the style of `Enqueue(pointer,length)`, and similar to the traditional *read* and *write* semantics.

Similarly, the node can fetch parameters that have been set for it with `GetParam(key)`, and check whether a parameter exists with `HasParam(key)`. In both cases, the key is a string that names the parameter. If a parameter needs to be coerced to a particular type, the CPN framework also provides the method `GetParam<type>(key)`.

Figure 5.2 revisits the implementation of Kahn's simple example from Section 2.2.1 but implemented in CPN framework. The first line is a macro

```
CPN_DECLARE_NODE_AND_FACTORY(Kahn_h, Kahn_h);

void Kahn_h::Process()
{
  IQueue<int> in = GetIQueue("in");
  OQueue<int> out = GetOQueue("out");
  int value = GetParam<int>("first");

  out.Enqueue(&value,1);
  while (true) {
    in.Dequeue(&value,1);
    out.Enqueue(&value,1);
  }
}
```

Figure 5.2: Implementation of Kahn's example process `h` with subclassing.

that declares a class named `Kahn_h` as a member-less subclass of `NodeBase`. It also declares a factory class for creating nodes of type `Kahn_h`, as well as a function that returns a node factory for `Kahn_h` nodes (named according to a convention). These products help the CPN Kernel to dynamically look up factories and create nodes of a particular type.

For most cases, all that remains is to define the code that implements the node in the `Process` method. The first few lines of `Kahn_h::Process` are the node requesting initialization information from the kernel. Note that there is a type assignment for the queue, and that a runtime exception will be thrown if there is a mismatch. The remaining lines of `Kahn_h::Process` are effectively the same as the simple example from Figure 2.4.

```
Kernel kernel( KernelAttr("kernel") );

NodeAttr nattr("h0", "Kahn_h"); // name h0, type Kahn_h
nattr.SetParam("first", 0); // set a parameter

kernel.CreateNode(nattr); // request creation
```

Figure 5.3: Instantiation of a node executing `Kahn_h`.

5.3.2 Instantiating a Node

Once a node type is defined, an instantiation can be created with the help of a CPN Kernel. First, an instantiation of the `CPN::NodeAttr` class is created so that a unique node name (designator) and node type are set, and any parameters are set. This attribute class is passed to the kernel's method `CreateNode`, and the kernel spawns a thread that executes the specified node. Figure 5.3 provides code that creates a kernel, and also a `NodeAttr` that calls out a `Kahn_h` node named `h0`. It then requests that the kernel spawn a node of the specified type.

5.3.3 Function Nodes

The CPN framework also provides *function* nodes, a way to create CPN nodes from functions and without subclassing. This method needs no macros for "boilerplate" code (repeated in many places with little or no alteration), may be slightly more straightforward for the developer, and has a syntax somewhat more like the examples in Section 2.2.1. However, nodes created this way cannot be looked up by name, do not have node factories, and cannot

```
void Kahn_h_function(NodeBase *node, int first)
{
  IQueue<int> in = node->GetIQueue("in");
  OQueue<int> out = node->GetOQueue("out");
  int value = first;

  out.Enqueue(&value,1);
  while (true) {
    in.Dequeue(&value,1);
    out.Enqueue(&value,1);
  }
}

int first = 1;
kernel.CreateFunctionNode("h1", Kahn_h_function, first);
```

Figure 5.4: Implementation of Kahn's example process `h` with a function.

be automatically loaded by the CPN Kernel's node loader. This means that loading nodes from a coordinating language (as discussed in Section 5.6) is not possible for function nodes.

The contents of Figure 5.4 is extremely similar to Figure 5.2. The differences are that the node parameter *first* is passed as a function parameter, and that a different kernel method, `CreateFunctionNode(...)`, is used to instantiate the node.

5.4 CPN Queues

The concept of requiring more data to be present than will be consumed upon execution is referred to as a *firing threshold*, and was introduced by Computation Graphs and discussed in Section 2.4. CPN also provides firing thresholds for producers, where a node can require more free space on its output than it may fill upon execution. The CPN semantics are defined in Section 4.2.3 to use `GetDequeuePtr` and `Dequeue` for inputs and `GetEnqueuePtr` and `Enqueue` for outputs.

This interface allows nodes to operate directly on queue memory, and data presented to nodes is already in a contiguous buffer. This reduces overhead by eliminating the need for nodes to copy and rearrange data, and simplifies the implementation of algorithms that interface to these queues. This interface is also intended to make up for the lack of circular address buffers in general purpose processors. By eliminating unnecessary data copying in high-throughput systems, the process is free for additional computation tasks. The component of the CPN framework that makes this interface possible is the `ThresholdQueue`.

5.4.1 Threshold Queues

The `ThresholdQueue` implements its apparent circular addressing by mirroring the beginning of the queue's data region (up to a maximum threshold) just past the end of the queue's data region. Using this methodology, the queue can provide a pointer to a contiguous block of data elements even when operating near the end of the data region. The queue manages this mirroring, and guarantees that the same data resides in both locations. Figure 5.5 illustrates the `ThresholdQueue` mirroring implementation.

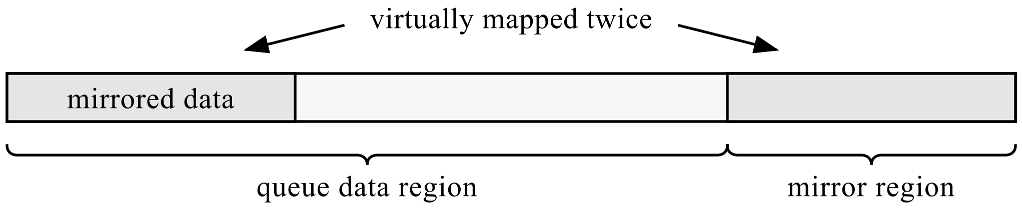


Figure 5.5: Creating the appearance of a circular queue with virtual memory.

With no hardware assistance, the `ThresholdQueue` has a trade-off between memory usage and overhead. Data copying need only occur when the indices are operating near the edges of the queue. When the data region is much larger than the mirror region, the queue rarely needs to copy data. When the mirror region is as large as the data region, copying may occur frequently, thereby increasing overhead and sacrificing performance. Using a larger data region will reduce the need to copy.

However, the virtual memory manager can be used to prevent the `ThresholdQueue` from having to copy data when managing the mirror region. The system call `mmap` is used to map virtual memory objects into the address space of a process. By mapping a shared memory object to multiple virtual addresses, the same physical memory pages appear at multiple locations, and apparent circular addressing is achieved. As a side effect, the queue data and mirror regions must both be multiples of the system memory page size, which is typically a power of two on the order of 4096 bytes. The queue size and threshold size are therefore rounded up to the next multiple of the page size when using the VMM. This mapping is maintained by the processor's virtual-

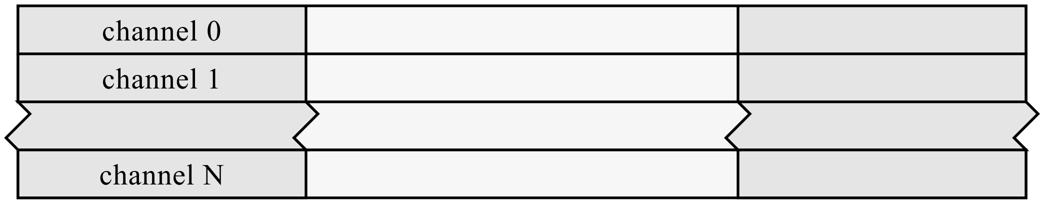


Figure 5.6: A multi-channel threshold queue.

to-physical address translation hardware, and the user process has contiguous access anywhere in the queue up to the length of the mirrored data. This can result in a significant performance gain, as shown in Section 5.4.2.

The same virtual memory approach is used for a multi-channel queue, except that there must be one circular mapping for each of the multiple channels. In this case, the same shared memory object is mapped a number of times as shown in Figure 5.6. The `ThresholdQueue` provides a foundation of the CPN framework for zero-copy interfaces and a reduction in overhead for high-throughput systems.

5.4.2 Threshold Queue Performance

I examined the performance of queues using this virtual memory mapping technique in [73]. This paper compares the performance of two different implementations of a frequency domain FIR filter using an overlap-save FFT: one with and one without a `ThresholdQueue`. These two implementations are very much like the two in Figures 4.6 and 4.7 from Section 4.2.4, except that they are not being used within a Process Network.

The basic operation that is repeatedly executed is this: compute a forward FFT, multiply the complex results by complex filter coefficients, and then perform an inverse FFT. We use the well-known "Fastest Fourier Transform in the West" (FFTW) library [66] for our forward and inverse FFTs, and a vector complex multiplication function that is hand optimized to use single instruction multiple data (SIMD) extensions for signal processing. The implementation that does not use the zero-copy queue must manually manage the overlap by copying data at each step. Before each forward FFT, overlap data is copied from the tail of the buffer to the head, and the new data is copied in. After each inverse FFT, the "good" (non-aliased) result data is copied to an output buffer. The virtual memory technique eliminates this need for management of the overlap, and increases the performance of the filter operation.

By varying algorithm parameters such as filter length and FFT length over a series of benchmarks, we are able to make some general performance evaluations about this implementation of zero-copy queues. We vary the FFT size over powers-of-two from 16 to 65536, and we vary the filter length as a percentage of the FFT length. The filter length is the same as the overlap length, so this test is also varying the percentage of overlap. We are interested in varying the parameters to explore the zero-copy queue performance over a wide workload.

We benchmark with a data set that is large enough to prevent it from fitting in cache, and run multiple trials to measure execution time.

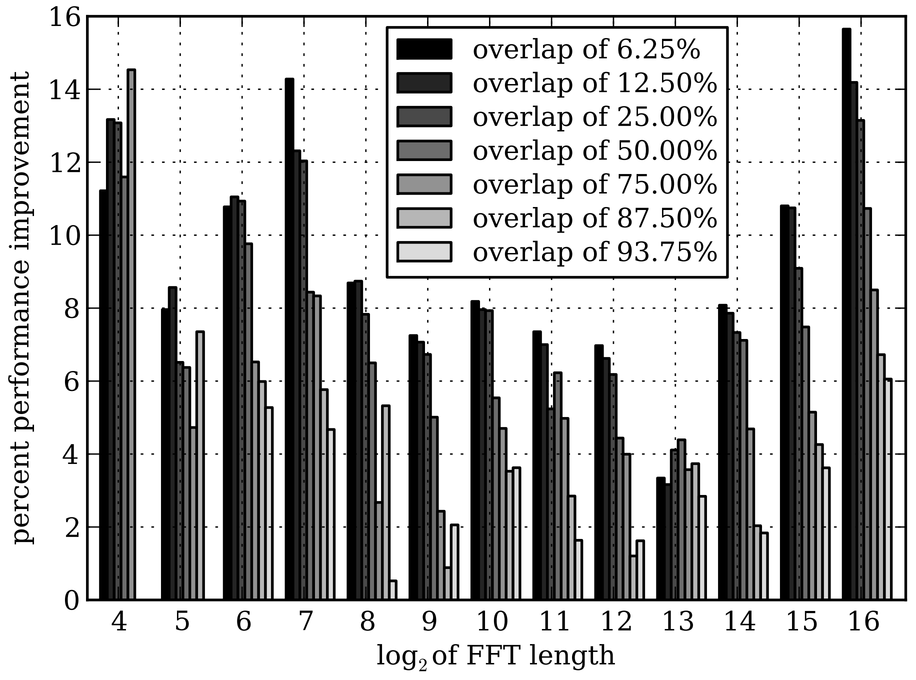


Figure 5.7: Performance improvement for zero-copy queues on the Opteron.

From the measured execution time, we compute the performance in floating-point operations per second (FLOPS) using the same method as the FFTW project [66]. For a forward FFT, a reverse FFT, and a complex multiply we use $10N \log_2 N + 6N$ FLOPS.

In 2006, we executed these benchmarks on two different hardware and software platforms. One was a 2.5 GHz PowerPC 970 (Apple Power Mac G5) running MacOS X 10.4.6, and the other a 1.8 GHz AMD Opteron in a Shuttle SN21G5 running 64-bit Red Hat Enterprise Linux 4. We have repeated these same benchmarks on two more current systems: one with 2.33 GHz Intel Xeon

“Woodcrest” model 5148 processors, and another with 2.4 GHz Intel Xeon “Nehalem” model L5530 processors. Both of these systems were running 64-bit Red Hat Enterprise Linux 5. These benchmarks use single-precision floating point with SIMD instruction sets. They also use only a single thread, and therefore only a single processor core.

In all cases, the trends and performance improvement for using zero-copy queues are similar. On the Opteron, the benchmark operates at just under 3 GFLOPS, which is about 20 percent of the theoretical peak and about 1.7 floating-point operations per cycle. Figure 5.7 (adapted from Figure 6 in [73]) presents the performance improvement of zero-copy queues over processor-executed copies, as measured in the Opteron system benchmark.

The performance improvement is a function of the time spent executing an algorithm and the time spent performing the overhead of copying data for managing the overlap. The frequency domain FIR filter is a moderately computationally intensive algorithm at $O(N \log_2 N)$, and the improvement is certainly a measurable at 5-10%. This margin would be greater for less computationally intensive algorithms or when the overhead due to copying is higher.

For high-throughput signal and image processing systems, zero-copy queues can eliminate the copies necessary for overlap management, and can make a measurable difference in performance.

```
QueueAttr qattrX(2*sizeof(int), sizeof(int));
qattrX.SetDatatype<int>().SetName("X");
qattrX.SetWriter("f", "out").SetReader("g", "in");
kernel.CreateQueue( qattrX );
```

Figure 5.8: Instantiation of the queue named X from Table 5.2.

5.4.3 Instantiating Queues

Queues in the CPN framework are built upon a `ThresholdQueue` and take advantage of the virtual memory system. Like a CPN node, a CPN queue is created with an attribute object that is passed to the CPN Kernel. An instantiation of the CPN::QueueAttr class is created, so that queue parameters can be set. Parameters include the queue length and maximum threshold size (corresponding to the queue data region and mirror region in Figure 5.5). These are the starting values for the queue size parameters, but they may grow as the program executes as instructed by the D4R algorithm described in Section 3.3. Other parameters to set include the data type that the queue will carry, the name of the queue, and the name and port for the nodes at each end of the queue. Once the parameters are set within the attribute object, it is passed to the kernel's method `CreateQueue`.

Figure 5.8 provides code that creates the queue named `X` from Kahn's simple example and the netlist in Table 5.2. A QueueAttr is created, parameters are set, and the queue is created when the attribute object is passed to the CPN Kernel.

5.4.4 Additional Queue Interfaces

CPN Queues provide interfaces beyond the CPN semantics defined in Section 4.2.3. In addition to `GetDequeuePtr(threshold,channel)` and `Dequeue(count)`, the `IQueue` also provides `Dequeue(pointer,count)` much like the traditional *read*. In addition to `GetEnqueuePtr(threshold,channel)` and `Enqueue(count)`, the `OQueue` also provides `Enqueue(pointer,count)` much like the traditional *write*. These interfaces perform implicit copies, but may not affect performance for reduced data rates.

For multi-channel queues, the stride between channels is fixed as shown in Figure 5.6. The CPN Queues provide a `ChannelStride` method so that pointer arithmetic can be performed instead of repeated function calls to get pointers to different channels, thus reducing calling overhead. The queues also provide a method to fetch the queue length and maximum threshold length, both of which were specified in the `QueueAttr` (but may have since grown).

Queues also contain a `Release` method, which informs the other side of the queue that there will be no further communication. If a CPN Node exits, then all of its queues are automatically released. Calls to `GetEnqueuePtr` on a released output queue will yield an exception, causing that node to exit. Calls to `GetDequeuePtr` on a released input queue return a NULL pointer. In this condition, nodes can consume any remaining tokens, perform any required shutdown, and pass the condition to its outputs by calling `Release`. This way, a CPN program can be brought down in a cascade of nodes through the system.

```
void DeterminateMerge<T>::Process()
{
  IQueue<T> in0 = GetIQueue("in0");
  IQueue<T> in1 = GetIQueue("in1");
  OQueue<T> out = GetOQueue("out");
  while (true) {
    // block until we would merge at least one set
    const T* in0p = in0.GetDequeuePtr(1);
    const T* in1p = in1.GetDequeuePtr(1);
    T* outp = out.GetEnqueuePtr(2);
    // compute how many total can be merged
    unsigned in0cnt = in0.Count();
    unsigned in1cnt = in1.Count();
    unsigned outcnt = out.Freespace()/2;
    unsigned N = min(in0cnt,in1cnt,outcnt);
    if (!N) break;
    // merge them all
    for (int i=0; i<N; i++) {
      outp[2*i] = in0p[i];
      outp[2*i+1] = in1p[i];
    }
    in0.Dequeue(N);
    in1.Dequeue(N);
    out.Enqueue(2*N);
  }
}
```

Figure 5.9: Low-overhead determinate merge with `Count()`.

Something that could be a point of contention is that the `IQueue` provides the methods `Empty()` and `Count()`, and the `OQueue` provides the methods `Full()` and `Freespace()`. These methods could allow a node to determine whether it will block before attempting to access a queue, which could lead to non-deterministic behavior. Some claim [31] that augmenting the PN model with explicit nondeterminism is commonly desirable for embedded software applications. Expected non-determinism, like a non-determinate merge, can be useful. However, my primary motivation for including these interfaces is performance for a common use case.

To illustrate, I use the example of determinately merging two streams, as shown in Figure 5.9. The `DeterminateMerge` node first obtains its two input queues and one output queue. The node then repeatedly attempts to merge some number of tokens. First, the node blocks on each of its 3 queues, ensuring that it could immediately merge at least one set of tokens onto the output. However, rather than proceeding to merge only 2 tokens, it computes the maximum number of tokens that it could possibly merge and does all that it can. This computation requires knowing the count of tokens in both inputs, and the amount of free space for tokens in the output, but this process is still completely determinate (with respect to how it merges the two streams into one).

This process has the potential for significantly lower overhead than a version that merges only one set of tokens at a time, while still producing the same result. If the tokens arrive one at a time, the process behaves correctly with low latency (but high overhead). However, if the tokens arrive in bursts or if the process is starved for compute time, the process will make the most of its time slice. The approach of performing as much work as possible when given the opportunity to execute should lead to an overall reduction in system overhead, and should be the preferred approach for implementing process nodes.

5.4.5 D4R

Section 3.3 discusses the distributed dynamic deadlock detection and resolution (D4R) algorithm. Because bounded execution of process networks (with blocking writes) can lead to artificial deadlock, an online deadlock resolution algorithm is required to prevent incomplete execution. Each CPN Node contains the D4R state variables, and all communication for D4R occurs over existing CPN Queues that are connected between nodes. The only time a D4R state transaction can occur is when a node is accessing one of its queues. All D4R communication can therefore occur inside queue transaction methods (such as `GetEnqueuePtr`), and deadlock detection is a feature of the network; support for D4R requires no coding on the part of CPN Nodes, and occurs invisibly during execution. D4R does incur some overhead due to additional locking and message traffic. Consequently, there is a CPN Kernel parameter that can disable the use of CPN at runtime.

5.5 Distribution on Multiple Hosts

The CPN framework is capable of distributing nodes across multiple compute hosts. Additional parameters that can be set (with a `KernelAttr`) when creating a CPN Kernel include the kernel name, the host name on which the kernel is executing, and a `CPN::Context`. The CPN Context class stores and maintains the state of the global CPN graph (the list of nodes and queues). In the single-host case, the single kernel invisibly creates an internal local context.

5.5.1 Remote Context

For multiple CPN Kernels to coordinate with each other to build a distributed system, they all communicate with a remote context daemon. This daemon must be executing somewhere in the system, and listens for connections from CPN Kernels. Kernel attribute objects contain a `RemoteContext`, which has a host name and port number where the context daemon can be contacted. Each kernel must have a unique name so that it can be uniquely identified. A context not only maintains the graph state, it also notifies kernels when they must take some action. Examples of such actions are creating a new node or queue.

An additional attribute that we could add for each node in our part list from Section 5.1 is the specification on which kernel (or host) the node will execute. When a CPN Node is created with a `NodeAttr` object, one of the methods that can be called is `SetKernel`, telling on which kernel the node should execute. When unspecified, the node will execute on the local kernel. However, with the proper attribute a node can be instructed to execute on any of the hosts (via a kernel) that is a member of the context.

Again, one could be concerned that having a single process (the remote context daemon) could be a bottleneck for a distributed system, and will limit scalability. While this is true in some sense, the only interaction with the remote context is when nodes and queues are being constructed and the CPN graph is being modified. This may typically happen at initialization time, but once nodes are instantiated and connected they interact directly with one another without assistance from either a kernel or the context.

This single remote context daemon process is also a design choice for simplicity of implementation. At some level a CPN Context is similar to a database, and distributed database implementations are a challenging problem in their own right. The CPN framework is implemented with inheritance hierarchy, and it would be possible to substitute a distributed implementation of a context if one were to be created. One advantage of a single remote context is that it is trivial to get a current snapshot of the CPN graph for examination.

5.5.2 Remote Queues

CPN Queues as described in Section 5.4 are shared memory objects, which clearly do not work on a distributed system. The CPN framework also provides the class `RemoteQueue`, so that nodes on different hosts can communicate in the same manner. The class `RemoteQueue` is a specialization of a `ThresholdQueue` that is split into two parts across a network socket. Tokens that are enqueued at the producer end of a remote queue are sent over the network. At the consumer end, received tokens await consumption by the receiving node in a `ThresholdQueue`.

For two nodes that are communicating, the fact that a peer may be remote is transparent. As in the non-distributed case, nodes request that the kernel provide a communication endpoint (via `GetIQueue` or `GetOQueue`) and then use the provided queue. Because the kernel (with the help of the context) knows where each node is executing and how the nodes and queues in the graph are to be connected, the kernel can create the network endpoints and provide them to each node. There is one additional parameter for remote queues, dubbed *alpha*. For each queue in a CPN system, the overall queue length is specified. The parameter alpha varies from zero to one, and specifies the split of queue length allocated between the reader and writer sides of the queue (where zero is all space on the reader side).

Current implementations of the `RemoteQueue` use the transmission control protocol (TCP) of the internet protocol (IP) suite, or TCP/IP, which will work on nearly any network interface. So far, remote queues have been implemented over both Ethernet and InfiniBand, a high-speed low latency switched fabric communication link that is used in high-performance computing[74]. It would also be trivial to create queues that operate over other types of reliable data stream.

5.6 CPN from a Coordination Language

While it is certainly possible to build a distributed CPN system by writing one C++ program for each host, this approach would be error prone and time-consuming to develop and deploy. Lee argues [3] that concurrent systems should be built with coordination languages based on sound, composable formalisms. I propose that a reasonable description language to build concurrent systems is a part list and a net list. These describe a composable system that coordinates with the formal CPN semantics. For distributed systems, I use one additional list: a node map which maps each node onto a particular network host.

JSON (JavaScript Object Notation) is a lightweight text-based data-interchange format that is both human and machine readable [75]. It is an open standard and in fairly wide use. The CPN framework provides a parser and loader that can read a simple JSON description of a CPN program to be built and executed. Figure 5.10 provides JSON that implements Kahn's simple example as described in Section 5.1. By adding a node map, one can specify how the (unmodified) program is to be distributed and executed across multiple compute hosts.

The CPN framework includes a parser for a similar dialect built on the widely used Extensible Markup Language (XML) [76]. A description of the CPN coordination language for both JSON and XML is available in the CPN tutorial, which is included in the CPN source code release [77].

```
{ "nodes": [
  { "name": "f", "type": "Kahn_f" },
  { "name": "g", "type": "Kahn_g" },
  { "name": "h0", "type": "Kahn_h",
    "param": { "first": 0 } },
  { "name": "h1", "type": "Kahn_h",
    "param": { "first": 1 } },
],
"queues": [
  { "name": "X", "datatype": "int32_t",
    "writernode": "f", "writerport": "out",
    "readernode": "g", "readerport": "in" }
  },
  { "name": "S", "datatype": "int32_t",
    "writernode": "g", "writerport": "out0",
    "readernode": "h0", "readerport": "in" }
  },
  { "name": "Y", "datatype": "int32_t",
    "writernode": "h0", "writerport": "out",
    "readernode": "f", "readerport": "in0" }
  },
  { "name": "T", "datatype": "int32_t",
    "writernode": "g", "writerport": "out1",
    "readernode": "h1", "readerport": "in" }
  },
  { "name": "Z", "datatype": "int32_t",
    "writernode": "h1", "writerport": "out",
    "readernode": "f", "readerport": "in1" }
  }
]}
```

Figure 5.10: Implementation of Kahn's simple example with JSON.

```
{ "nodemap": [
  { "f": "kernel0" },
  { "g": "kernel1" },
  { "h0": "kernel0" },
  { "h1": "kernel1" },
]}
```

Figure 5.11: A mapping of Figure 5.10 onto two hosts with JSON.

With a CPN Remote Context daemon and multiple hosts running (identical) programs containing CPN Kernels, a parser program can push a system description out to be loaded and executed on a workstation cluster. The CPN framework has the ability to locate, load, and instantiate node classes that are compiled into dynamically loadable shared libraries. This way code that implements all of the different types of nodes simply needs to be available on a shared network volume.

The power of building composable, scalable distributed systems from a simple text description is not in programmers editing text files, but in leveraging electronic design automation (EDA) tools. Designers could draw block diagrams of a system using schematic capture tools of node libraries, and any additional nodes that are needed can be easily developed. Part lists and netlists could be automatically extracted and turned into a scalable, distributed system. Designers need not be experts in concurrent and distributed systems to build and use them. Exploring the space of mapping the system across multiple compute hosts could be done automatically by manipulating the machine-readable mapping file, or manually specified by the designer by grouping nodes.

Highly specialized or optimized signal processing libraries that have been developed by domain experts can easily be used within the framework because communication and computation have been separated. It is simple to wrap an algorithm into a CPN Node (much like what was done with FFTW) and use it in a distributed, concurrent software system.

5.7 CPN as an Embedded Library

As mentioned in Section 5.2, a CPN Kernel spawns a new thread in which its duties will be performed, and then returns control to the calling thread. This way, the CPN framework does not take complete control of a (Unix) process, and can be used as a library that is embedded within other applications. The CPN Kernel therefore provides a few methods to allow the calling thread to examine the state of the running CPN program:

`Terminate()` instructs the kernel to terminate

`Wait()` waits for the kernel to terminate (after `Terminate` is called)

`WaitForNodeStart(nodename)` waits for a named node to start

`WaitForNode(nodename)` waits for a named node to terminate

`WaitForAllNodes()` waits for all nodes to terminate

If a main program instantiates a kernel and then creates nodes and queues, the main program should typically wait for the nodes to terminate before exiting.

```
Kernel kernel( KernelAttr("kernel") );
// create a mythical counter node
kernel.CreateNode(NodeAttr("counter", "counter"));
// create an external reader
kernel.CreateExternalReader("result");

// attach a queue between counter and external reader
QueueAttr qattr(2*sizeof(int), sizeof(int));
qattr.SetWriter("counter", "out");
qattr.SetExternalReader("result");
kernel.CreateQueue( qattr );

// now read from the external reader
IQueue<int> result = kernel.GetExternalIQueue("result");
int value;
while( result.Dequeue(&value,1) ) {
  printf("%d\n", value);
}
kernel.DestroyExternalEndpoint("result");
```

Figure 5.12: Steps to create and use an external reader.

It the main program exits without waiting, the kernel will go out of scope and be terminated by its destructor.

The CPN Kernel also provides methods for attaching to a CPN queue from outside a CPN program. This could be useful for injecting data into a CPN program, or for reading results out of a CPN program. This is achieved by using external readers and external writers, which are accessed via the following CPN Kernel methods:

`CreateExternalReader(name)` create a named external reader

`CreateExternalWriter(name)`	create a named external writer
`GetExternalIQueue(name)`	returns an IQueue ready to be accessed
`GetExternalOQueue(name)`	returns an OQueue ready to be accessed
`DestroyExternalEndpoint(name)`	destroy a named reader or writer

When initializing a `QueueAttr`, the consumer end can be specified as an external reader by using `SetExternalReader(name)` instead of `SetReader` as in Section 5.4.3. For an external writer, `SetExternalWriter(name)` is used instead of `SetWriter`.

Figure 5.12 provides an example of using an external reader. Before creating a queue, the external reader must be created. The queue is specified to use the external reader as its consumer end. The `IQueue` is then requested from the kernel, and can be read from as if it were part of the CPN program.

5.8 Conclusion

This chapter presents details about the CPN framework implementation, which is intended for high-throughput computationally intensive algorithms on symmetric multiprocessing workstations and workstation clusters. This scalable software framework uses POSIX threads and C++ template data types, and uses a layered approach based on the C++ inheritance mechanism to build interfaces and functionality. Using a simple coordination language, developers can use this framework to build high-performance, distributed systems from deterministic, composable components.

Chapter 6 presents case studies that demonstrate the capabilities of the CPN framework.

Chapter 6

CPN Case Studies

In Chapter 4, I describe the enhancements I have made to Kahn's Process Network model to make an efficiently implementable model, Computational Process Networks, which preserves Kahn's formal properties of determinacy and scalability. Chapter 5 presents details on the implementation of the CPN framework, and how it can be used to build systems that execute on multi-core or distributed compute hosts. This chapter presents case studies that exercise the CPN framework and demonstrate its capabilities and utility.

Section 6.1 provides an introduction. Section 6.2 presents a case study using the classic Sieve of Eratosthenes, a simple algorithm for finding prime numbers. Section 6.3 presents a case study that exercises the framework implementation with randomly generated program graphs. Section 6.4 presents a 3D circular convolution sonar beamformer and replica correlator that is representative of a real-time signal processing algorithm that can be implemented with the provided framework. Section 6.5 provides a conclusion.

6.1 Introduction

In this chapter, I provide the results of benchmarks that have been executed on two target platforms. The first platform represents a fairly large multi-core system. It is a server-class machine with two 2.66 GHz Intel Xeon "Westmere" model X5650 processors and 6 GB of memory. Each processor has six-cores with Hyper-Threading [78], so this platform appears to have 24 processors. This server is running Red Hat Enterprise Linux 5.5.

The second target on which these CPN benchmarks have been performed is a cluster system that has been maintained specifically for this project. This cluster has a root node and 7 diskless compute nodes that are booted from the root over a gigabit Ethernet network. The nodes are additionally networked together with InfiniBand [74], a high-speed low latency communication link used in high-performance computing. These links use 4X InfiniBand, which operates at 8 gigabits per second. Each node in this cluster contains a pair of dual-core 2.33 GHz Intel Xeon "Woodcrest" model 5148 processors. This cluster is running Red Hat Enterprise Linux 5.5, the root node has 16 GB of memory, and each compute node has 8 GB of memory.

The code that implements these case studies is included as part of the CPN framework distribution. I begin with the Sieve of Eratosthenes.

6.2 Sieve of Eratosthenes

The first example that Kahn provides [48] of a Process Network performing a useful computation is the Sieve of Eratosthenes [79], which is a simple method for finding prime numbers. Eratosthenes was a Greek mathematician who lived circa 200 BC [80]. The algorithm is extremely simple, operating on a list of sequential integers from 2 up to some value N:

1. pick the next number from the start of the list and declare it a prime,
2. remove all multiples of this prime from the list, and
3. repeat.

After reaching a prime that is greater than or equal to $\sqrt{N}$, the algorithm ends and all that remains in the list are the primes up to N.

Kahn attributes to [81] the prime sieve in the form of concurrent processes that filter multiples of prime numbers. This form of the algorithm is simple as well, and starts with a base process that creates a stream of sequential integers starting at 2. Following the base process is a series of filter processes. Each filter process reads the first number on its input and declares that number a prime. It then repeatedly reads from its input, eliminates any multiples of its prime, and sends the remaining integers to its output. Each filter in the sequence is filtering the next prime number. Figure 6.1 shows a prime sieve in the form of a series of filter processes.

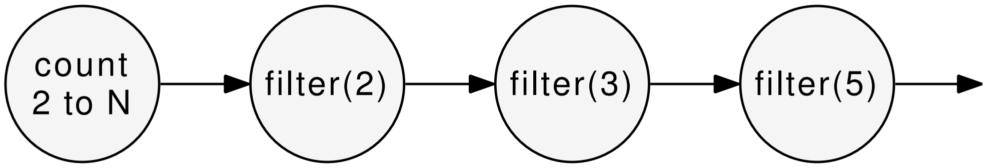


Figure 6.1: A prime sieve as a series of concurrent processes.

The Sieve of Eratosthenes is an example of an algorithm that requires dynamic execution, and cannot be implemented by more restrictive static models of computation such as SDF (Section 1.4.2) or CG (Section 2.4). The prime sieve also demonstrates the composability of the PN model, also not available in SDF or CG. The number of filters that need to exist in the sieve is not known in advance. Instead, the series of filtering processes is created recur-

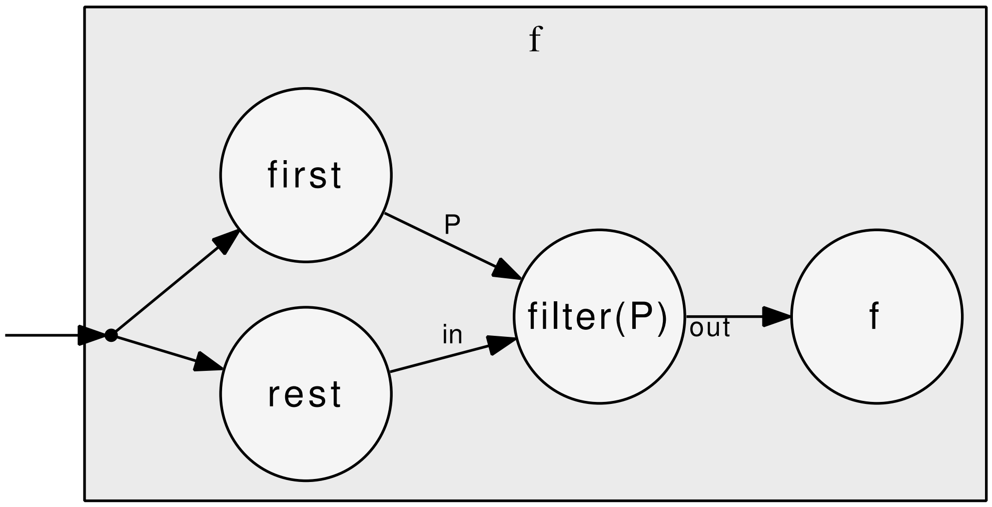


Figure 6.2: A recursive node f that creates prime filters.

sively, and each process creates an identical subprocess to where it can send its filtered output. Figure 6.2 shows the definition of a node f that recursively creates prime filters. This definition also separates the first token, known to be a prime (P), from the remaining prime candidates to be filtered.

6.2.1 Implementation

One thing to notice about the prime sieve is that there is a significant imbalance in the quantity of prime candidates at each node. The closer a node is to the beginning, the more prime candidates it must examine. This makes the first several nodes an extreme bottleneck. Why should the base node emit even numbers when they are immediately going to be filtered? Why not emit a 2, followed by only the odd numbers up to N? The prime wheel [82] is a generalization of this idea, and is commonly used as an improved emitter of prime candidates.

The "circumference" of a prime wheel of order M is the product of the first M prime numbers. The "spokes" are the numbers that are prime candidates each time the wheel revolves. For example, a 2^{nd} order prime wheel has a circumference of $2 \cdot 3 = 6$. All prime candidates greater than 6 must be of the form $6k+1$ or $6k+5$, where k is a positive integer. This method significantly reduces the number of prime candidates, but does not scale well. For example, a 6^{th} order wheel has a circumference of 30030, and an 8^{th} order wheel has a circumference of 9699690.

For implementation of a prime sieve in the CPN framework, one could make the observation that a filter node does only a very small amount of work per execution: it consumes a single prime candidate from its input, determines whether the candidate is a multiple of its prime number, and if not, places the single candidate on its output queue. As discussed in Section 5.3, CPN nodes should execute for at least on the order of a thread context switch to avoid being dominated by dynamic scheduling overhead. One way that prime sieve filter nodes can be made to do more work at each firing is with multi-token transactions, a feature of the CPN framework. A node can read a set of multiple prime candidates, eliminate all of the multiples of its prime number, and then send the remaining list of non-candidates to its output.

An additional method for increasing the amount of work at each filter node is to have it filter for more than one prime number. A naive approach would be to have a multiple-prime filter node perform the same operations that multiple single-prime filter nodes perform (as in simple node clustering). However, I implement a node that filters multiple primes more efficiently. This node maintains a heap data structure with one entry per prime number being filtered. Each entry in the heap contains a current working multiple of the prime, and the heap is reverse-sorted on these working multiples.

To check if a candidate is prime, it is compared to the smallest working multiple in the heap. If the candidate is larger than the smallest working multiple, the working multiple is increased (by a multiple of the prime number) and the heap is reheapified. This is repeated until the candidate is not larger than the smallest working multiple. If the candidate is equal to the smallest working multiple, then the candidate is not prime and therefore blocked by the filter as a composite of the same prime as the working multiple. If the candidate is smaller than the smallest working multiple, then it has passed the sieve and may be a new prime number. An original version of this node increased working multiples by twice the prime (thus skipping the multiple of two). The current version alternates between skipping twice the prime and four times the prime, thereby making a prime wheel of order 2.

A multiple-prime filter node is initialized to know the maximum number of primes that it should filter. When a candidate passes the sieve and the filter node has room for an additional prime, the candidate is a new prime number. The filter node adds a new entry for the prime to the heap and sets the working multiple value to the square of the new prime. If the filter node is already filtering its instructed maximum number of primes, it will send the passing candidate down the line to the subsequent filter node, creating one if it does not exist. If primes are being computed only up to some number N, reaching a prime of $\sqrt{N}$ or greater indicates that no further filtering primes are needed, and all further incoming numbers are prime.

6.2.2 Results

Figure 6.3 shows execution results on the 24-core computer for several different variations of the prime sieve, versus the number of prime candidates N. For each point on the plot, several runs of the prime sieve are performed and the minimum execution time taken, which includes setup and tear down

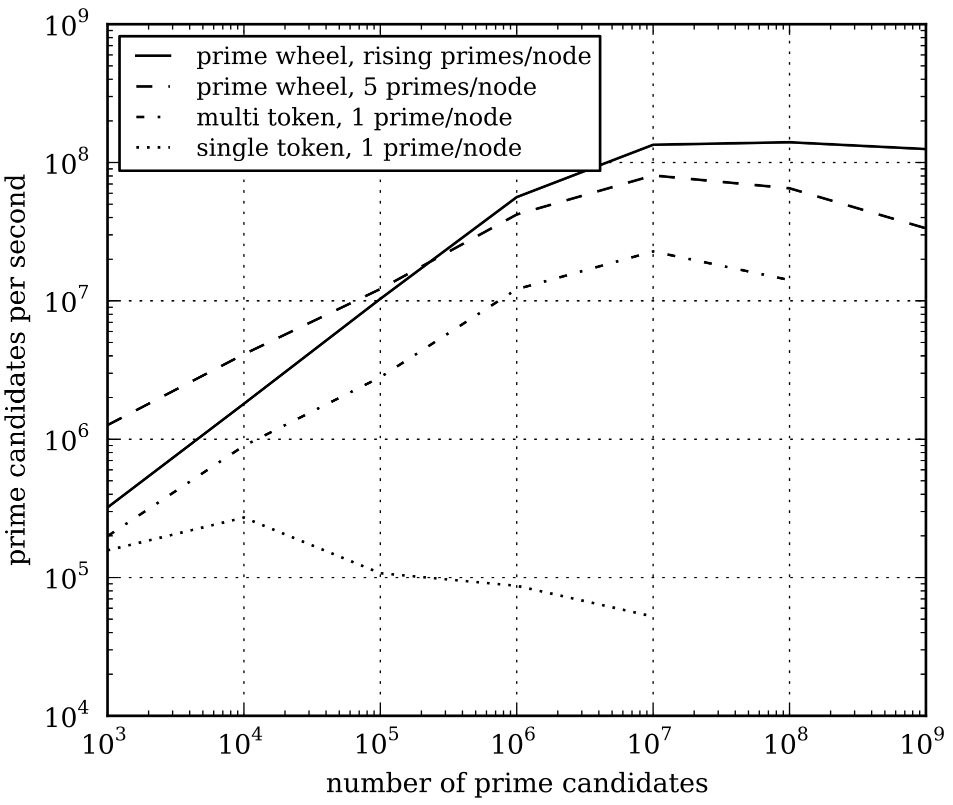


Figure 6.3: Prime sieve results on target #1 (large SMP).

of the full CPN graph. The rate of prime filtering is simply computed as the number of prime candidates divided by the execution time.

The series labeled "single token, 1 prime/node" is implemented as Kahn did in [48]: the base node generates a sequential series of integers (starting at 2), each filter node consumes one token at a time, and each filter node filters a single prime number. The *prime-counting function*, $\pi(N)$, gives the number of primes less than or equal to N [83], and $\pi(N) \approx N/\ln N$. For $N = 10^4$, the sieve can stop adding new filters when a prime greater than $\sqrt{N} = 100$ has been reached. This occurs when there are about 22 filter nodes (and threads). For $N = 10^7$, this estimate grows to 392 filter nodes.

Again, to prevent significant scheduling overhead due to thread context switching, CPN nodes should perform larger-grain computation. The series labeled "multi token, 1 prime/node" simply uses the multi-token firing threshold feature of the CPN framework. For this and the remaining series, firing thresholds are set to 2500 tokens. A filter node will block until it has 2500 prime candidates in its input queue, and it will also block until there is sufficient space for 2500 in its output queue. Once these conditions are met, the node filters all available prime candidates. Clearly this is a big improvement (over 400 times faster at $N = 10^7$), but still has scaling problems. For $N = 10^8$, this is about 1085 filtering primes and filter nodes.

The series labeled "prime wheel, 5 primes/node" uses the multiple-prime filter node (from Section 6.2.1) to filter for 5 primes at each node. This gives an additional performance improvement, but ultimately only divides the number of filter nodes by a constant and postpones the scaling problem. For $N = 10^9$, this is about 3052 filtering primes in 610 filter nodes.

The final series is labeled "prime wheel, rising primes/node". Here, each filter node computes how many primes it should filter based on its distance away from the base node, as specified by simple polynomial coefficients. In this example, the polynomial used is $\lfloor 0.005k^3 \rfloor$, where k is the distance of the filter node from the base node. (If the result is less than one, then one prime is used.) With this polynomial, the first seven filter nodes use a single prime, and then that number grows rapidly. For $N = 10^9$ there are 41 nodes with the 3000 filtering primes, and the final nodes are filtering hundreds of primes. At $N = 10^7$, this is more than 2500 times faster than the original case of single token firings and one prime per node.

Changing the polynomial will change the number of primes being filtered at each node, and the shape of the final series. The polynomial affects the number of threads and the amount of work performed in each thread. To tune for a particular problem size (N, the number of prime candidates) and

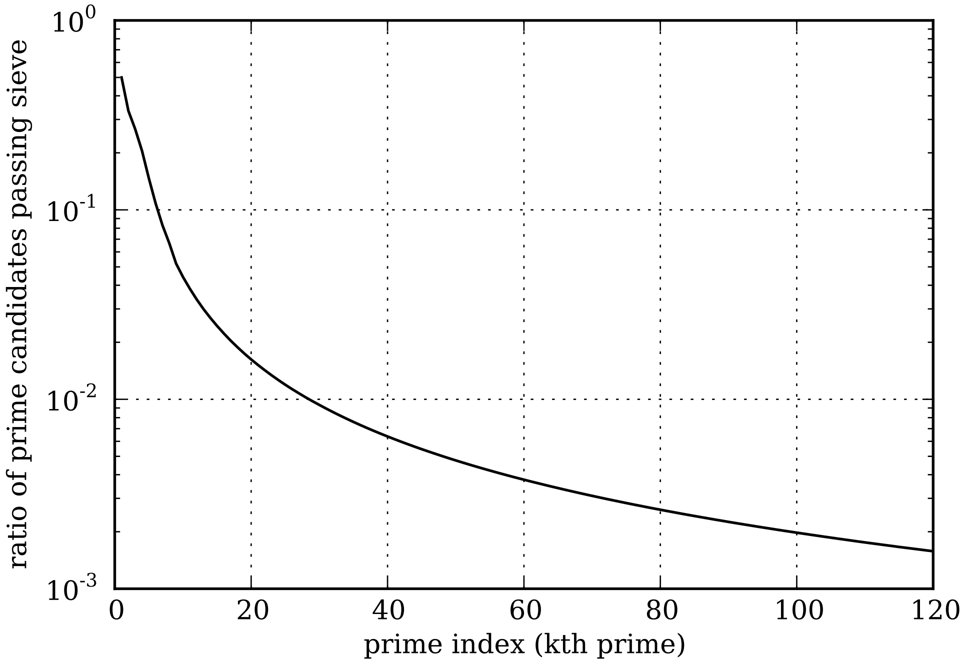


Figure 6.4: An estimate of the work performed at each prime filter.

amount of concurrency, one could compute the primes per filter as a function of N and the number of desired filter nodes.

A way to estimate the amount of work at each filter node is by estimating how many prime candidates it must filter. Figure 6.4 shows the ratio of prime candidates that remain after filtering up to the k^{th} prime. This illustrates the severe imbalance in workload, and the motivation for using a prime wheel for the first several primes. The *primorial* (or prime factorial), $p_k\#$, is the product of the first k primes [84]. This workload ratio is approximately equal to (6.1).

$$\frac{\pi(p_k\#) - k + 1}{p_k\#} \tag{6.1}$$

Again, $\pi(x)$ is the prime-counting function [83].

In any case, this prime sieve case study is included because of historical significance to Process Networks, and because it demonstrates some design choices when using the CPN framework. If a fast prime number generator is needed for a practical purpose, a more modern algorithm should be used.

Figure 6.5, provides execution results from the second target platform, the cluster computer. In this case the same CPN program is run repeatedly, each with a different mapping of prime filter nodes onto different computers in the cluster. The primes up to $N = 10^9$ are computed, again using a 5-prime wheel and a rising number of primes per node. Because each computer has 4 processors, the first 4 CPN nodes are run on the first computer and the next 4 nodes on the next computer. When there are more CPN nodes than

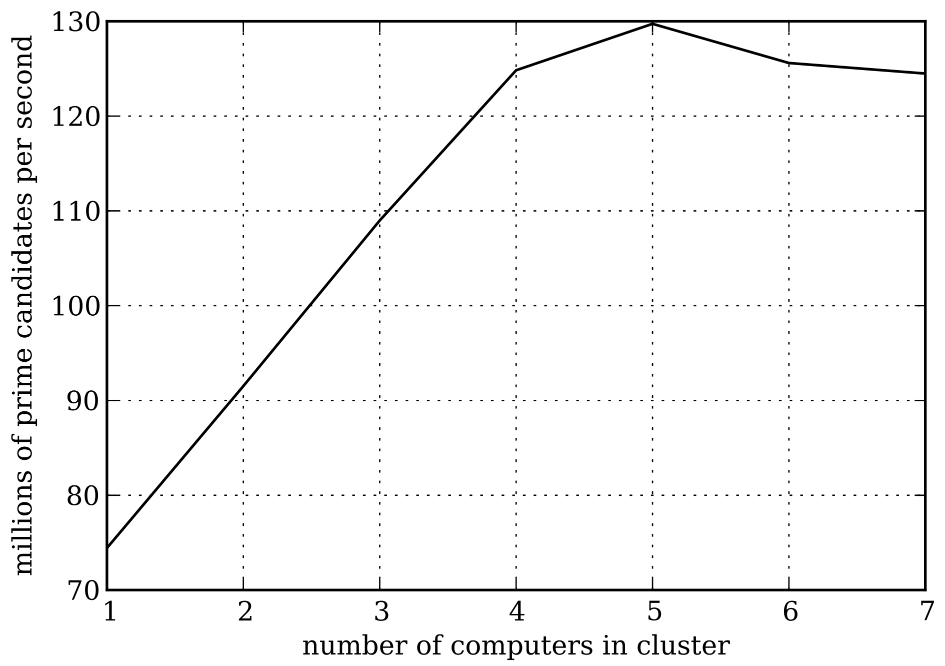


Figure 6.5: Prime sieve results on target #2 (cluster).

computers, they cycle around again to the first computer. That is, the k^{th} CPN node is run on computer number $\lfloor k/4 \rfloor \mod M$, where there are M computers being used.

The prime sieve is effectively a pipeline, and the performance is limited by the slowest stage. In the Sieve of Eratosthenes, the first stages are the slowest stages by a large margin (based on their unbalanced workload as shown in Figure 6.4). This significantly restricts the scalability of the prime sieve. There is some speedup as computers are added in this distributed case, but performance is hamstrung from the source. I have not spent significant effort tuning this prime sieve for the cluster system. Again, it is of historical significance, and more modern algorithms exist for finding prime numbers.

6.3 Randomly Generated Graphs

The second case study is included to demonstrate the robustness and stability of the CPN framework. It has also been used during development as a tool to find and resolve implementation problems. If my claim is to “leave your concurrency issues to the CPN framework,” then the implementation had better be solid. This case study pseudo-randomly and dynamically builds and modifies CPN program graphs. Queues are randomly created, briefly used for communication, and then deleted. With some probability, nodes are randomly created and randomly expire. In particular, this case study stress-tests the CPN Kernel (Section 5.2) and the CPN Remote Context (Section 5.5.1).

To randomly generate graphs, I begin with a simple pseudo-random number generator: a linear feedback shift register (LFSR). An LFSR is a shift register with an input bit that is an exclusive-or combination of other bits in the register. The bits that are combined and fed back are called the *taps*, and they describe a *feedback polynomial*. For a register of n bits, a *maximal* LFSR will create a sequence of $2^n - 1$ numbers before repeating, and every possible n-bit value will be represented except for 0. Creating a maximal LFSR is a simple as using a *maximal-length* feedback polynomial, which can be computed or found in a reference. LFSRs are easy to implement in both hardware and software, and are used in many applications including digital counters, cryptography, and communications [85]. Using an LFSR, a determinate pseudo-random sequence of a chosen length can be fully described by two integers: the polynomial and the *seed*, or initial value.

In this case study, each node in the system uses an LFSR (with identical parameters) to decide what operation it will perform. Each node contains an instantiation of the `RandomInstructionGenerator` class, which treats the pseudo-random series of integers as computer "opcodes" to describe how the system will operate. There are only four opcodes:

CREATE creates a new node,

DELETE deletes an existing node,

CHAIN marks a node to be used in a communication chain, and

NOOP is no operation.

The probability with which CREATE and DELETE operations will occur is specified as a parameter at initialization (currently set to 1% each). DELETE gets an argument describing which node should be deleted by taking the next random number from the LFSR. Nearly all the remaining opcodes map to the CHAIN operation. Which node to use in the chain is given by the integer value of the CHAIN opcode itself. The number of CHAIN opcodes is the largest multiple of the number of live nodes in the system (so that there is no bias in the selection of nodes for chaining). The remaining opcodes are NOOP.

The initial number of nodes is also an initialization parameter. Each node is numbered in a sequence starting at zero. Newly created nodes are put at the end of the sequence, but nodes can be deleted from anywhere (as given by a random argument). Once a node number is deleted it cannot be reused. If there is only one node, it cannot be deleted.

The `RandomInstructionGenerator` maintains a list of nodes that have been selected for chaining. This list grows as CHAIN opcodes are issued, until there is a collision: the specified node is already in the chain list. Upon this collision, the process of actually building the chain begins.

The first node in the chain list becomes a producer. The producer creates a queue from itself to the second node in the chain, sends a few tokens (once the connection is made), and releases the queue. The second node in the chain becomes a transmuter. A transmuter creates a queue from itself to the next node in the chain. It reads the tokens from its input queue, verifies their value, and releases the input queue. It then sends tokens on its output queue and releases the output queue. This sequence continues until we reach the final node in the chain list. The final node in the chain becomes a consumer. The consumer simply reads and verifies the tokens from its input, and then release the queue.

Each node has its own `RandomInstructionGenerator` instantiation, and the nodes are not synchronized (because each runs in a separate thread). However, nodes block when trying to attach queues to a peer, and this serves as a synchronization between them. When a node is created or deleted,

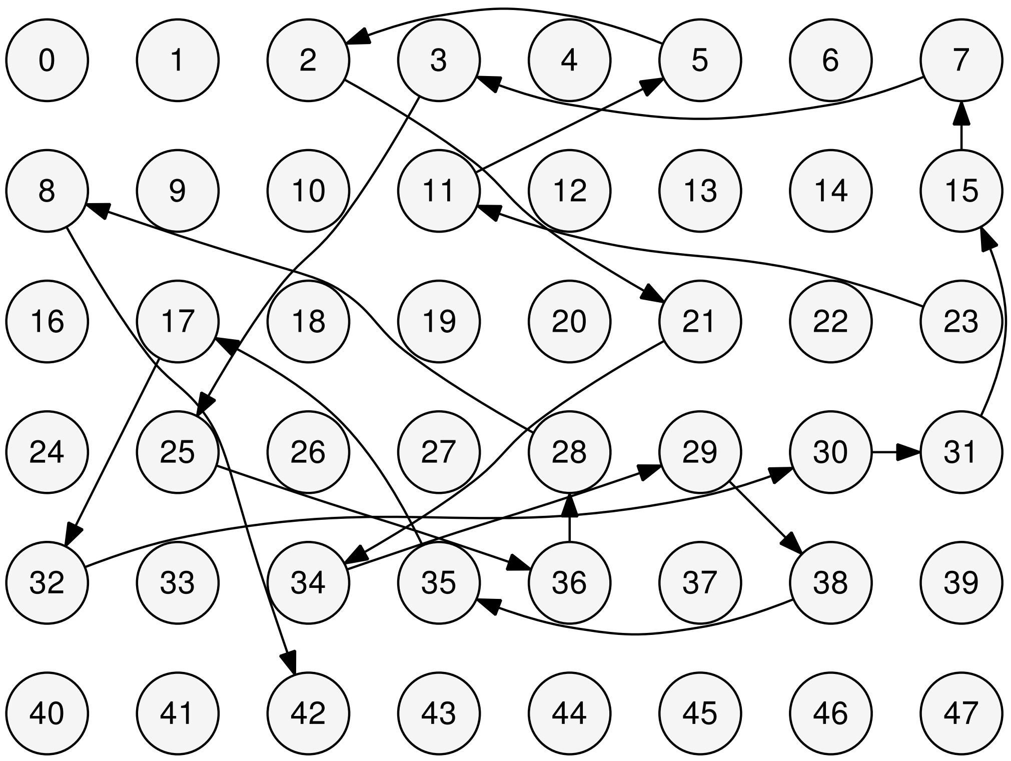


Figure 6.6: An example of a randomly generated CPN graph.

every node waits for that operation to complete (via `WaitForNodeStart` or `WaitForNode` interfaces to the CPN Kernel). This also serves as barrier synchronization for the system.

Figure 6.6 shows an example of a random CPN graph that is generated by `RandomInstructionGenerator`. To create this figure, the textual output was parsed and converted, and then automatically rendered with Graphviz [86], an open source graph visualization software package.

The programs to execute this case study are included in the CPN software framework distribution, and have repeatedly been executed both on the 24-core computer, and on the cluster (each described in Section 6.1). The program `RandomInstruction` executes the test from within a single (Unix) process. The initial number of nodes and the number of opcode iterations to execute are specified as command-line parameters. A local CPN Context is created and all kernel interactions are via function calls with mutual exclusion (mutex) locks. More than one CPN Kernel (in the same single process) can also be specified on the command line. In this case, the nodes are assigned to the multiple kernels in a round-robin fashion. This stress test exercises the CPN Kernel and interaction with a local CPN Context, and executes successfully with 50000 nodes. It has repeatedly run for a millions of iterations on 1000 nodes for more than 72 hours without failure.

The program `RemoteRandomInstruction` executes the test in a distributed fashion. It requires that a `RemoteContext` daemon is running (Section 5.5.1). Multiple copies of `RemoteRandomInstruction` can be run, and all will coordinate and communicate via the daemon. In this case, the initial number of nodes, the number of opcode iterations, and other parameters are specified in a JSON [75] configuration file. This stress test exercises the CPN Kernel, the `RemoteContext` daemon, and their interaction. This test has repeatedly run for millions of iterations on the cluster system using 8 hosts and 1000 nodes, and it has run for more than 72 hours without failure.

If all goes well, the output generated by each of these program is uninteresting. They should and do execute as instructed, and run to completion without error. The execution time is fairly inconsequential. The stated goal of this test is to demonstrate robustness and stability of the CPN framework, and no effort has been made to optimize the opcode generator and parser. As a point of reference, `RemoteContext` can execute with 1000 nodes and 10000 opcode iterations in under 3 seconds. Unsurprisingly, increasing the number of nodes, opcode iterations, or kernels will increase execution time. A distributed execution across a network will also increase execution time.

Valgrind[87][88] is a set of dynamic analysis tools for instrumenting programs and examining their behavior. Valgrind can perform profiling and automatic detection of various types of errors, such as memory management errors. This random CPN graph case study has also been run through the valgrind memory checking tool, *memcheck* [89], with no errors found. The CPN framework contains extensive unit tests, and has also passed the memcheck tests.

The final case study is an example of a high-throughput signal processing system, the intended target application domain for the CPN framework.

6.4 Sonar Beamformer and Correlator

Sonar is a method for using acoustic waves to detect and locate objects or environments, typically underwater [90]. The origin of the word sonar is the acronym **So**und **N**avigation **a**nd **R**anging. Sonar can be used for navigation, obstacle avoidance, communication, underwater mapping, and detection and identification of other vessels or objects. A *passive* sonar operates only by listening to sound in the environment. An *active* sonar emits sound (called a *ping*) and listens for reflections. The distance to an object can be measured by the difference in the time of the ping and the arrival time of the object's reflection.

The receiver in a sonar typically consists of an array of sensor elements that convert sound pressure to a voltage. This voltage can be amplified and digitized, and then digital signal processing applied to compute a desired result. A *beamformer* is spatial filter that can focus an array of elements to determine from which direction a sound arrived. A simple way to implement a beamformer is as a weighted delay and sum of sensor elements [91], where the delays are the signal propagation time from each of the sensors onto a plane that is perpendicular to the desired steering direction. In (6.2), a single beam output $b(t)$ is computed from a number of sensors $x_n(t)$ that are weighted by α_n and delayed by τ_n.

$$b(t) = \sum_{n=1}^{N} \alpha_n x_n(t - \tau_n) \tag{6.2}$$

To form an image, it is desirable to "look" in many directions simultaneously by forming multiple beams that are steered in multiple directions. Figure 6.7 shows an example of a set (or *fan*) of beams that could be formed for a sonar system.

More elements can increase the sensitivity of an array, so that it can detect smaller signals. More elements can also decrease the width of each beam, yielding a more accurate measurement of a signal's arrival angle. When the beams narrow, more beams must be formed to cover the same angular

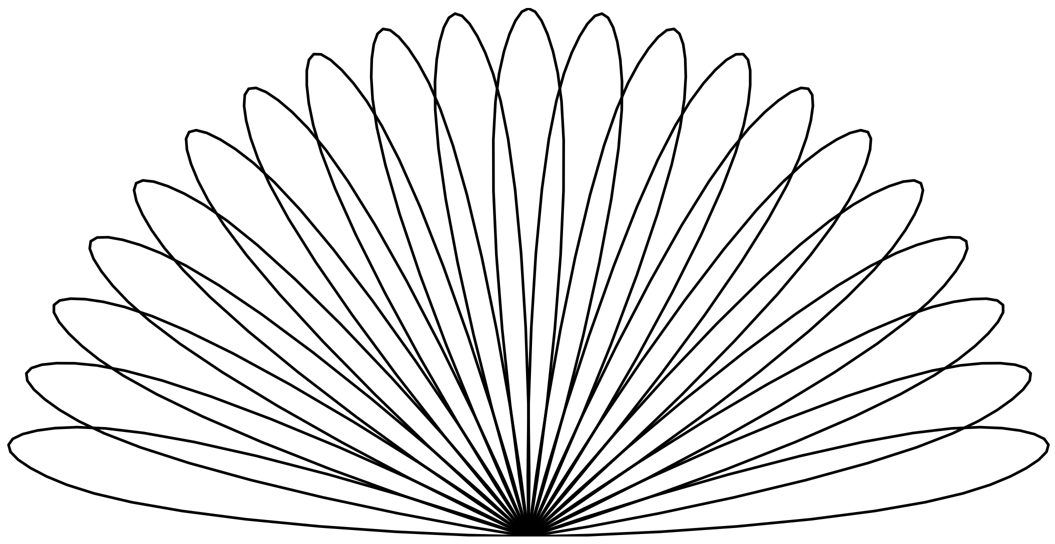


Figure 6.7: An example of multiple beams formed for a sonar system.

sector without introducing gaps in coverage. Also, the range resolution of a sonar (accuracy of distance measurements) is improved by increasing the signal bandwidth, and therefore the sampling frequency. The desire for high resolution drives sonars to many elements and wide bandwidths. Consequently, a beamformer for a high-resolution sonar is a high-throughput, computationally intensive signal processing application. A beamformer is also an example of a multi-channel sliding window algorithm.

Another important component of an active sonar is a replica correlator, also known as a pulse compressor or a matched filter. The received signals should correlate well with what was transmitted. Given that the transmitted pulse is known, the received signals can be filtered to look for that known signal. The optimal filter (not accounting for distortions) is the time-reversed

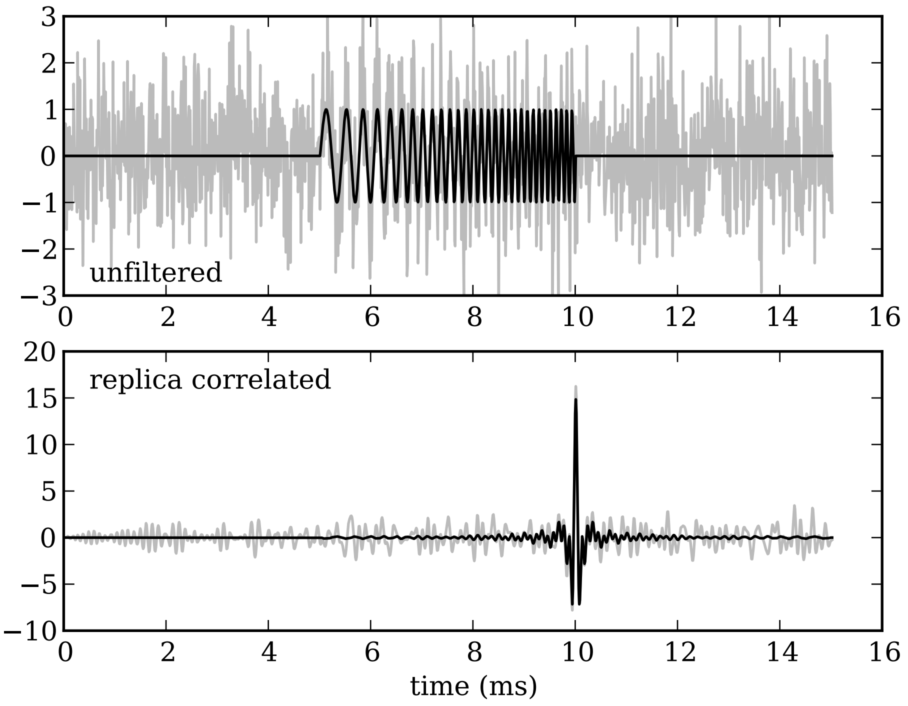


Figure 6.8: An example of replica correlation of a noisy signal.

complex conjugate of the transmit pulse, or replica. Matched filtering maximizes the output signal-to-noise ratio [65], which improves signal detection in the presence of noise. For certain types of pulses such as a linear frequency modulated (LFM) sweep or *chirp*, matched filtering will also compress the received signal in time, thus augmenting the range resolution.

Figure 6.8 shows an example of replica correlation. The upper plot contains an LFM chirp, and the same chirp with additive noise masking the signal (in gray). The lower plot contains the result of replica correlation for both signals. The signal amplitude is significantly increased and is now easily discernible above the noise, and the pulse has been compressed in time.

The benefit from matched filtering is proportional to the product of the pulse's duration and bandwidth, or the time-bandwidth product. The desire for high resolution drives sonars to longer pulses and higher bandwidth, both directly increasing the amount of computation required to implement the filter. A replica correlator is also an example of high-throughput, computationally intensive signal processing application, as well as a multi-channel sliding window algorithm.

6.4.1 Algorithm Description

This case study uses the same sonar array and beamforming algorithm that I presented in [92]. The array geometry is a cylinder with 256 *staves* around a circle, where each stave is a vertical column of 12 elements. This array has a total 3072 total elements. Each of the staves lies on one of 560 points that is equidistant around the circle. Figure 6.9 shows a top-down view of one quarter of the array. There are regular gaps in the stave spacing around the circle to allow for mechanical structure in the array.

The beamformer in this application is separated into two stages: vertical and horizontal. The horizontal beamformer performs replica correlation in addition to beamforming. Figure 6.10 provides a block diagram of the full beamforming system.

The vertical beamformer (also called a staveformer) forms 3 stave outputs for each vertical column of 12 elements, computed by weighted delaying

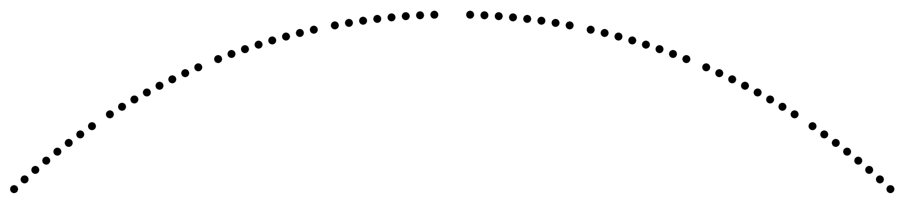


Figure 6.9: A top view of one quarter of the array geometry.

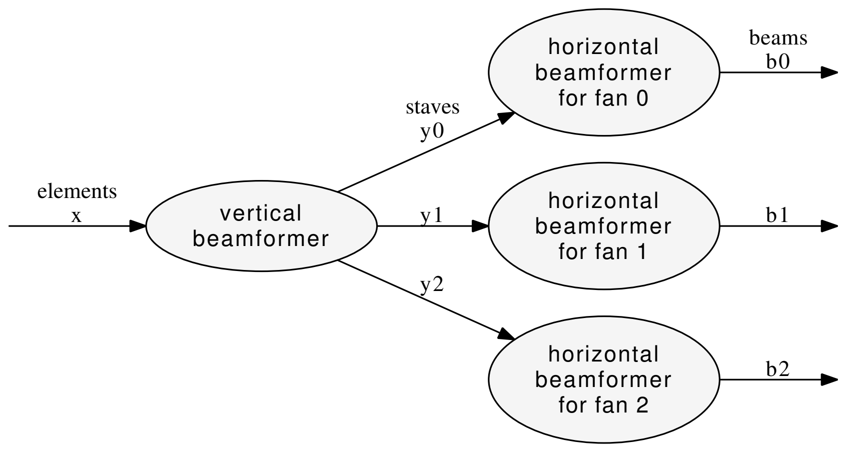


Figure 6.10: A block diagram of the full beamforming system.

and summing. The sampled element data input to the vertical beamformer is $x[s][e][n]$, where s is the stave index, e is the vertical element within the stave, and n is a sample index in time. The element data has already been frequency-shifted to baseband and rate-reduced. The weights and delays for vertical beamforming are applied together by a bank of weighted 4-tap fractional delay FIR filters of the form $h_v[12][4]$. The algorithm from [93] is used to compute these fractional delay filters. Each of the 3 sets of vertical beams can use a different h_v so that the resulting sets of vertical beams can have different steering angles or beam shapes.

$$y_0[s][n] = \sum_{e=0}^{11} \left[\sum_{k=0}^{3} x[s][e][n-k] h_{v0}[e][k] \right] B_0[e] \tag{6.3}$$

The first set of vertical beams is computed as shown in (6.3) for all 256 staves, where B_0 is a correction factor for imposing a time delay at baseband. The other two sets of vertical beams (y_1 and y_2) are similarly computed from h_{v1} and B_1 or h_{v2} and B_2. Each output time sample computed by the vertical beamformer requires about 74 thousand floating point operations per set of stave outputs, or 221 thousand for all three sets.

The stave outputs from the vertical beamformer are subsequently processed by a horizontal beamfomer. Because the staves lie on equidistant points around a circle, the horizontal beam outputs can be computed using circular convolution beamforming [94]. In the general (non-symmetric) case, unique weights and delays could be required for each stave's contribution to each beam. By computing 560 equally spaced beams from the 560 equally spaced points around the circle, the weights and delays become symmetric. Of the 560 equidistant points around the circle, only 256 points coincide with a physical stave. The remaining empty 304 positions are computed as staves of zero, where this zero insertion can be thought of as spatial upsampling. I refer to these spatially upsampled staves as “virtual” staves, with the three sets of virtual staves being y_{v0}, y_{v1}, and y_{v2}. To save on transmission bandwidth, these virtual staves exist only inside the horizontal beamformer.

In this implementation, the weights and delays are again applied at the same time with a bank of weighted fractional delay filters [93]. Because of the symmetry in the weights and delays, this bank of filters can be reduced in size by a factor of 560. In this case, the same weights, delays, and (geometrically relative) staves are used to form each of the beams. The set of horizontal filters is of the form $h_h[M][K]$, where K is the length of each filter and M is 560, both the number of beams and the number of virtual staves. The baseband correction factors are also multiplied into the filters.

The first set of horizontal beams is computed as in (6.4).

$$b_0[d][n] = \sum_{k=0}^{K-1} \sum_{m=0}^{M-1} y_{v0}[m][n-k] h_{h0}[(d-m) \bmod M][k] \tag{6.4}$$

This can also be expressed (6.5) as a circular convolution of vectors in space around the array, and convolution of vectors in time. For additional intermediate steps, see [92].

$$b_0[n] = \sum_{k=0}^{K-1} (y_{v0}[n-k] \overset{M}{\circledast} h_{h0}[k]) \tag{6.5}$$

Circular convolution can be efficiently implemented with the FFT, allowing all 560 beams to be calculated at once. Linear convolution can also be efficiently implemented with the FFT, i.e. by using overlap-save [65]. Using the FFT reduces the algorithm complexity from $O(KM^2)$ operations per sample to $O(KM \log_2 KM)$, and significantly reduces the number of operations required. Matched filtering (replica correlation) is also performed in the horizontal beamformer by convolving the transmitted waveform replica with the beamforming filters. The overlap size simply must be increased by the length of the replica.

This horizontal beamformer operates on blocks of 8192 samples in time, and overlaps adjacent blocks by 2048 samples. For processing one block of data, the steps of the horizontal beamformer are:

1. FFT in time of 256 staves
2. Upsample to 560 virtual staves and FFT in space
3. Multiply by beamforming filter coefficients
4. Inverse FFT in space
5. Multiply by correlation filter coefficients
6. Inverse FFT in time

For the horizontal beamformer (and replica correlator) to compute a single block of 8192 samples, it takes about 908 million floating point operations. Of

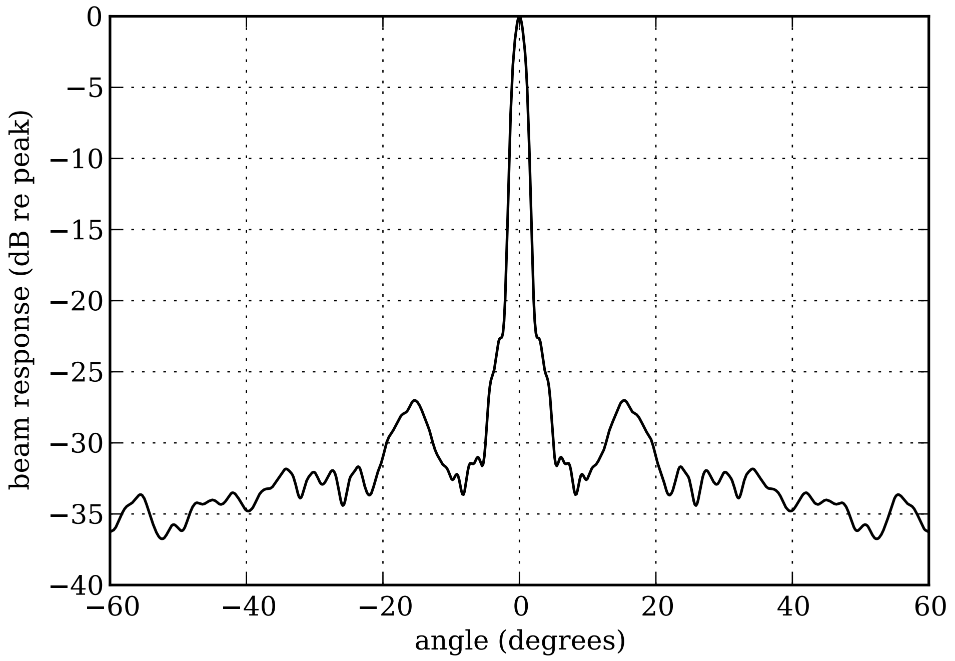


Figure 6.11: A beampattern computed by the presented beamformer.

the 8192 samples produced, 2048 must be discarded because of overlap. This derates the useful work by 25%. One horizontal beamformer therefore requires about 83 thousand floating point operations per sample.

For a nominal sample rate of 50 kHz, the element data rate (with complex 16-bit integers) is about 614 MB/s, the stave data rate (for three fans of complex 32-bit floating point) is about 307 MB/s, and the output data rate (also three fans of complex floating point) is about 672 MB/s. The computational load of the full beamforming system with a 3-output vertical beamformer and 3 horizontal beamformers is about 23.5 GFLOPS.

One way to measure the performance of an array and beamformer is with a *beampattern*, which shows the amplitude response versus angle. Figure 6.11 shows a beampattern generated with the presented beamformer by processing simulated data. A functioning beamformer has a strong response the desired steering direction, and a weak response in other directions. This beamformer has about a 27 dB difference between the main lobe and the next largest lobe. A beamforming system such as this can be used to image an underwater environment in three dimensions [95].

6.4.2 Implementation

These beamforming algorithms have been implemented in C++ and leverage various versions of the Intel Streaming SIMD Extension (SSE) [96] instruction set. These kernels use techniques [97] to achieve high-performance native signal processing. To reduce memory accesses at the vertical beamformer input, this implementation produces all three outputs from a single pass over the element data. To compute the required FFTs, this horizontal beamformer implementation uses the Fastest Fourier Transform in the West (FFTW) [66] library.

The *corner turn* is a commonly used method to improve efficiency when operating on multi-dimensional data [98][99]. A corner turn is simply copying data to change the storage order in memory, typically transposing the rows and columns. Memory is relatively slow, so this can be an expensive operation. This especially true for large data sets that do not fit in the processor's cache.

However, providing sequential memory access for a subsequent stage of the algorithm often leads to a significant overall performance gain. The presented horizontal beamformer kernel implements two corner turns: one before the forward FFT in space, and another after the inverse FFT in space. The first corner turn also upsamples the 256 staves to 560 virtual staves by inserting zeros. Each corner turn operates on a data set of about 35 MB.

The vertical and horizontal beamformer kernels are both good examples of data parallelism, and each use loop parallelization features of OpenMP [8]. The horizontal beamformer additionally uses the thread support that is included in the FFTW library.

On a single processor core from target #1, the vertical beamformer kernel can sustain operation at about 49 thousand samples per second (ksps), which is about 10.8 GFLOPS. Intel lists the base GFLOPS for the X5650 (on all 6 cores) as 63.984 GFLOPS [100]. The vertical beamformer kernel operates at about 102% of this published number (for only a single core).

On the same single processor, a single horizontal beamformer kernel can sustain operation at about 19.1 ksps, or about 1.59 GFLOPS. The horizontal beamformer kernel operates at only about 14.9% of Intel's published base GFLOPS. That is not because of a lack of effort in optimizing this kernel. The execution time of the horizontal beamformer is dominated by FFTW. On a single processor, about 81% of execution time is spent performing the forward and reverse FFT operations. The remaining time is split about equally between corner turns and SIMD vector complex multiplication. FFTW is

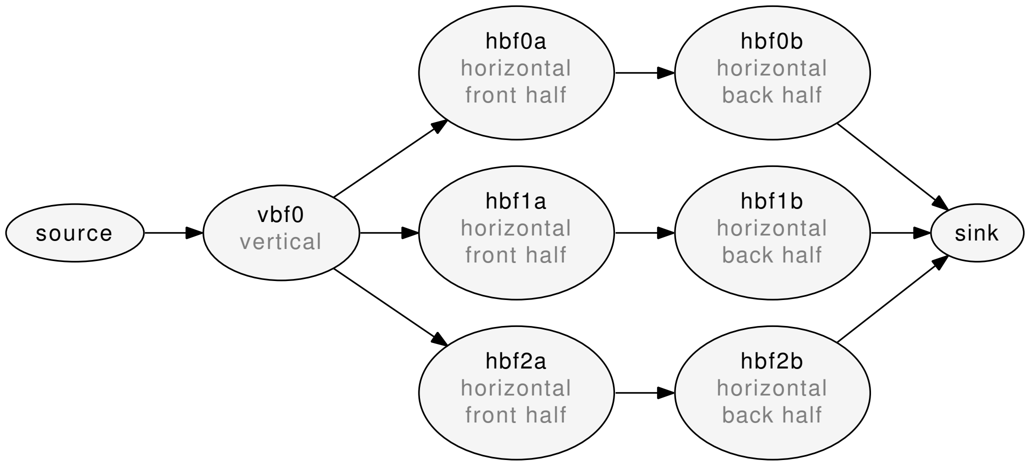


Figure 6.12: CPN beamformer topology on target #1 (large SMP).

known for being computationally efficient, but large FFTs, large data sets, and non-sequential memory accesses do not make efficient use of cache. Consider that if the horizontal beamformer and replica correlator were to be computed without an FFT as in (6.4), they would require about 32.5 million floating point operations per sample. That is about 390 times as many operations as the presented algorithm.

6.4.3 Target #1 Results

To implement this beamforming system within the CPN framework, CPN Nodes were created as described in Section 5.3 to create vertical and horizontal beamformer nodes. Because of the large compute load of the horizontal beamformer, it was split approximately in half. The "front" half executes both forward FFTs and multiplication of the beamforming filter coefficients, and the

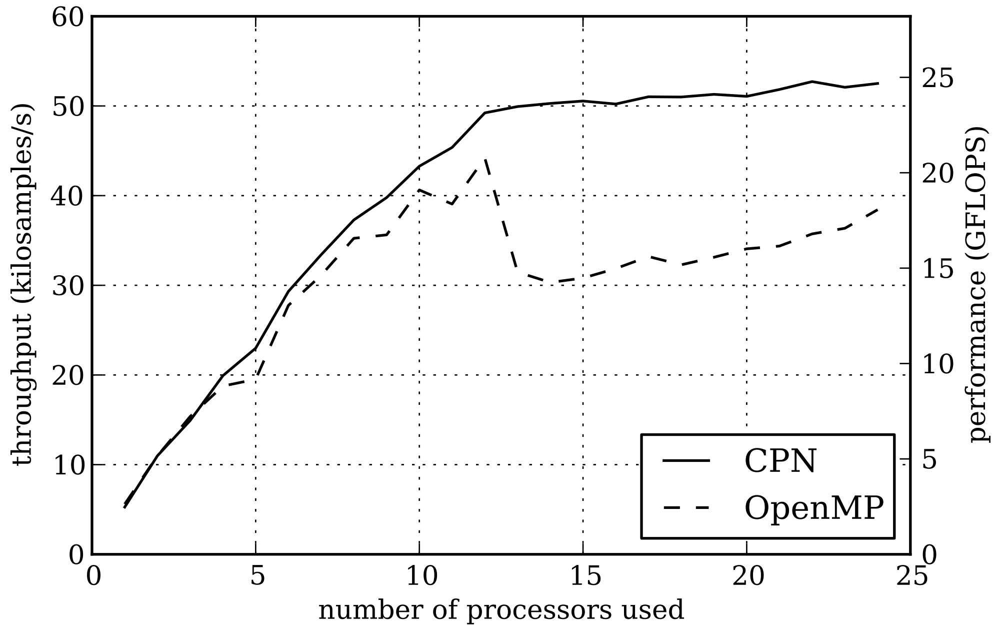


Figure 6.13: Beamformer performance results on target #1 (large SMP).

"back" half executes the remaining steps. Figure 6.12 shows the topology of the beamformer as a CPN program.

Figure 6.13 shows execution results of this CPN beamformer on platform #1, the 24-core SMP computer. As in [92], I also include results using only OpenMP [8] to execute the same optimized beamformer kernels in succession. While varying number of processors in use, I measure the execution time and convert it to beamformer throughput in samples per second. I also compute the GFLOPS based on the number of math operations per sample from Section 6.4.1. On a single processor, both beamformers operate at just over 5 ksps, with the OpenMP beamformer being about 5% faster. The CPN beamformer scales fairly linearly from 1 to 12 processors, with a speedup of 9.3 at 12 processors. At 12 processors, the performance begins to plateau as the transition is made to the Hyper-Thread [78] cores. The CPN beamformer reaches and surpasses the throughput goal of 50 ksps. On all 24 processors, the CPN beamformer operates at 52.5 ksps, which is just under 25 GFLOPS. The OpenMP version scales well for a small number of processors, but is limited in scalability by its sequential model underpinnings. When the Hyper-Thread cores are used, the OpenMP-only beamformer behaves poorly.

6.4.4 Target #2 Results

I also provide performance results on target #2, the cluster system. The processor performance on this system is considerably less than target #1. On a single 4-core compute host, the vertical beamformer kernel (instrumented with OpenMP) can sustain operation at about 99.4 ksps, or about 22.0 GFLOPS. A single horizontal beamformer kernel on a single compute host can sustain about 28.8 ksps, or about 2.39 GFLOPS. Intel lists the base GFLOPS for the 5148 processor as 18.664 GFLOPS [100]. Relative to the published reference, the vertical kernel runs at 118%, and the horizontal kernel at only 12.8%. Again, significant effort has been spent to optimize the horizontal kernel, and it spends the majority of its execution time in the FFTW library.

The same executable from target #1 (and Figure 6.12) runs on a single cluster compute host at only 11.2 ksps, which is about 5.3 GFLOPS and 22% of the target rate. The distributed capabilities of the CPN framework, described in Section 5.5, can be used to increase execution performance on this distributed target. A CPN Remote Context Daemon (Section 5.5.1) can be run on one host of the cluster, and a CPN Kernel run on each compute host in the cluster. CPN Nodes can then be distributed on other compute hosts as instructed by a mapping file, as described in Section 5.6. By changing only the mapping file, the same program runs at 41 ksps. Referring to Figure 6.12, the source node and the vertical beamformer node are run on one host, and every other CPN node is run on its own host. This is only a speedup of 3.7 on 8 hosts, but there is significant overhead due to high-throughput network traffic. At 41 ksps, the network bandwidth into the sink node is 551 MB/s, or 55% of the 8 gigabit InfiniBand link.

I expect the horizontal beamformer nodes to be a bottleneck, because the beamformer kernel benchmarks have shown them to be the slowest nodes in the feed-forward network. The horizontal beamformer compute hosts have some idle time, indicating that they are not fully overlapping computation and communication. To address this, I add more computation that does not depend on expensive external network communication. The horizontal beamformer kernel operates on a large data set (about 36.7 MB), and breaking it into a pipeline requires each stage to have the entire data set. To gain more computation without additional communication, I add data parallelism to each horizontal beamformer with simple fork and join nodes, for round robin time multiplexing as shown in Figure 6.14. The fork node divides the incoming stream into multiple data sets (with the necessary overlap) for multiple con-

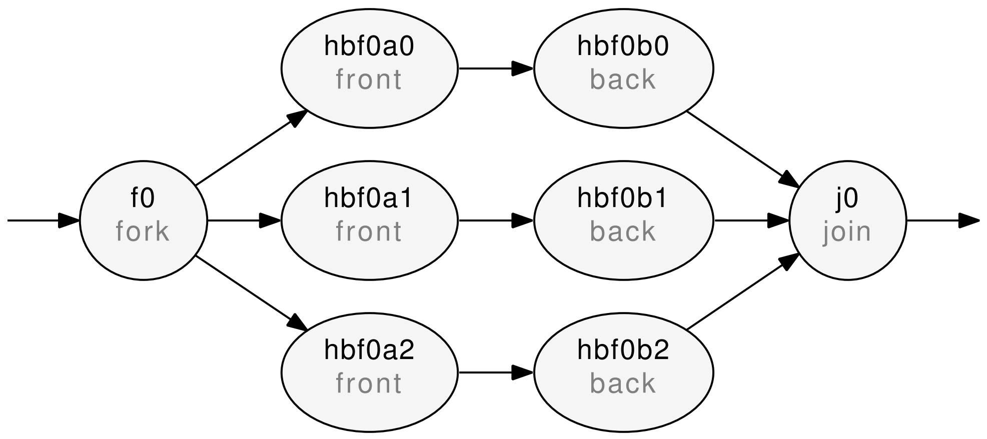


Figure 6.14: Fork and join round robin within a single horizontal beamformer.

current beamformers, and the join node restores the results to a single stream. These nodes incur some expense because they are copying high-throughput data, but they can also reduce the data rate into and out of the servers running horizontal beamformer nodes.

Figure 6.15 shows the beamformer results on target #2, the cluster system. On a single compute host, the overhead of the fork and join make the round robin version a few percent slower. However, the throughput on the full cluster is 51.8 ksps for the case of 3-way round robin. This surpasses the throughput goal of 50 ksps, and operates at about 24.3 GFLOPS. This is a speedup of 4.6 over the original non-time-multiplexed case from Figure 6.12. The data rate entering the sink compute host is now 696 MB/s, which is 69.6% of the 8 gigabit InfiniBand link.

To achieve this level of performance, simple load balancing was per-

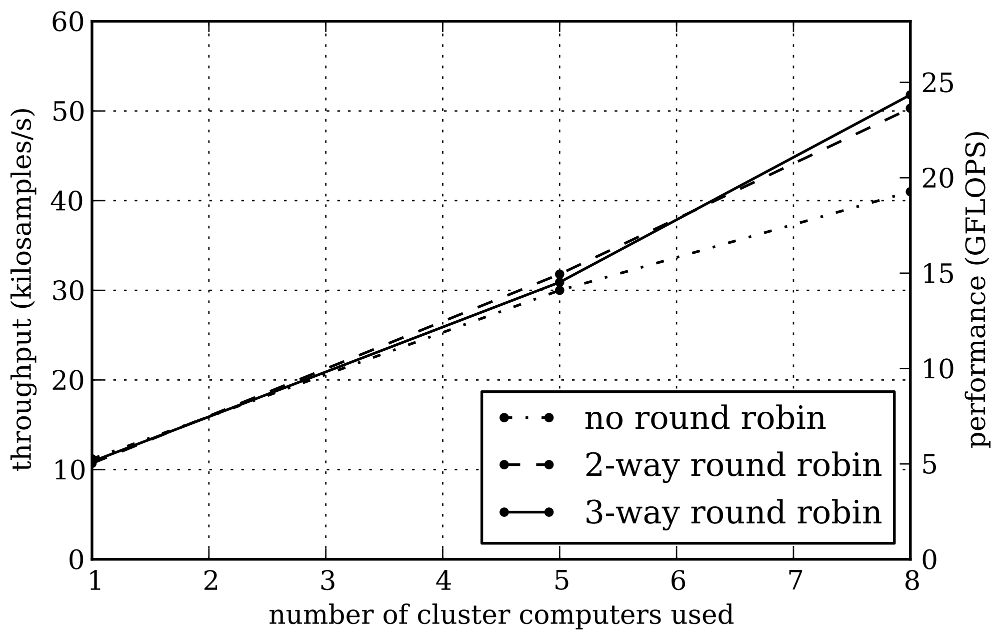


Figure 6.15: Beamformer performance results on target #2 (cluster).

formed by manually modifying the node mapping file. This was without any need to modify the executable program. The best results are obtained when the horizontal beamformers are equally distributed among 6 compute hosts, and those 6 compute hosts contain only horizontal beamformers. The fork nodes are grouped with the source and vertical beamformer, and the join nodes are grouped with the sink. Figure 6.16 shows the full beamforming system as executed on the cluster, including each node mapping.

In this case, mapping the CPN nodes onto compute hosts in the cluster was performed by inspecting the load on each host and manually editing a file. Because it is easy to change this file, exploring the design space of node

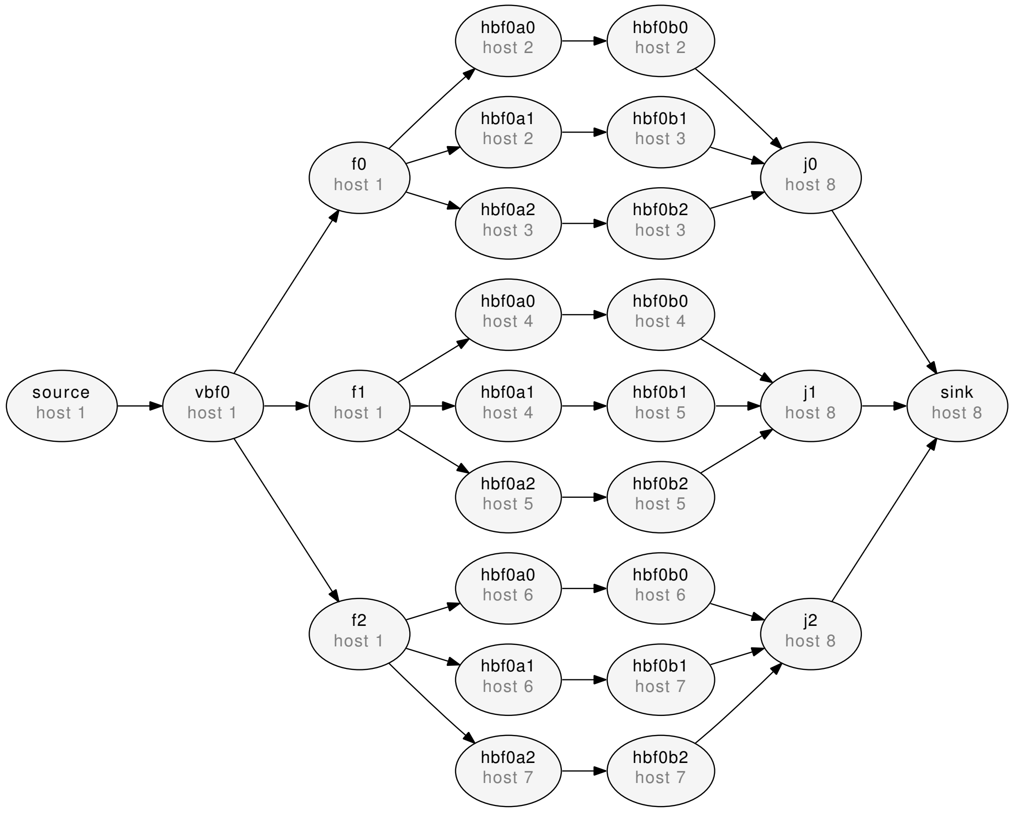


Figure 6.16: The full CPN beamformer and mapping on target #2 (cluster).

mappings is also easy. Because this mapping file is in a format that can easily be automatically parsed and modified by a computer, automated mapping is an interesting avenue to be explored.

I also explored a two-fan beamforming system (refer to Figure 6.10). Rather than forming three sets of staves and having three horizontal beamformers, there are only two. This can easily be explored because the number of fans is a command-line argument to the beamformer case study application.

This reduces the computational load by about 50%, and ideally would increase the beamformer throughput by a corresponding 50%. This two-fan case also used 3-way round robin in the horizontal beamformers. The measured beamformer throughput in the two-fan case is 74.1 ksps, which represents a 43% increase in throughput.

6.5 Conclusion

This chapter presents three case studies that exercise the CPN framework and demonstrate its capabilities and utility. These case studies are executed on two different target platforms: a fairly large SMP system with 12 cores and 12 Hyper-Thread cores, and a cluster with 8 quad-core processors and a high-speed interconnect.

The first case study is the classic Sieve of Eratosthenes, a simple algorithm for finding prime numbers. Using multi-token firings, multiple filtering primes per filter, and some load balancing, the rate of prime computation was increased several thousandfold over the base case on the SMP system. This case study also operates on the cluster system, but does not see much speedup because of the significant load imbalance in filters of a prime sieve. The prime sieve case study is included because of historical significance to Process Networks, and because it demonstrates some design choices when using the CPN framework.

The second case study, which randomly generates CPN graphs, demonstrates the robustness and stability of the CPN framework. This case study pseudo-randomly and dynamically builds and modifies CPN program graphs. Queues are randomly created, briefly used for communication, and deleted. With some probability, nodes are randomly created and randomly expire. This case study has been executed on both target platforms for many hours without failure.

The final case study is a 3D circular convolution sonar beamformer and replica correlator, which is an example of a high-throughput signal processing system. This beamformer has a nominal output data rate of 672 MB/s, and requires about 23.5 GFLOPS to compute its result. This system achieves its real-time goal of 50 ksps on both target platforms, and achieves near-linear speedup on the SMP target.

Chapter 7, which follows, concludes this dissertation.

Chapter 7

Conclusion

This dissertation presents the Computational Process Network model, which extends Kahn's formal Process Network model to make it efficiently implementable. Specifically, it adds multi-token transactions, multi-channel queues, firing thresholds, and zero-copy semantics. This dissertation presents the distributed dynamic deadlock detection and resolution (D4R) algorithm which is required in general for bounded execution of PN, and suitable for use with either KPN or CPN. This dissertation also presents a CPN framework implementation and case studies. The CPN framework implementation targets high-throughput computationally intensive algorithms on workstations and workstation clusters.

7.1 Conclusions

Chapter 1 provides an introduction, and motivates the need for formal models in concurrent computing systems. It describes desirable properties for a formal model such as determinism, scalability, boundedness, and composability. It provides a very high-level view of modern computer system architectures, discusses common practices for programming of concurrent systems, and discusses common problems encountered with these approaches. It provides a background on formal models for concurrency that can be used to address these difficulties, introduces Kahn's Process Network model [35], and proposes the Computational Process Network model.

Chapter 2 covers Process Networks in greater detail. It introduces Kahn's simple semantics of blocking *get* and non-blocking *put*. It recreates Kahn's illustrative sample program in the style of an object-oriented C++ programming language, discusses the formal consequences, and details the underlying mathematical model. A PN program can be described by system of continuous equations on streams. This set of equations has a unique least fixed point solution that corresponds to the behavior of the executing processes. The value of tokens on all of the queues depends only on the solution to the fixed point equation, not on execution order of the processes in the system. Kahn's model is composable and determinate. However, termination of a general PN program is undecidable in finite time, as is boundedness of the queues.

Parks [36] attempts to address boundedness with clever scheduling of the processes. By allowing queues to block on writing to a full queue and including online artificial deadlock detection and resolution, bounded PN programs can be completely executed in bounded memory. Geilen and Basten [38] find a flaw in Parks' approach, and show that an algorithm that detects local deadlocks is required for bounded execution. They also argue that bounded scheduling will only yield a complete execution if the PN program is effective, meaning all tokens that are produced are also eventually consumed.

I show that effectiveness is too strong of a restriction; there exist non-effective Process Network programs that will achieve complete execution when using bounded scheduling. This requirement of effectiveness is particularly restrictive for sliding window algorithms, where tokens commonly may remain in queues at termination. I provide counterexamples, and argue that to achieve a complete execution with bounded scheduling, a Process Network must be fair. I define a fair Process Network to mean that all of the processes in the network are fair, and that the scheduler is fair. If this is true, then eventually the network will progress and converge to the unique least fixed point.

Chapter 2 finally covers Computation Graphs [46], which introduce the concept of firing thresholds. A firing threshold allows a node to require the presence of more tokens than it will consume upon firing. Computation Graphs are generally not effective, but can be statically scheduled in bounded memory.

Chapter 3 introduces the concept of deadlock, discusses the necessary conditions for deadlock, and introduces wait-for graphs. It summarizes some previous work in distributed deadlock detection, and then presents the distributed dynamic deadlock detection and resolution (D4R) algorithm. The D4R algorithm is described in detail, including state variables, state transitions, and algorithm proof. Case studies that illustrate the effectiveness of the D4R algorithm are also included. The D4R algorithm is suitable for use in a distributed Process Network implementation, to assist in bounded scheduling so that complete execution can be achieved. If D4R detects a deadlock it will determine whether it is artificial and, if so, identify the queue that must be lengthened to resolve the deadlock. D4R can be used with bounded scheduling of either KPN or CPN.

Chapter 4 proposes the Computational Process Network model. CPN is based on the formalisms of Kahn's PN model, but with enhancements that are designed to make it efficiently implementable. CPN preserves the formal properties of KPN: it is determinate, scalable, and composable. The enhancements of CPN include multi-token transactions to reduce execution overhead, multi-channel queues for multi-dimensional synchronous data, and firing thresholds for queues as both consumers and producers. Firing thresholds help separate computation from communication, and allow memoryless node implementations for sliding window algorithms that are common in digital signal processing systems. Nodes implementing such algorithms can be memoryless because the overlap state is retained on the queues. The zero-copy semantics of CPN reduce the operations required for queue transactions, especially for sliding window algorithms. CPN uses bounded scheduling with the D4R algorithm to achieve complete execution in bounded memory where possible.

The semantics of CPN are: *GetDequeuePtr*, *Dequeue*, *GetEnqueuePtr*, and *Enqueue*. To prove that the formal properties of KPN are preserved, I describe how any CPN program can be transformed into a PN program by adding queues and modifying each process. A CPN program is therefore a set of continuous, monotonic functions of a complete partial order with a least fixed point that corresponds to the behavior of the executing processes [35]. I also provide code that illustrates this transformation from CPN to KPN by implementing the CPN semantics only in terms of Kahn's simple *get* and *put* operations.

KPN and CPN are both are Turing complete, so KPN can eventually compute anything that CPN can compute. However, I show that CPN can compute a particular important class of problems in fewer operations than KPN. Specifically, CPN efficiently models sliding window algorithms that are common in digital signal processing systems.

Table 7.1: Properties of Models of Computation.

	Model			
Property	SDF	CG	KPN	**CPN**
Determinism	✓	✓	✓	✓
Boundedness	✓	✓	*	*
Scalability			✓	✓
Composability			✓	✓
Firing Thresholds		✓		✓
Zero-copy Semantics				✓

Table 7.1 summarizes some desirable properties of the models of computation that have been discussed in this dissertation. The proposed D4R algorithm contributes to the ability to perform bounded scheduling of both KPN and CPN programs. There exists a large and interesting class of programs that are executable to completion with bounded scheduling, although some PN programs are unbounded. In Chapter 2 I argue that the set of PN

programs that can be used with bounded scheduling is larger than what was presented in previous literature [38].

Chapter 5 details the CPN framework, which is a high-performance implementation of the CPN model targeting multi-core and distributed POSIX systems. The CPN framework is released as open source software [77]. Each node in a CPN program corresponds to a POSIX thread. However, the complication of thread programming is not exposed to users of the CPN framework. The formal CPN model underpinnings provide determinacy with concurrent execution.

The CPN interface to queues provides firing thresholds. This allows nodes to operate directly on queue memory, and data presented to nodes is already in a contiguous buffer. This reduces overhead by eliminating the need for nodes to copy and rearrange data, and simplifies the implementation of algorithms that interface to these queues. These queues implement apparent circular addressing by using the virtual memory manager to map the same physical memory pages to multiple virtual addresses.

Chapter 5 also describes how nodes and queues are created within the CPN framework. It describes how a CPN Kernel helps nodes to coordinate with each other to build a CPN program. Nodes connect and communicate using the same interface regardless of whether their peer is local or remote, and nodes can be distributed across multiple compute hosts.

A CPN program can be constructed from a simple JSON [75] based coordination language that describes a part list, netlist, and node mapping. This allows designers to create a distributed parallel system from simple descriptions and dynamically loadable libraries of nodes. Using the CPN framework, developers can build high-performance, high-throughput, distributed systems from deterministic, composable components.

Chapter 6 presents case studies that exercise the CPN framework and demonstrate its capabilities and utility. Benchmarks results are presented from two target platforms. The first is a fairly large SMP system that dual 6-core processors with Hyper-Threading [78]. The second is a cluster of 8 computers connected with InfiniBand [74], each with 4 processors.

The first case study is the Sieve of Eratosthenes [79], an ancient and simple method for finding prime numbers. This prime sieve is of historical interest to Process Networks, as it is the first example Kahn provides of PN performing a useful computation [48]. I first implement the sieve as did Kahn: one prime number being filtered per node, consuming one input at a time. Multi-token firings are an obvious but important enhancement in CPN. With a prime wheel generator, multi-token firings, and multiple prime filters per CPN node, I sped the prime sieve several thousandfold on the large SMP target. The prime sieve also executes on the cluster system.

The second case study is execution of randomly generated graphs, included to demonstrate the robustness and stability of the CPN framework. This case study pseudo-randomly and dynamically builds and modifies CPN program graphs. Queues are randomly created, briefly used for communication, and then deleted. With some probability, nodes are randomly created and randomly expire. On both platforms, this case study runs for more than 72 hours without failure.

The third case study is a circular convolution sonar beamformer and replica correlator, an example of high-throughput signal processing system. This system is implemented with two highly optimized beamformer kernels that leverage OpenMP [8] and the FFTW [66] library. This beamformer has a nominal output data rate of 672 MB/s, and requires about 23.5 GFLOPS to compute its result. This system achieves its real-time goal of 50 ksps on both target platforms, and achieves near-linear speedup on the SMP target.

The Computational Process Network model preserves the formal properties of Process Networks, while reducing the operations required to implement algorithms operating on overlapping continuous streams of data. The CPN framework provides a scalable platform for rapid development of high-throughput computationally intensive algorithms on workstations and workstation clusters.

7.2 Future Work

The CPN model extends Kahn's PN model, making it efficient for sliding window algorithms on high-throughput multidimensional signal processing systems, and with a zero-copy interface. As a model, CPN is fairly feature complete. However there are a number of research topics that could be explored, and probably apply to both KPN and CPN. There are also a number of improvements that could be made to the CPN framework implementation.

7.2.1 Improved D4R

As pointed out by Basten and Hoogerbrugge [2], there are examples of artificial deadlock that can occur without a cycle in the wait-for graph (see Figure 3.12). The current D4R algorithm is based on cycle detection, which is a necessary condition for the classical definition of deadlock. In the presented examples, there is a wait-for chain from an output (sink) node through at least one process that is blocked on a write. I believe that an edge-chasing algorithm similar to D4R could detect this condition. Much of the information needed to detect this condition is already included in the state variables maintained and exchanged by the D4R algorithm. However, resolution of such an artificial deadlock must be a heuristic and could lead to unnecessary queue growth. An improved D4R algorithm could apply to both CPN and KPN.

7.2.2 Distributed Scheduling and Node Migration

Load balancing on an SMP machine is fairly automatic. However, a lack of load balancing in a distributed system leads to a loss of performance. It would be possible to extend the CPN framework to permit node migration across compute hosts in a distributed target. A distributed scheduler could then examine the load across the distributed system and automatically migrate nodes for load balancing. Some work has been done in this area [101] with a focus on KPN. Node migration may be an expensive operation, so it would be important to prevent inappropriate or repeating migrations.

7.2.3 Remote DMA Queues

Remote direct memory access (RDMA) is a method for transferring information from one computer's memory to another without involving the operating system, and can reduce overhead for high-throughput communication. This seems to be a good fit for queues in the CPN framework, and could lead to improved scalability for distributed targets. RDMA is available over IP networks in the form of iWARP [102], and also available over InfiniBand.

7.2.4 Node Clustering

As discussed in Chapter 5, the CPN framework works best with coarse-grain parallelism. If design automation tools were used to build CPN graphs, it would be useful to permit the capture of fine-grain parallelism that could be clustered together for reduced dynamic scheduling overhead. This area has been explored in some depth by the Ptolemy Project [34], for example. Clusters generated from a tool like Ptolemy could be embedded in a CPN node, or CPN could be used as a high-level coordinator from Ptolemy.

7.2.5 Distributed Framework State

A distributed CPN system currently uses a single `RemoteContext` daemon to maintain the description of the entire graph. Whenever a node or a queue is created or destroyed, there must be an interaction with this daemon.

This simplifies the design, but could become a bottleneck and limit scalability. At some level a CPN Context is similar to a database, so it may be possible to leverage research in the area of distributed databases.

7.2.6 Fault Tolerance

As currently implemented, the CPN framework is fault intolerant. If a specified compute node is missing or crashes, the whole system may be adversely affected. This issue is ignored in Kahn's formal model, but is an important consideration for implementation of useful system development tools.

7.2.7 Additional Targets and Applications

Although I have targeted Linux workstations in this dissertation, PN and CPN can be applied to a large variety of targets and applications. Nearly anything that can carry a reliable stream of data could be made into a queue, and nearly anything that can transform data from one form to another could be represented as a process. For applications that require a specialized implementation component (such as custom high-speed logic), it would be easy to integrate a gateway to that component with proxy CPN nodes or queues. PNs have been targeted to programmable logic chips [103], and it is easy to image a CPN node that could offload massive calculations with CUDA [104].

The CPN model and framework could be used in a large variety of high-throughput multi-channel signal and image processing applications, including sonar, radar, seismology, video processing, and communications.

Bibliography

[1] W. Huang and D. Qi, “A local deadlock detection and resolution algorithm for process networks,” in *Proc. Int. Conf. on Computer Science and Software Engineering*, 2008, pp. 311–314. [Online]. Available: http://dx.doi.org/10.1109/CSSE.2008.1468

[2] T. Basten and J. Hoogerbrugge, “Efficient execution of process networks,” in *Proc. Communicating Process Architectures*, Bristol, UK, Sep. 2001, pp. 1–14. [Online]. Available: http://www.es.ele.tue.nl/~tbasten/papers/eepn.pdf

[3] E. A. Lee, “The problem with threads,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-1, Jan. 10 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

[4] J. Ousterhout, “Why threads are a bad idea (for most purposes),” in *USENIX 1996 Annual Technical Conf.*, San Diego, CA, Jan. 1996. [Online]. Available: http://home.pacbell.net/ouster/threads.pdf

[5] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner, “Symmetric multiprocessing in Solaris 2.0,”

in *Proc. IEEE Computer Society Int. Conf., Digest of Papers.*, Feb 1992, pp. 181–186.

[6] *IEEE 1003.1-2001*, IEEE Computer Society Portable Application Standards Committee (PASC) Std. [Online]. Available: http://www.pasc.org/

[7] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[8] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP.* Morgan Kaufmann, 2000.

[9] A. Barak and O. La'adan, "The MOSIX multicomputer operating system for high performance cluster computing," *Journal of Future Generation Computer Systems*, vol. 13, pp. 4–5, 1998.

[10] A. Barak and A. Shiloh, "The MOSIX project website," http://mosix.org.

[11] "OpenSSI clusters for Linux," http://openssi.org.

[12] M. Sharifi and K. Karimi, "DIPC: The Linux way of distributed programming," *Linux J.*, 1999.

[13] "The DIPC project," http://flash.lakeheadu.ca/~kkarimi/dipc.html.

[14] "The LinuxPMI project," http://linuxpmi.org.

[15] B. Nitzberg and V. Lo, “Distributed shared memory: a survey of issues and algorithms,” *Computer*, vol. 24, no. 8, pp. 52–60, Aug 1991.

[16] M. T. Jones, *BSD Sockets Programming from a Multi-Language Perspective.* Rockland, MA, USA: Charles River Media, Inc., 2003.

[17] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce.* Morgan and Claypool Publishers, 2010.

[18] “The message passing interface (MPI) standard,” http://www.mcs.anl.gov/research/projects/mpi/, [Online; accessed 27-January-2010].

[19] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, “MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes,” in *Proc. ACM/IEEE Conf. on Supercomputing*, Los Alamitos, CA, USA, 2002, pp. 1–18.

[20] G. Burns, R. Daoud, and J. Vaigl, “LAM: An open cluster environment for MPI,” in *Proc. Supercomputing Symposium*, 1994, pp. 379–386. [Online]. Available: http://www.lam-mpi.org/download/files/lam-papers.tar.gz

[21] J. M. Squyres and A. Lumsdaine, “A component architecture for LAM/MPI,” in *Proc. European PVM/MPI Users’ Group Meeting*, no. 2840, Venice, Italy, Sept. 2003, pp. 379–387.

[22] J. Squyres and B. Barrett, "Open MPI community meeting," in *Proc. ACM/IEEE Conf. on Supercomputing*, New York, NY, USA, 2006, p. 5.

[23] P. S. Pacheco, *Parallel programming with MPI.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, p. 7.

[24] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 2nd ed. Morgan Kaufmann Publishers, 1998, p. 715.

[25] D. Lea, *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[26] J. Corbet, "The big kernel lock strikes again," http://lwn.net/Articles/281938/, 2008, [Online; accessed 24-February-2010].

[27] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal, "Debugging race conditions in message-passing programs," in *Proc. SIGMETRICS Symposium on Parallel and Distributed Tools*, New York, NY, USA, 1996, pp. 31–40.

[28] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proc. Int. Joint Conf. on Artificial Intelligence*, San Francisco, CA, USA, 1973, pp. 235–245.

[29] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[30] A. Turjan, B. Kienhuis, and E. Deprettere, “Classifying interprocess communication in process network representation of nested-loop programs,” *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 2, p. 13, 2007.

[31] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf, “YAPI: Application modeling for signal processing systems,” in *Proc. Design Automation Conf.* ACM Press, 2000, pp. 402–405.

[32] B. Kienhuis, E. Rijpkema, and E. Deprettere, “Compaan: deriving process networks from Matlab for embedded signal processing architectures,” in *Proc. Int. Workshop on Hardware/Software Codesign*, May 2000, pp. 13 –17.

[33] H. A. Andrade and S. Kovner, “Software synthesis from dataflow models for G and LabVIEW,” in *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, 1998, pp. 1705–1709.

[34] “The Ptolemy project,” http://ptolemy.eecs.berkeley.edu/, [Online; accessed 17-March-2011].

[35] G. Kahn, “The semantics of a simple language for parallel programming,” *Information Processing*, pp. 471–475, Aug. 1974.

[36] T. M. Parks, “Bounded scheduling of process networks,” Ph.D. dissertation, EECS Department, University of California, Berkeley, CA 94720-1770, Dec. 1995, Technical Report UCB/ERL-95-105.

[Online]. Available: http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/

[37] E. Lee and T. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995. [Online]. Available: http://ptolemy.eecs.berkeley.edu/papers/95/processNets/

[38] M. Geilen and T. Basten, "Requirements on the execution of Kahn process networks," in *Proc. European Symposium on Programming*, 2003, pp. 319–334. [Online]. Available: http://citeseer.ist.psu.edu/geilen03requirements.html

[39] G. E. Allen, P. Zucknick, and B. L. Evans, "A distributed deadlock detection and resolution algorithm for process networks," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Proc.*, Honolulu, HI, Apr. 2007. [Online]. Available: http://users.ece.utexas.edu/~bevans/papers/2007/distributedPN/distributedPNICASSP2007Paper.pdf

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

[41] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[42] ——, "Static scheduling of synchronous dataflow programs for digital

signal processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24–35, Jan. 1987.

[43] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs.* Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[44] S. Edwards, *Languages for Digital Embedded Systems.* Kluwer Academic Publishers, 2000.

[45] J. L. Pino and K. Kalbasi, "Cosimulating synchronous DSP applications with analog RF circuits," in *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, Nov. 1998, pp. 1710–1714. [Online]. Available: http://ptolemy.eecs.berkeley.edu/publications/papers/98/asilomar98/

[46] 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.

[47] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans, "Trends in multicore DSP platforms," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 38–49, Nov. 2009.

[48] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," *Information Processing*, pp. 993–998, Aug. 1977.

[49] B. Stroustrup, *The C++ Programming Language.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[50] S. C. Kleene, *Introduction to Metamathematics.* New York, NY, USA: D. Van Nostrand Co., 1952.

[51] A. A. Faustini, "An operational semantics for pure dataflow," in *Proc. Colloquium on Automata, Languages and Programming.* London, UK: Springer-Verlag, 1982, pp. 212–224.

[52] S. Brookes, "On the Kahn principle and fair networks," in *Proc. Conf. Mathematical Foundations of Programming Semantics*, May 1998.

[53] W. W. Wadge, "An extensional treatment of dataflow deadlock," *Theoretical Computer Science*, vol. 13, no. 1, pp. 3 – 15, 1981.

[54] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts; 7th Ed.* Hoboken, NJ, USA: Wiley, 2004.

[55] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed. Cambridge University Press, May 2008.

[56] M. Singhal, "Deadlock detection in distributed systems," *IEEE Computer*, vol. 22, no. 11, pp. 37–48, 1989.

[57] A. D. Kshemkalyani and M. Singhal, "A one-phase algorithm to detect distributed deadlocks in replicated databases," *IEEE Trans. on Knowl. and Data Eng.*, vol. 11, no. 6, pp. 880–895, 1999.

[58] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, 1983.

[59] M. Prieto, J. Villadangos, F. Farina, and A. Cordoba, "An o(n) distributed deadlock resolution algorithm," in *Proc. Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 48–55.

[60] A. G. Olson and B. L. Evans, "Deadlock detection for distributed process networks," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Proc.*, Philadelphia, PA, Mar. 2005, pp. 73–76.

[61] S. Lee and J. Kim, "Performance analysis of distributed deadlock detection algorithms," *IEEE Trans. Knowledge and Data Engineering*, vol. 13, no. 4, pp. 623 –636, Jul. 2001.

[62] "Gregory Allen's home page," https://webspace.utexas.edu/gallen/.

[63] P. Druschel and L. L. Peterson, "Fbufs: a high-bandwidth cross-domain transfer facility," *SIGOPS Oper. Syst. Rev.*, vol. 27, pp. 189–202, Dec. 1993.

[64] Y. A. Khalidi and M. N. Thadani, "An efficient zero-copy I/O framework for UNIX," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 1995.

[65] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[66] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[67] P. Plauger, A. Stepanov, M. Lee, and D. R. Musser, *The C++ Standard Template Library.* Prentice Hall, 2000.

[68] L. John, V. Reddy, P. Hulina, and L. Coraor, "A comparative evaluation of software techniques to hide memory latency," in *Proc. Hawaii Int. Conf. on System Sciences*, 1995, pp. 229–238.

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

[70] "GNU lesser general public license," http://www.gnu.org/licenses/lgpl.html, [Online; accessed 27-February-2011].

[71] "CLOC count lines of code," http://cloc.sourceforge.net/, [Online; accessed 27-February-2011].

[72] "The pthreads-win32 project," http://sourceware.org/pthreads-win32/, [Online; accessed 27-February-2011].

[73] G. Allen, P. Zucknick, and B. Evans, "Zero-copy queues for native signal processing using the virtual memory system," in *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, Nov. 2006, pp. 1188–1191.

[74] “Infiniband trade association,” http://www.infinibandta.org/, [Online; accessed 27-February-2011].

[75] “JSON (JavaScript Object Notation),” http://www.json.org/, [Online; accessed 27-February-2011].

[76] E. Ray, *Learning XML.* O’Reilly Media, 2001.

[77] “Gregory Allen’s CPN page,” https://webspace.utexas.edu/gallen/CPN/, [Online; accessed 27-February-2011].

[78] Intel, “Intel hyper-threading technology,” http://www.intel.com/info/hyperthreading/, [Online; accessed 5-March-2011].

[79] J. Conway and R. Guy, *The Book of Numbers.* Springer, 1995.

[80] Wikipedia, “Eratosthenes,” http://en.wikipedia.org/wiki/Eratosthenes, [Online; accessed 5-March-2011].

[81] M. D. McIlroy, “Coroutines,” Bell Telephone Laboratories, Murray Hill, NJ, Internal Report, May 1968.

[82] HaskellWiki, “Prime wheels,” http://www.haskell.org/haskellwiki/Prime_numbers#Prime_Wheels, [Online; accessed 5-March-2011].

[83] D. Shanks, *Solved and unsolved problems in number theory.* New York, USA: Chelsea Publishing Co., Inc., 1985.

[84] H. Dubner, “Factorial and primorial primes,” *J. Recr. Math.*, vol. 19, pp. 197–203, 1987.

[85] S. W. Golomb, *Shift Register Sequences.* Laguna Hills, CA, USA: Aegean Park Press, 1981.

[86] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull, “Graphviz and dynagraph – static and dynamic graph drawing tools,” in *Graph Drawing Software*, M. Junger and P. Mutzel, Eds. Springer-Verlag, 2004, pp. 127–148. [Online]. Available: http://www.springer.com/math/cse/book/978-3-540-00881-1

[87] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2007, pp. 89–100.

[88] “Valgrind,” http://valgrind.org/, [Online; accessed 17-March-2011].

[89] N. Nethercote and J. Seward, “How to shadow every byte of memory used by a program,” in *Proc. of the Int. Conf. on Virtual Execution Environments*, 2007, pp. 65–74.

[90] R. J. Urick, *Principles of Underwater Sound.* New York, NY: McGraw-Hill Book Company, 1975.

[91] R. G. Pridham and R. A. Mucci, “A novel approach to digital beamforming,” *Journal Acoustical Society of America*, vol. 63, no. 2, pp. 425–434, Feb. 1978.

[92] J. F. Bridgman, G. E. Allen, and B. L. Evans, "Multi-core sonar beamforming with computational process networks," in *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, Nov. 2010, pp. 1254–1258.

[93] S.-H. Yu and J.-S. Hu, "Optimal synthesis of a fractional delay FIR filter in a reproducing kernel Hilbert space," *IEEE Signal Processing Letters*, vol. 8, no. 6, pp. 160 –162, Jun. 2001.

[94] D. R. Farrier, T. S. Durrani, and J. M. Nightingale, "Fast beamforming techniques for circular arrays," *The Journal of the Acoustical Society of America*, vol. 58, no. 4, pp. 920–922, 1975.

[95] T. L. Henderson, "Wide-band monopulse sonar: Processor performance in the remote profiling application," *IEEE Journal of Oceanic Engineering*, vol. 12, no. 1, pp. 182–197, 1987.

[96] A. J. C. Bik, *The Software Vectorization Handbook: Applying Intel Multimedia Extensions for Maximum Performance.* Intel Press, 2004.

[97] G. E. Allen, B. L. Evans, and L. K. John, "Real-time high-throughput sonar beamforming kernels using native signal processing and memory latency hiding techniques," in *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, 1999, pp. 25–28.

[98] S. Tucker, R. Vienneau, J. Corner, and R. Linderman, "Swathbuckler: HPC processing and information exploitation," in *IEEE Conf. on Radar*, April 2006, pp. 710–717.

[99] "HPEC challenge: Corner turn benchmark," http://www.ll.mit.edu/HPECchallenge/ct.html, [Online; accessed 13-March-2011].

[100] "Intel Xeon microprocessor export compliance metrics," http://www.intel.com/support/processors/xeon/sb/CS-020863.htm, [Online; accessed 16-March-2011].

[101] Z. Qian, M. Zeng, D. Qi, and K. Xu, "A dynamic scheduling algorithm for distributed Kahn Process Networks in a cluster environment," *IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, vol. 2, pp. 36–42, 2008.

[102] Borkar, Cohn, Cox, Gleason, Gross, Kung, Lam, Moore, Peterson, Pieper, Rankin, Tseng, Sutton, Urbanski, and Webb, "iWarp: an integrated solution to high-speed parallel computing," *Supercomputing Conference*, vol. 1, pp. 330–339, 1988.

[103] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn Process Networks: The Compaan/Laura approach," in *Proc. Conf. on Design, Automation and Test in Europe*, 2004, pp. 10 340–10 348. [Online]. Available: http://portal.acm.org/citation.cfm?id=968878.968962

[104] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 2010.

Vita

Gregory Eugene Allen was born to Gordon Eugene and D'Maris Anne Allen on October 30, 1968, in Austin, Texas. He attended Austin's John H. Reagan High School, and graduated salutatorian in 1987. In 1991, Greg received his Bachelor of Science in Electrical Engineering from The University of Texas at Austin, graduating with Highest Honors. In 1993, he returned to The University as a part-time graduate student. In 1998, he earned his Master of Science in Electrical Engineering. Greg has been employed with Applied Research Laboratories (ARL) at UT's J. J. Pickle Research Campus since the summer of 1986, when he was hired through the High School Apprentice Program. Since 1988 he has worked on high-frequency, high-resolution sonar systems in the Sonar Development Division of the Advanced Technology Laboratory at ARL. In addition to design engineering, Greg has been involved in system-level testing, installation, and deployment aboard U.S. Navy submarines, including a surfacing at the North Pole in 1993. Greg lives with his wife and their 3 children in northwest Austin. Greg's daughter Sabrina, a 2002 victim of international parental child abduction, is still missing.

Permanent address: 9612 Slate Creek Trail
Austin, Texas 78717

This dissertation was typeset with LaTeX by the author.