Computational Process Networks

A model and framework for high-throughput signal and image processing systems

Greg Allen <gallen@arlut.utexas.edu>
6 April 2010
ESPL Group Meeting

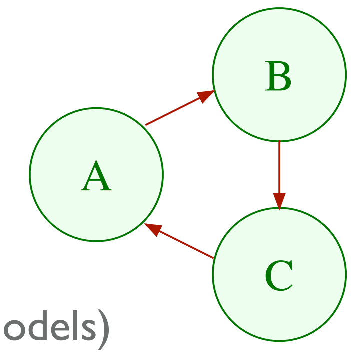
ESPL Embedded Signal Processing Laboratory
The University of Texas at Austin

ARL
The University of Texas at Austin

Introduction

- Many embedded systems require concurrent implementations
- High-throughput, high-performance applications
  - Sonar beamforming
  - Synthetic Aperture Radar (SAR) image processing
- Traditionally implemented in custom hardware or custom integration of embedded processors
- Commercial workstations/clusters can be viable platforms
  - Multi-core (SMP) computing
  - Distributed (cluster) computing
- Using commodity platforms saves significant time and money

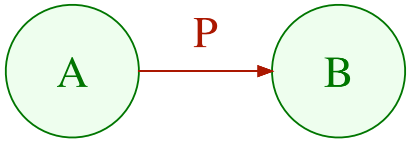
Parallel Programming



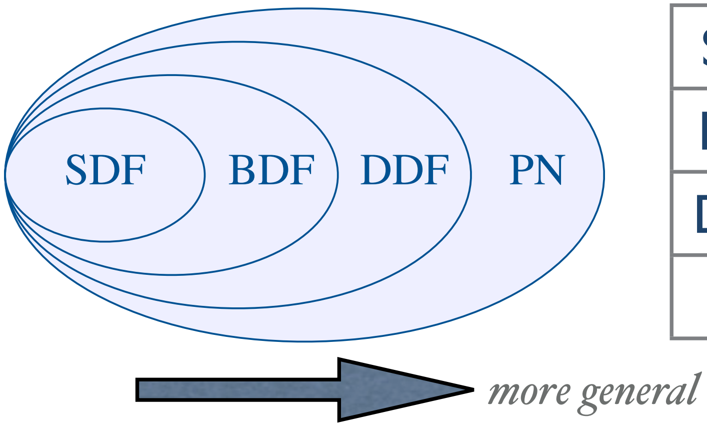
- *Problem:* Effective parallel programming is difficult
  - Hard to predict and prevent deadlock
  - Hard to achieve determinate execution
  - Hard to make scalable software (e.g. rendezvous models)
- Current approaches typically lack formal underpinnings
  - Threads are "wildly nondeterministic" [Lee 2006]
  - MPI is the "assembly language" of cluster computers
- *Solution:* Formal models for concurrent systems

Dataflow Models

- Programs are modeled as a directed graph
  - Each node represents a computational unit
  - Each edge represents a one-way FIFO queue



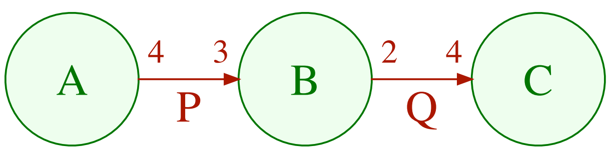
- A node may have any number of input and output edges, and may communicate *only* via these edges
- Models functional & data parallelism in systems



SDF	Synchronous Dataflow (Agilent ADS)
BDF	Boolean Dataflow
DDF	Dynamic Dataflow
PN	Process Networks (NI LabVIEW)

Synchronous Dataflow (SDF)

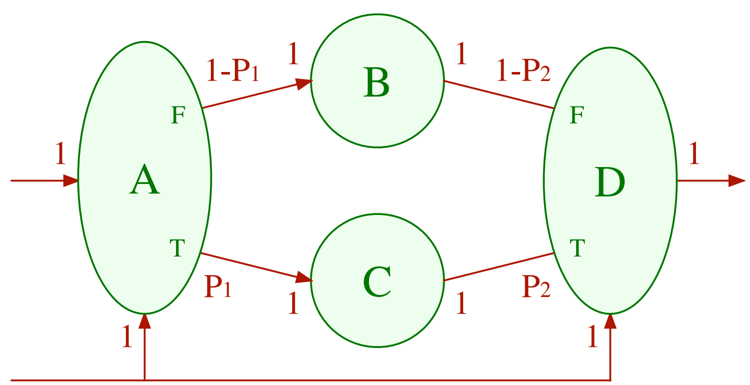
- Firing behavior of each node is known and static
- Termination and boundedness are decidable [Lee, 1986]
  - Flow of control and memory usage can be compiled
  - Schedule constructed once and repeatedly executed
- Well-suited for synchronous multi-rate signal processing
- Used in design automation tools [HP, Cadence]



Schedule	Memory
AAABBBBCC	12 + 8
ABABCABBC	6 + 4

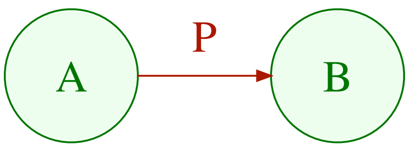
Boolean Dataflow (BDF)

- Turing complete
- Adds switch & select, which give if/then/else
- Termination and boundedness are undecidable
- Quasi-static scheduling with clustering of SDF



Process Networks (PN)

- A *networked set* of Turing machines
- Mathematically provable properties [Kahn, 1974]
    - Guarantees determinate execution of program
    - Allows concurrent execution of nodes



- Dynamic firing rules at each node:
    - *Blocking reads:* suspend a node's execution when it attempts to consume data from an empty queue
    - *Non-blocking writes:* never suspend a node for producing data (so queues can grow without bound)

Bounded Scheduling of PN

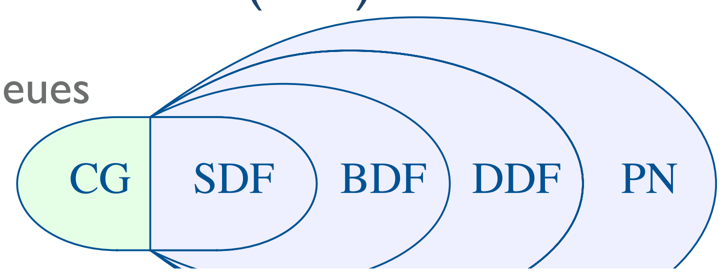
- Kahn's original PN model assumes infinite memory!
- Clever dynamic scheduling of the nodes allows execution in bounded memory, if it is possible [Parks 95]
    - May introduce *artificial deadlock* due to queue bounds
    - Dynamic deadlock detection & resolution required
    - Lengthen shortest deadlocked full queue to resolve

	Parks '95	Geilen & Basten '03
Deadlock detector	Global deadlocks	Local deadlocks
Preserves PN properties	No (counterexamples)	Yes, if an *effective* PN

- Detailed deadlock detection algorithms were not provided

Computation Graphs (CG)

- Each FIFO queue has static parameters [Karp & Miller, 1966]
  - A - number of tokens initially present
  - U - number of tokens inserted by producer at each firing
  - W - number of tokens removed by consumer at each firing
  - T - number of tokens in queue before consumer can fire (T≥W)
- Termination and boundedness are decidable
  - Statically scheduled, iterative scheduling algorithms
- Slightly more general than Synchronous Dataflow (SDF)
  - SDF is special case where T = W for all queues



Computational Process Networks (CPN)

- New model for high-performance parallel computation
  - Formal underpinnings, but implementable, scalable, and efficient
- Begin with the Process Network model [Kahn, 1974]
  - Provides formal determinism with parallel/distributed execution
- Utilize bounded scheduling and distributed deadlock detection and resolution (our D4R algorithm published in ICASSP 07)
  - Permits execution in finite memory where possible
- Include extensions to aid performance:
  - Multi-token transactions to reduce framework overhead
  - Multi-channel queues for multi-dimensional synchronous data
  - *Firing thresholds* from Computation Graphs [Karp & Miller, 1966]

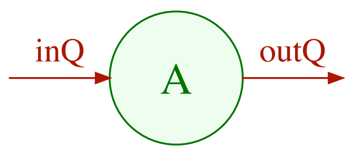
LabVIEW's "G" language uses single-token transactions

Firing Thresholds

- A node can access more tokens than it will discard
  - Models algorithms on overlapping continuous streams of data, which are very common in DSP, e.g. digital filters, overlap-and-save FFTs
  - Allows overlapping input streams without data copies
- A node can access more free space than it will fill (the dual)
  - Allows variable-rate outputs without data copies
- Decouples computation from communication
- Permits a zero-copy queue implementation
  - Nodes can operate directly from/to queue memory
  - Frees the CPU for computation tasks instead of copying
  - CPUs are fast, memory is relatively slow
  - Moving data is expensive, often the limiting factor for performance

A Sample CPN Node

Frequency domain FIR filter using overlap-save 1024 FFT




```
// CPN code
typedef complex<float> T;
T filter[1024];
while (true) {
    // blocking calls to get in/out data pointers
    const T* inPtr = inQ.GetDequeuePtr(1024);
    T*       outPtr = outQ.GetEnqueuePtr(1024);

    // do the math
    fft(inPtr, outPtr, 1024);
    cpx_multiply(filter, outPtr, outPtr, 1024);
    ifft(outPtr, outPtr, 1024);

    // complete the node transactions
    inQ.Dequeue(512);
    outQ.Enqueue(512);
}
```

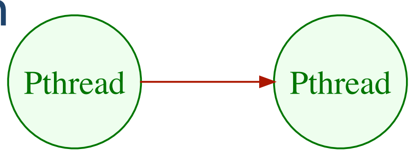
```
// PN code
typedef complex<float> T;
T filter[1024];
T tmpData[1024];
while (true) {
    // do overlap-save, get new data
    memcpy(tmpData, tmpData+512, 512*sizeof(T));
    inQ.get(tmpData+512, 512);

    // do the math
    fft(tmpData, tmpData, 1024);
    cpx_multiply(filter, tmpData, tmpData, 1024);
    ifft(tmpData, tmpData, 1024);

    // copy out the results
    outQ.put(tmpData, 512);
}
```

CPN Implementation

- C++ with POSIX Pthreads
- Built-in distributed deadlock detection algorithm
- A CPN node maps readily onto a single thread



  - Node granularity (execution time) should be larger than a Pthread context switch (~10 μs)
  - Increasing node granularity reduces scheduling overhead
- SMP OS dynamically schedules nodes as data flow permits
- For distributed implementations, CPN Nodes are statically assigned to hosts, and do not migrate

CPN Kernel

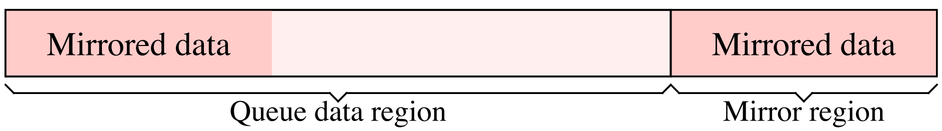
- Runs as "main" thread of a process, typically one per host
- Contains tables which describe the CPN program
  - Table of hosts in entire CPN program (and how to contact them)
  - Table of CPN nodes, and how they are mapped to hosts
  - Table of CPN queues, and to which nodes they connect
- CPN nodes are threads created by the kernel
  - Dynamic linking to support addition of node types
- CPN nodes ask the kernel for connection to a queue
  - Kernel matches up ends of queues
  - Details about queues are hidden from nodes

CPN Implementation

- CPN Queues connect between nodes
- Queue type determined by kernel based on node locations
  - Zero-copy queues where possible (shared memory)
    - Between nodes in the same process on the same host
    - Highest performance, lowest overhead
  - Queues over TCP for distributed systems
    - Supports Ethernet and Infiniband
    - Other connection mediums easily supported

Zero-copy Queues

- Queues use thresholds to allow zero-copy operation
  - Nodes operate directly on queue memory to avoid undesired copies
  - Overlapping-window algorithms are efficient
- Queues use mirroring to keep data contiguous



  - Same idea as modulo addressing in DSPs -- circular buffers
  - The virtual memory manager maintains data circularity with hardware
- Presented at Asilomar 2006

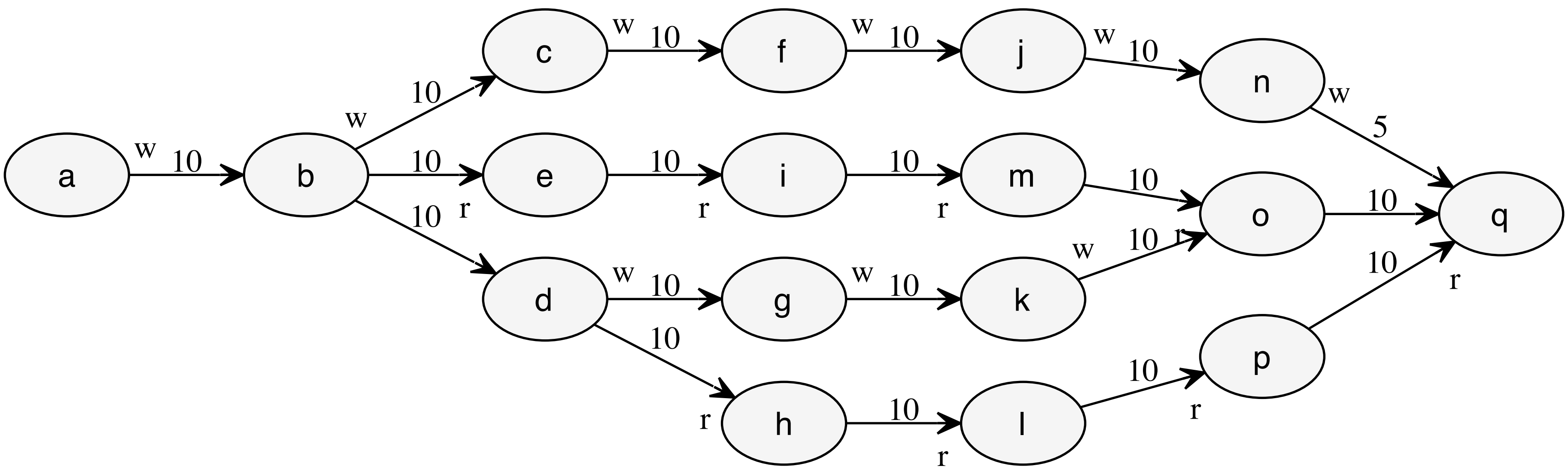
Dynamic Operations

- CPN Nodes can, with the help of the kernel:
  - Create new CPN Nodes
  - Create new CPN Queues
  - Attach to CPN Queues
  - Unattach from CPN Queues (marking for deletion)
- CPN Nodes can terminate themselves (but not others)

Case Studies Underway

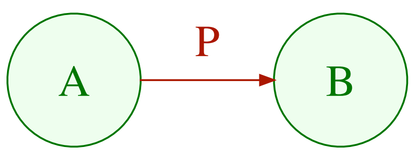
- #0 - Distributed Dynamic Deadlock Detection and Resolution
  - Testing and verification of D4R algorithm
  - For a given program, find and resolve artificial deadlock
  - Addressing some examples from literature
  - Bug found and addressed since ICASSP 07
  - Examining some “difficult” examples with no cycles

D4R Sample Program

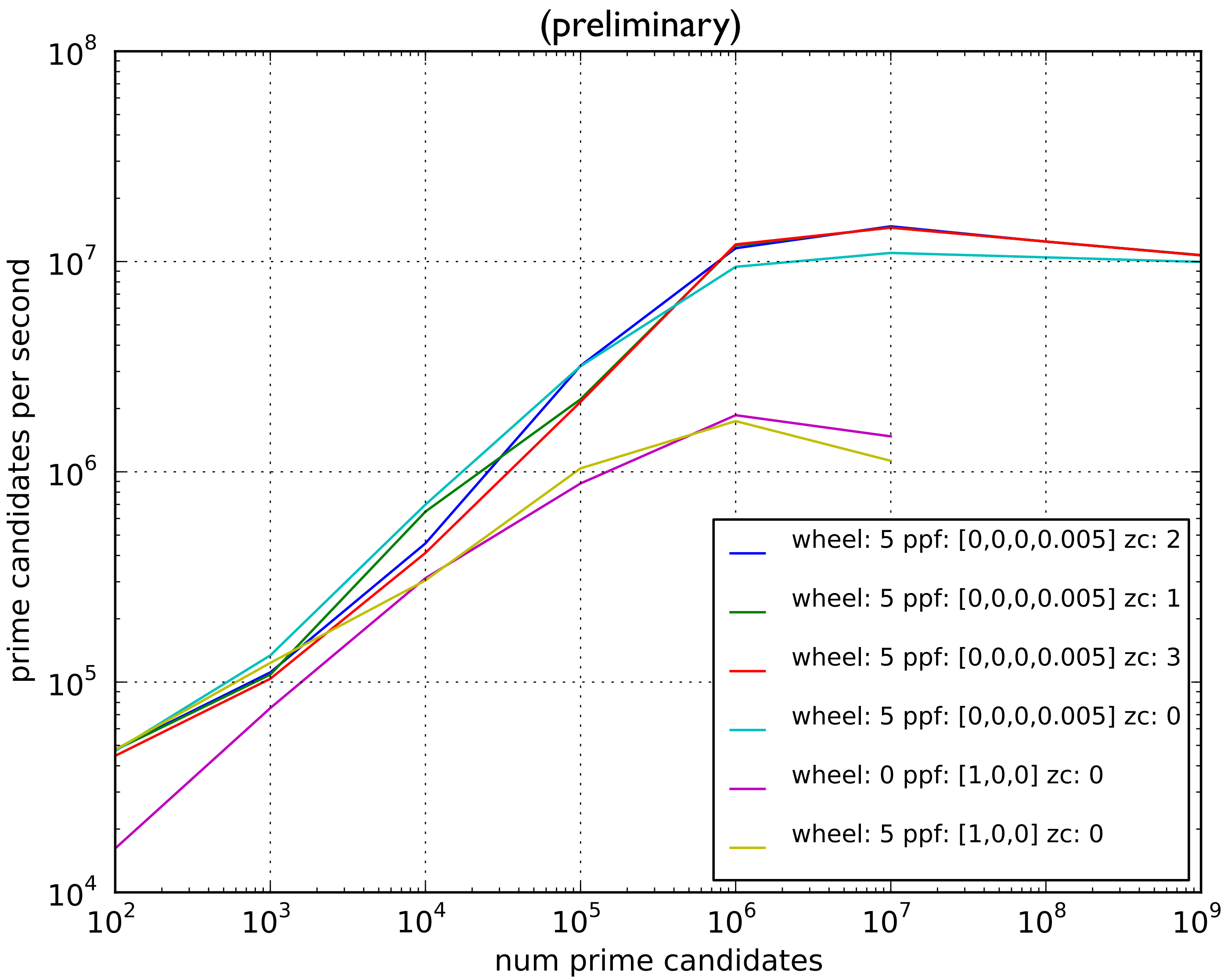


Case Studies Underway

- #1 - Sieve of Eratosthenes (Prime Number Sieve)
  - Base node generates fountain of prime candidate numbers
  - Subsequent nodes sieve out some number of primes
  - Nodes and queues are recursively generated as prime candidates pass final existing node
  - Classic PN example, used in Kahn[77] and Park's thesis
  - Demonstrates dynamic node and queue creation



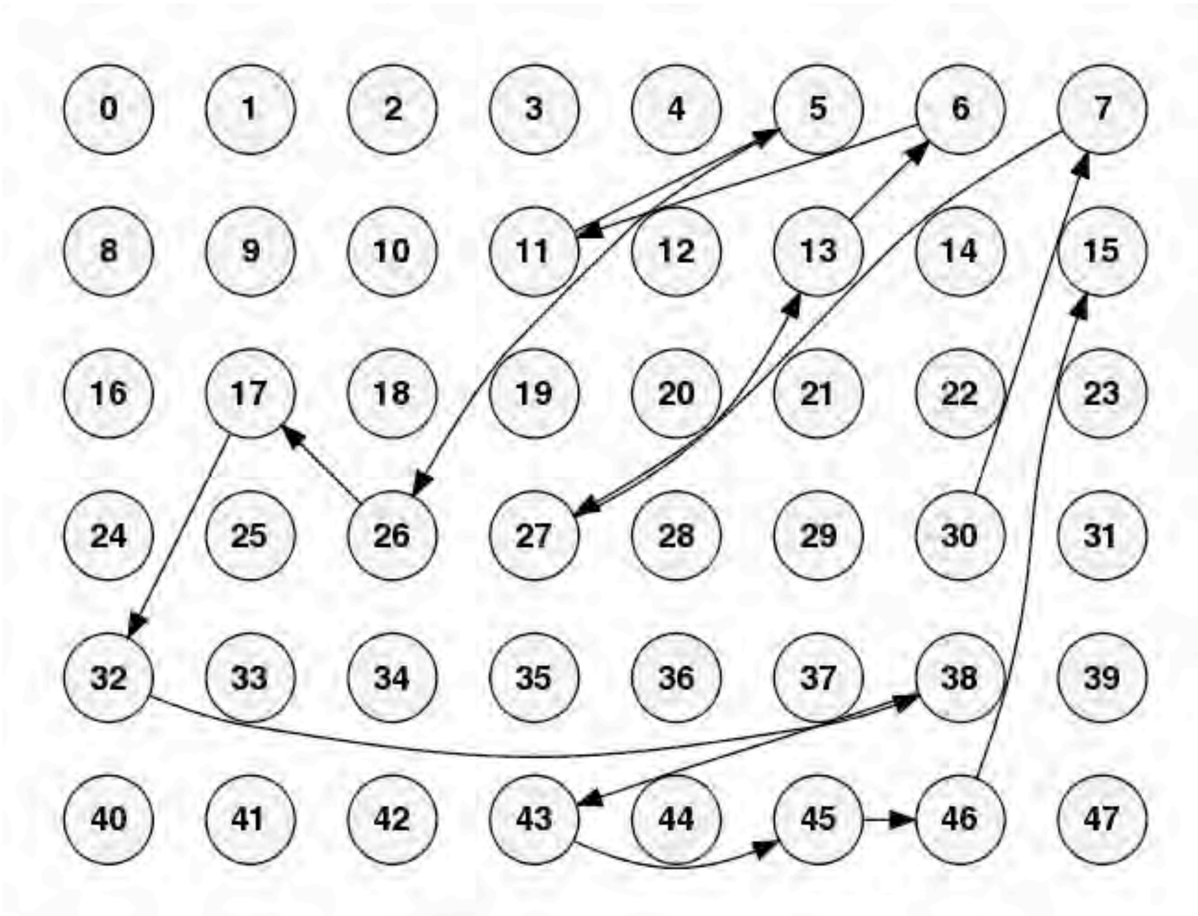
Prime Sieve Results



Case Studies Underway

- #2 - Randomized Connections
  - Some number of nodes are created
  - A pseudo-random number generator gives “instructions” to make random connections between nodes
  - Chains of nodes connect, exchange data, and disconnect
  - Nodes can also randomly terminate or be created
  - Demonstrates dynamic queue connection and disconnect, and random node creation and termination

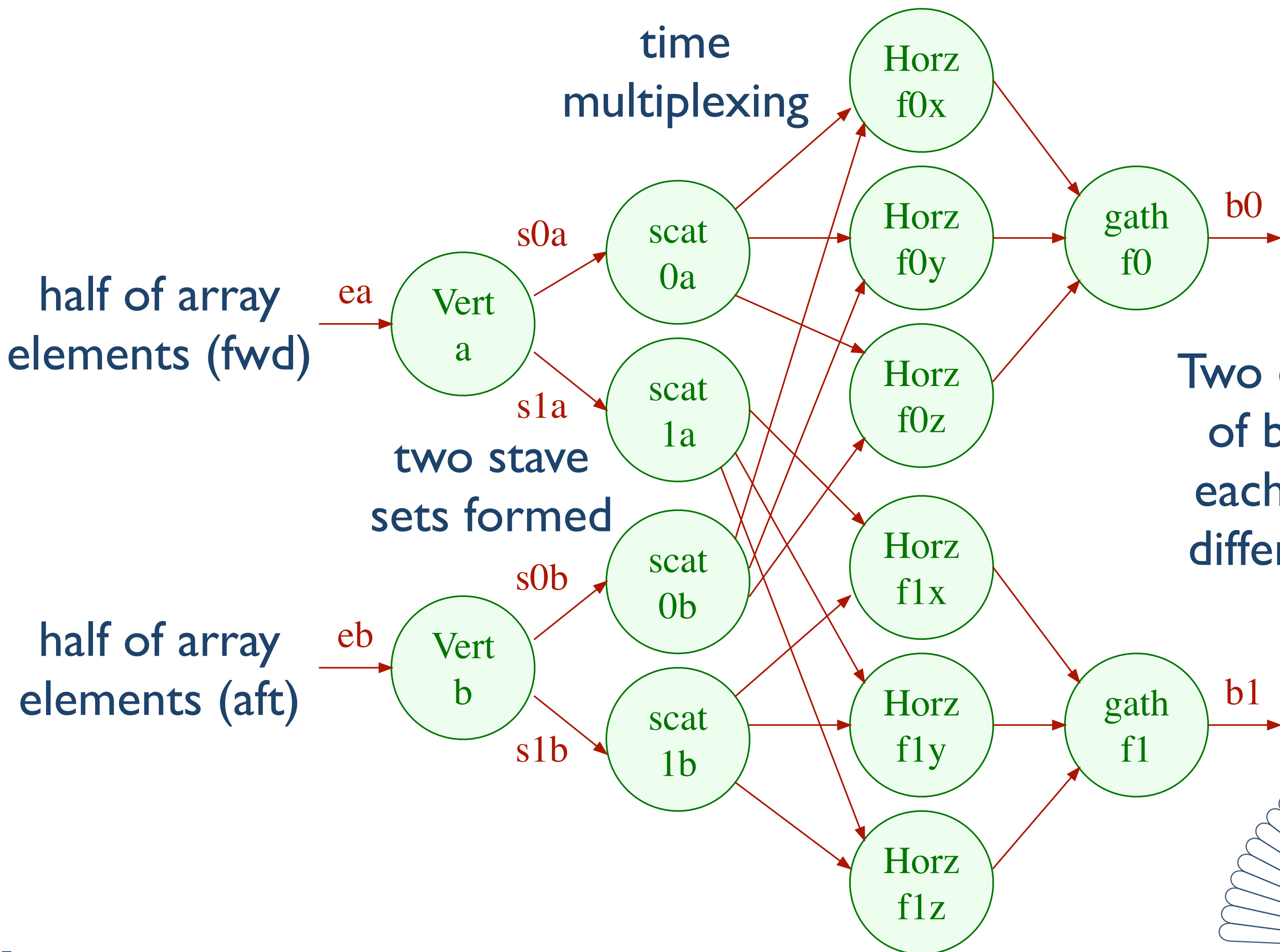
Case Studies Underway



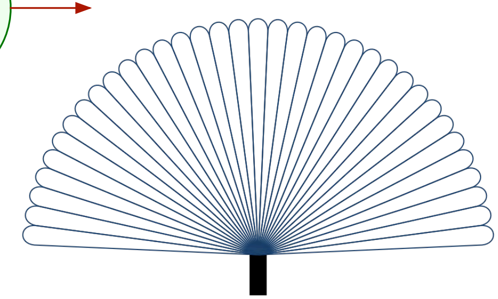
Case Studies Underway

- #3 - 3D Circular Convolution Beamformer and Correlator
  - Feed-forward network with functional and data parallelism
  - CPN is utilized for it parallel, distributed nature
  - Vertical beamforming from element data
  - Horizontal circular convolution beamforming and replica correlation
  - Approximately 1GB/s input data, ~50 GFLOPS
  - Demonstrates high performance and high throughput of distributed CPN framework

3D Beamformer Diagram



Two different "fans" of beam outputs, each formed from different stave sets



Conclusion

- Computational Process Network, a model for high-performance and high-throughput parallel signal processing
  - Determinate (and bounded) execution with concurrency
  - Firing thresholds for zero-copy interfaces
  - Multi-token firings, and multi-channel queues
- Framework implementation and case studies underway
- Heterogeneous targets of a wide range are possible
- Programs built from deterministic, composable components