Computational Process Networks

a model and framework for high-throughput signal processing

Gregory E. Allen

Ph.D. Defense

25 April 2011

ARL
The University of Texas at Austin

Embedded Signal Processing Laboratory
ESPL
The University of Texas at Austin

Wireless Networking & Communications Group

Committee Members:

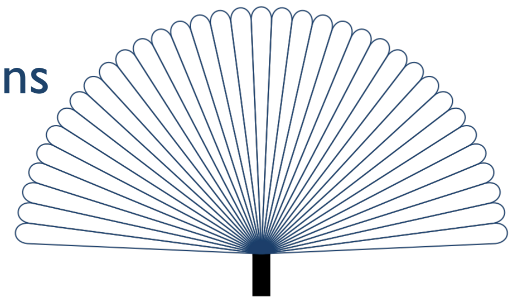
James C. Browne
Craig M. Chase
Brian L. Evans (Advisor)
Lizy K. John
Charles M. Loeffler

Outline

- Need for speed
- Dataflow models
- Contributions
  - Dynamic Distributed Deadlock Detection & Resolution (D4R)
  - The Computational Process Network (CPN) model
  - CPN framework implementation & case studies
- Conclusion

Introduction

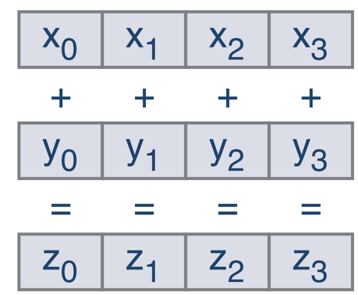
- High-throughput, high-performance applications
  - Sonar beamforming (100s of MB/s, 10s of GFLOPS)
  - Synthetic Aperture Radar (SAR) processing
- Traditional embedded concurrent implementations
  - Custom hardware
  - Custom integration of embedded processors



- Commercial workstations and clusters
  - Multi-core symmetric multiprocessing (SMP) computing
  - Distributed (cluster) computing, high-speed interconnect
  - Significant savings in design time

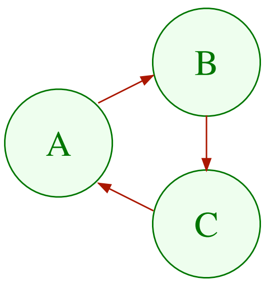
High Performance

- Single-task approaches
  - Single Instruction Multiple Data (SIMD) for data parallelism
  - Hand-optimized signal processing kernels and libraries
  - Memory latency hiding to reduce input/output bottleneck
  - Lock memory buffers to avoid swapping to disk
  - Fixed-priority real-time scheduling



- Executing tasks efficiently on parallel hardware
  - **Scalability:** more parallel hardware gives more performance
  - **Determinate:** always gets same answer (no race conditions)
  - **Locking:** prevent concurrent access to shared resources

Concurrent Programming



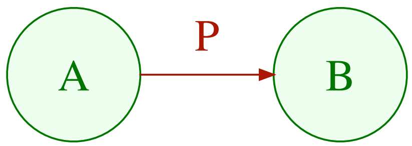
- Tension between scalability, determinism, and deadlock
  - Deadlock: processes waiting on each other in a cycle
  - Coarse-grained locks yield systems that do not scale well
  - Insufficient locking may cause non-determinate execution
- Industry approaches leave concurrency issues to programmer
  - Threads are "wildly nondeterministic" [Lee 2006]
  - Message Passing Interface compared to "assembly language"
- Formal models can handle complications of concurrency

Outline

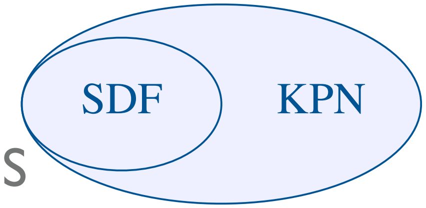
- Need for speed
- Dataflow models
- Contributions
  - Dynamic Distributed Deadlock Detection & Resolution (D4R)
  - The Computational Process Network (CPN) model
  - CPN framework implementation & case studies
- Conclusion

Dataflow Models

- Programs are modeled as directed graphs
  - Each node represents a computational unit
  - Each edge represents a one-way first-in first-out queue
  - Nodes may have any number of input or output edges
  - Nodes may communicate only via these edges

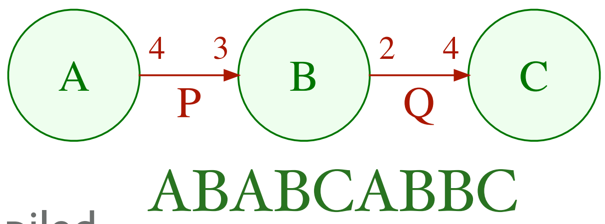


- Dataflow naturally models functional parallelism in systems
- Example dataflow models
  - Synchronous Dataflow (SDF), used in Agilent ADS
  - Kahn Process Networks (KPN), used in NI LabVIEW



Static Dataflow Models

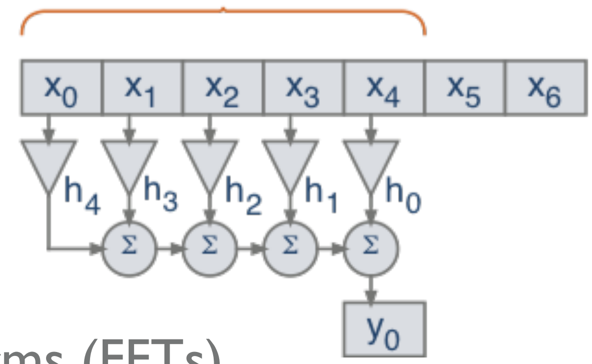
- Firing behavior of each node is known and static
  - Synchronous Dataflow (SDF) [Lee 1986]
  - Computation Graphs (CG) [Karp & Miller 1966]
- Termination & boundedness decidable
  - Flow of control and memory usage can be compiled
  - Schedule constructed once and repeatedly executed



- CG has parameters at each queue
  - U: number of tokens inserted by the producer at each firing
  - W: number of tokens removed by the consumer at each firing
  - T: (*firing threshold*) tokens present before consumer fires T≥W
- SDF is a special case of CG where T=W for all queues

Firing Thresholds

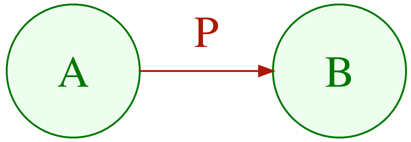
- A node can access more tokens than it will dequeue
- Model sliding window algorithms
  - Common in signal processing
  - Digital filters, y = x * h
  - Overlap-and-save fast Fourier transforms (FFTs)



- Queue maintains state and node becomes memoryless
  - Prevents node from having to make local copy of state
  - Enables optimizations for data management in queue

Kahn Process Networks

- A *networked set* of Turing machines, dynamically scheduled
- Determinate execution regardless of execution order
  - Sequential or concurrent execution
  - Mathematically proven model [Kahn 1974]
  - **Composable:** nodes can be clustered into a hierarchy to create larger, more complex systems



- Dynamic firing rules at each node
  - *Blocking reads:* suspend execution when attempting to read from an empty queue (necessary for determinism)
  - *Non-blocking writes:* never suspend a node for producing
- Possibly **unbounded:** may require infinite memory
- Termination and boundedness are undecidable in finite time

Dataflow Model Properties

	Model of Computation		
Property	SDF	CG	KPN
Determinism	✔	✔	✔
Boundedness	✔	✔	
Scalability			✔
Composability			✔
Firing Thresholds		✔	

SDF: Synchronous Dataflow
CG: Computation Graphs
KPN: Kahn Process Networks

Contributions

- Distributed Dynamic Deadlock Detection and Resolution (D4R)
  - For execution of KPN and CPN in bounded memory
- New model: Computational Process Networks
  - Built on formal underpinnings of KPN
  - Add firing thresholds and maintain scalability and composability
  - Bounded scheduling and enhancements for efficient implementation
- CPN Implementation and Case Studies
  - High-performance, scalable, distributed, and low overhead
  - Open-source implementation framework on POSIX (Unix) systems

CPN preserves the formal properties of KPN and reduces operations to implement common signal processing algorithms.

KPN Bounded Scheduling

- Execute KPN in bounded memory, if possible [Parks 95]
  - Place bounds on queue sizes and use blocking writes
  - Queue bounds may introduce *artificial deadlock*
  - Requires dynamic detection & resolution of deadlocks
  - Lengthen shortest deadlocked full queue to resolve
- Effective: all tokens produced are eventually consumed
- Fair: nodes cannot indefinitely ignore any input or output

	Parks 95	Geilen & Basten 03	D4R
Deadlock type specified	Global	Local	Local
Complete execution	No	Yes, if *effective* KPN	Yes, if *fair* KPN

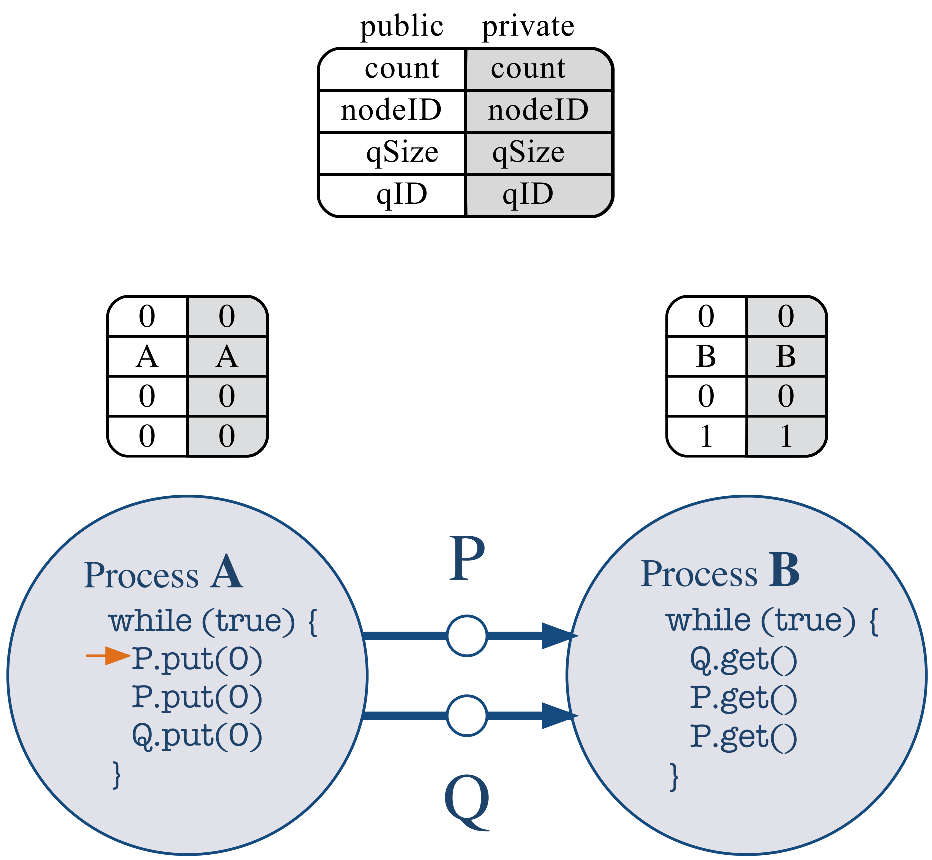
- Distributed algorithm by [Mitchell & Merritt 84] can detect local deadlocks in KPN [Olson & Evans 05]

Contribution #1: D4R

- D4R algorithm for KPN and CPN [Allen & Evans 07]
  - Based on a different priority-based distributed algorithm [M&M 84]
  - Detects whether deadlock is present
  - Determines whether a detected deadlock is real or artificial
  - If artificial, identifies the node blocked on culpable queue
  - Artificial deadlock is resolved by enlarging the culpable queue

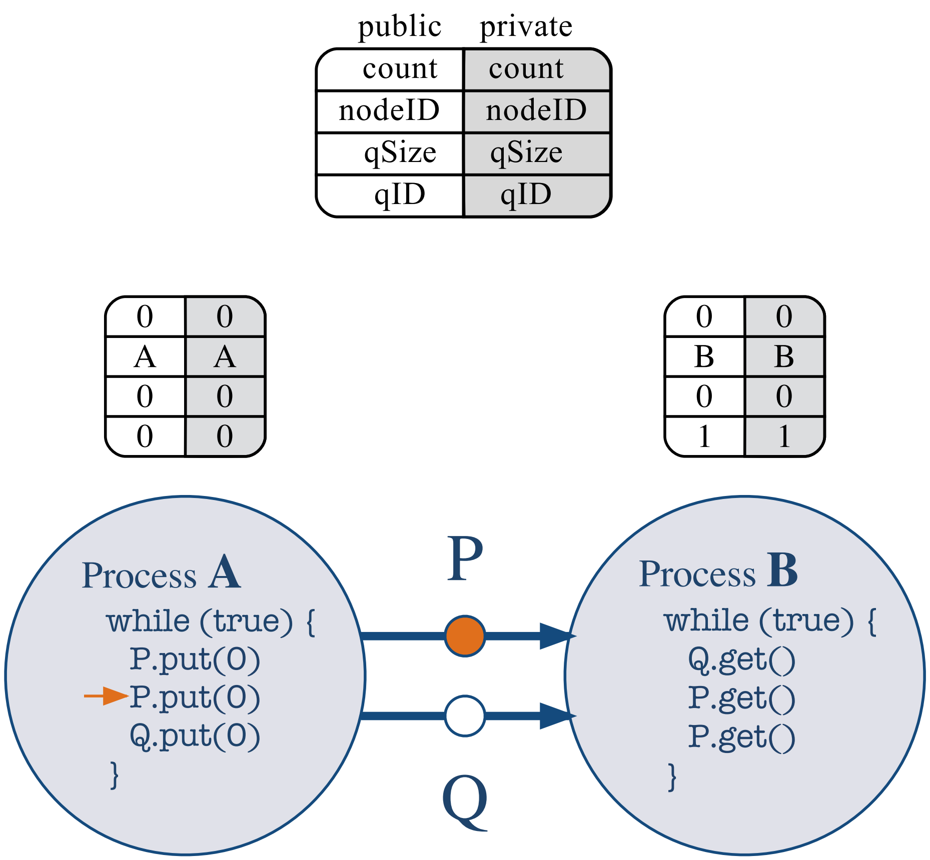
- Distributed and scalable
  - Each process contains D4R state variables
  - D4R state transactions occur between interacting processes

Example: Artificial Deadlock Detection & Resolution with D4R



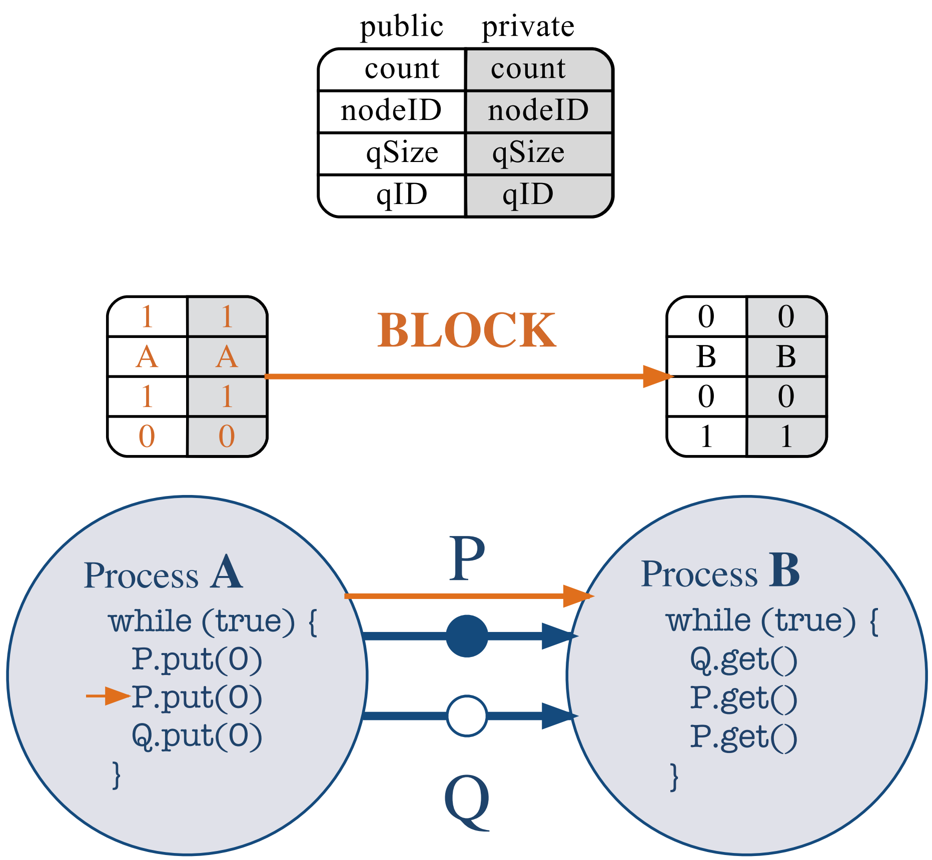
- Each node is an independent process
- Each node contains D4R state variables
  - Four state variables
  - Public and private sets
- Four state transitions
- Nodes directly interact
- Example is feed-forward
- One of several possible orders of execution

Example: Artificial Deadlock Detection & Resolution with D4R



1. **A** writes to **P**

Example: Artificial Deadlock Detection & Resolution with D4R



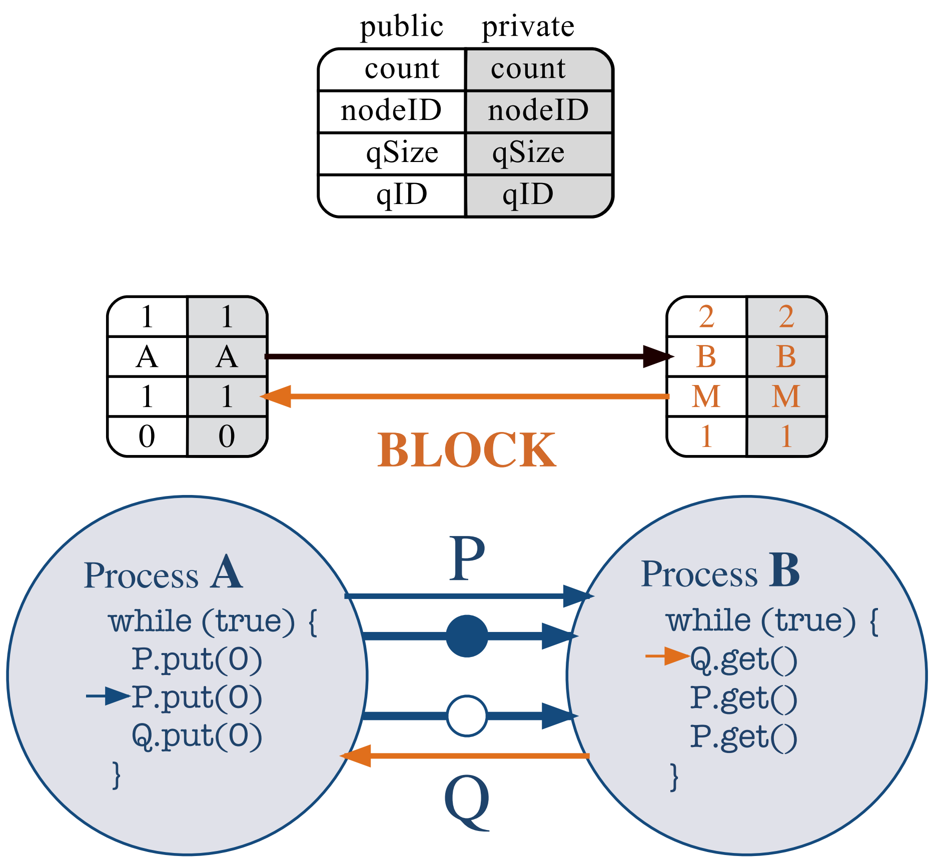
1. **A** writes to **P**
2. **A** *blocks* writing to **P**

   D4R state updated for ***A****:*

   count incremented and

   qSize set to size of ***P***

Example: Artificial Deadlock Detection & Resolution with D4R



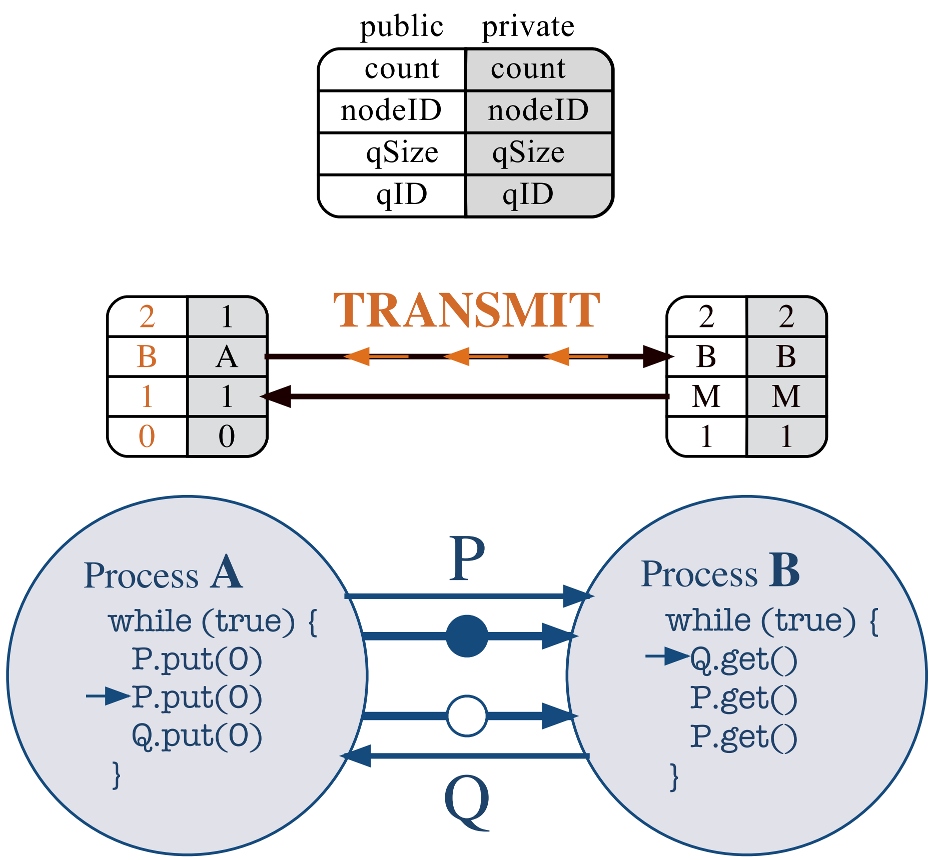
1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**

   D4R state updated for ***B****:*

   count incremented and

   qSize set to MAX_UINT

Example: Artificial Deadlock Detection & Resolution with D4R



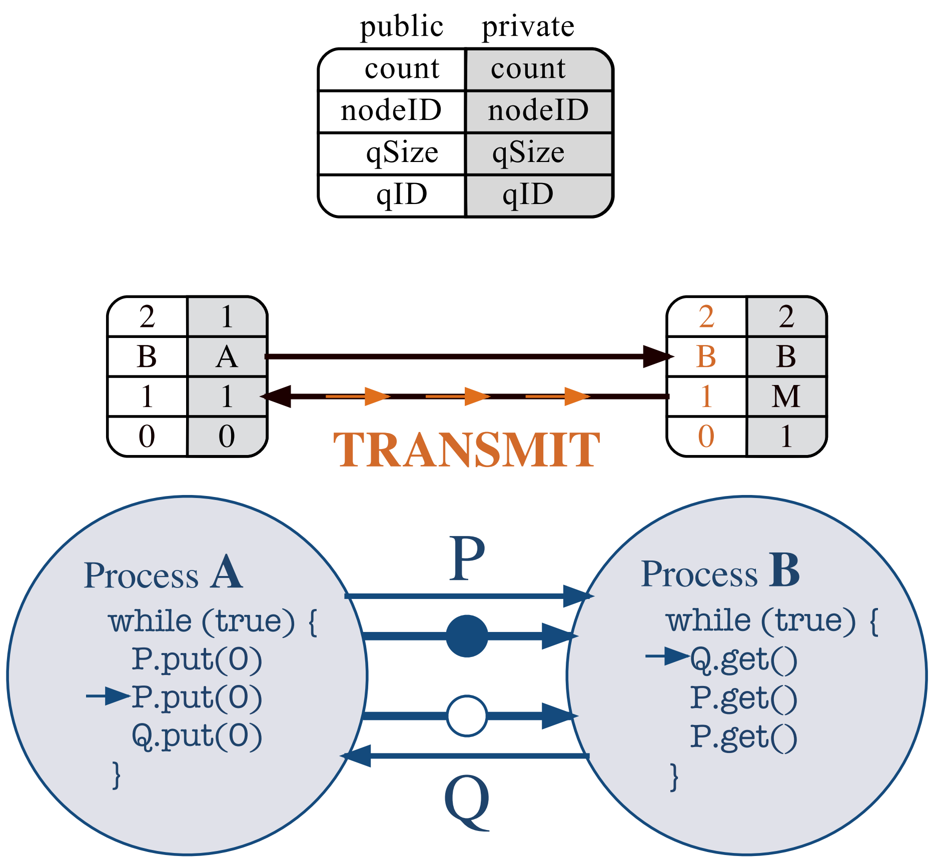
1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**

   *D4R state updated for **A**:*

   keep larger count:nodeID

   and smaller qSize:qID

Example: Artificial Deadlock Detection & Resolution with D4R



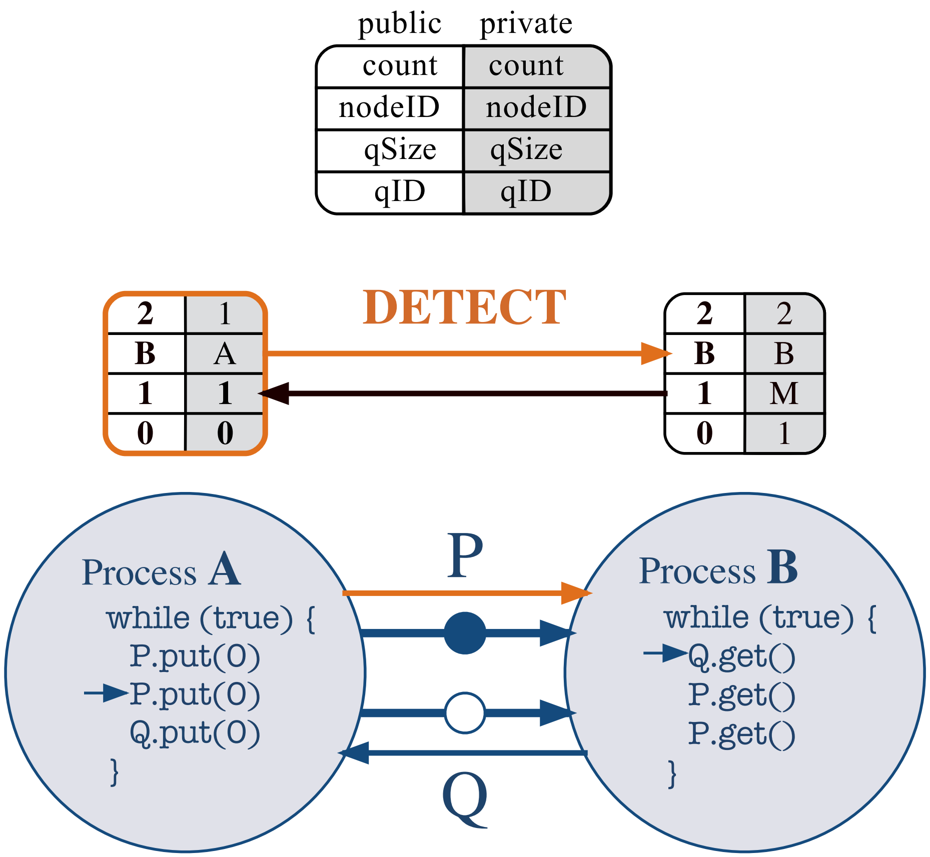
1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**
5. **A** *transmits* to **B**

   D4R state updated for ***B****:*

   keep larger count:nodeID

   and smaller qSize:qID

Example: Artificial Deadlock Detection & Resolution with D4R



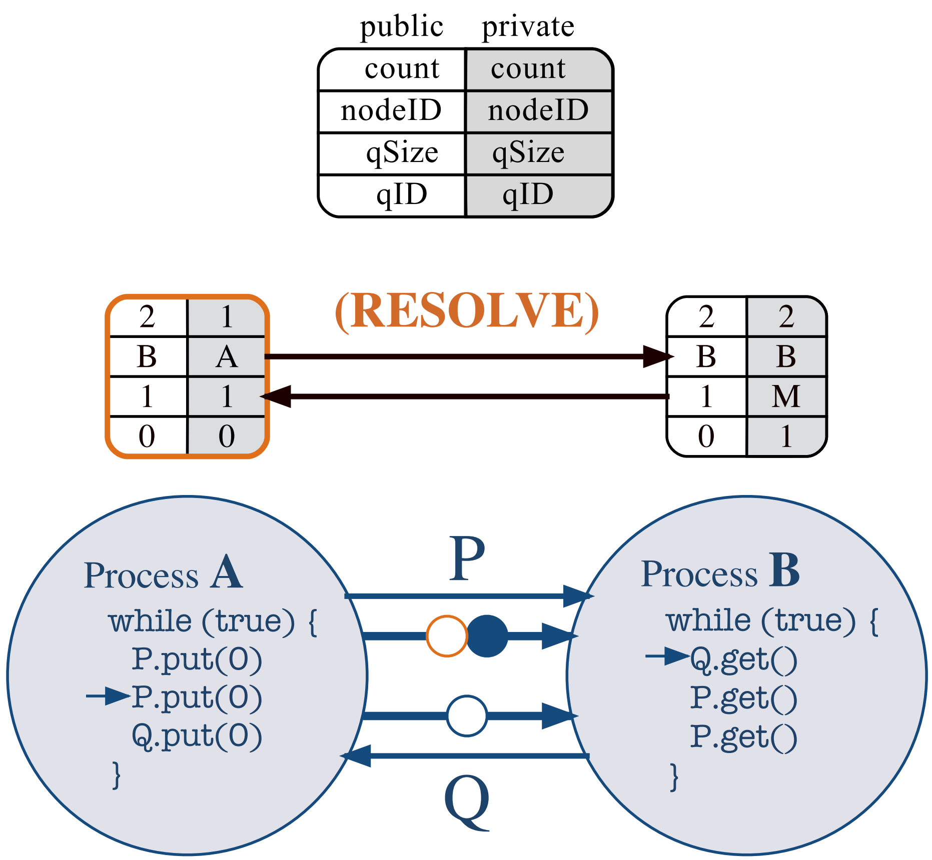
1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**
5. **A** *transmits* to **B**
6. **A** *detects* deadlock

   if qSize != MAX_UINT,

   deadlock is artificial and

   ***A** blocked on culpable queue*

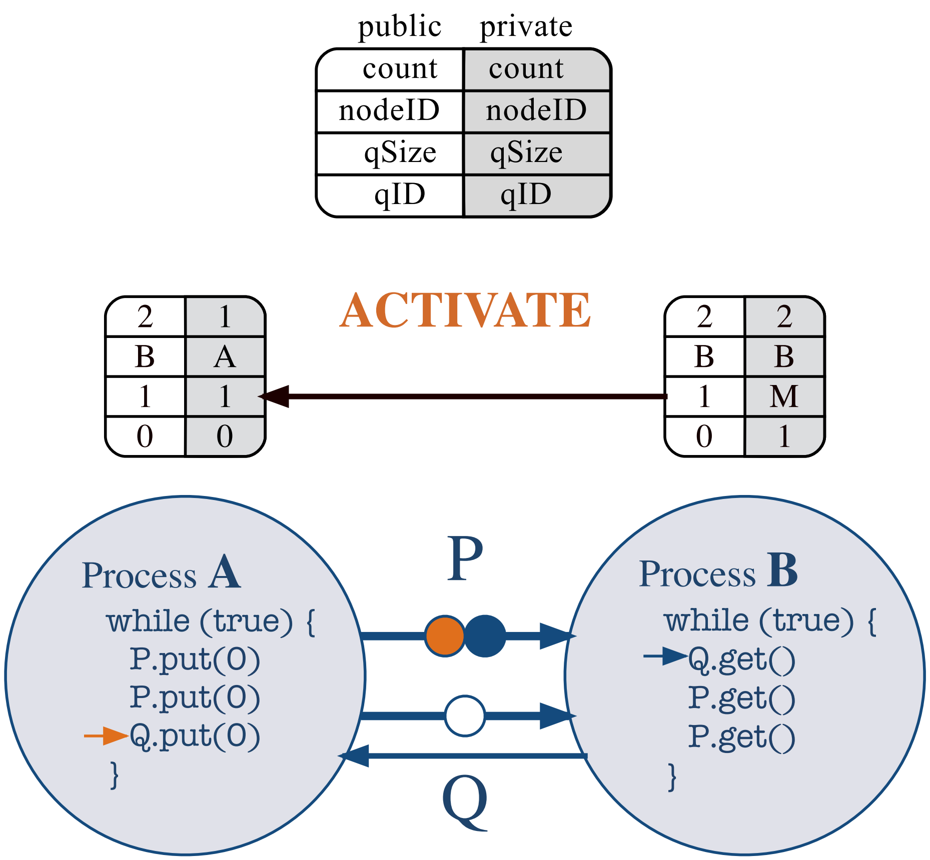
Example: Artificial Deadlock Detection & Resolution with D4R



1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**
5. **A** *transmits* to **B**
6. **A** *detects* deadlock
7. deadlock resolved

culpable queue grows

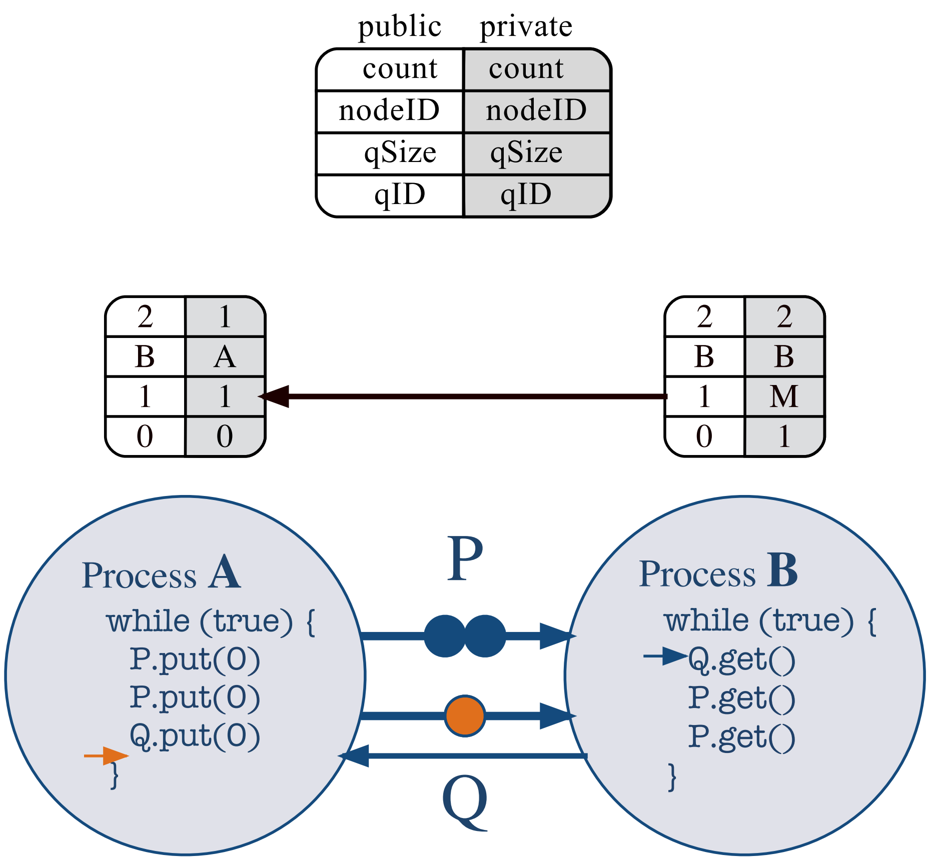
Example: Artificial Deadlock Detection & Resolution with D4R



1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**
5. **A** *transmits* to **B**
6. **A** *detects* deadlock
7. deadlock resolved
8. **A** activates, writes to **P**

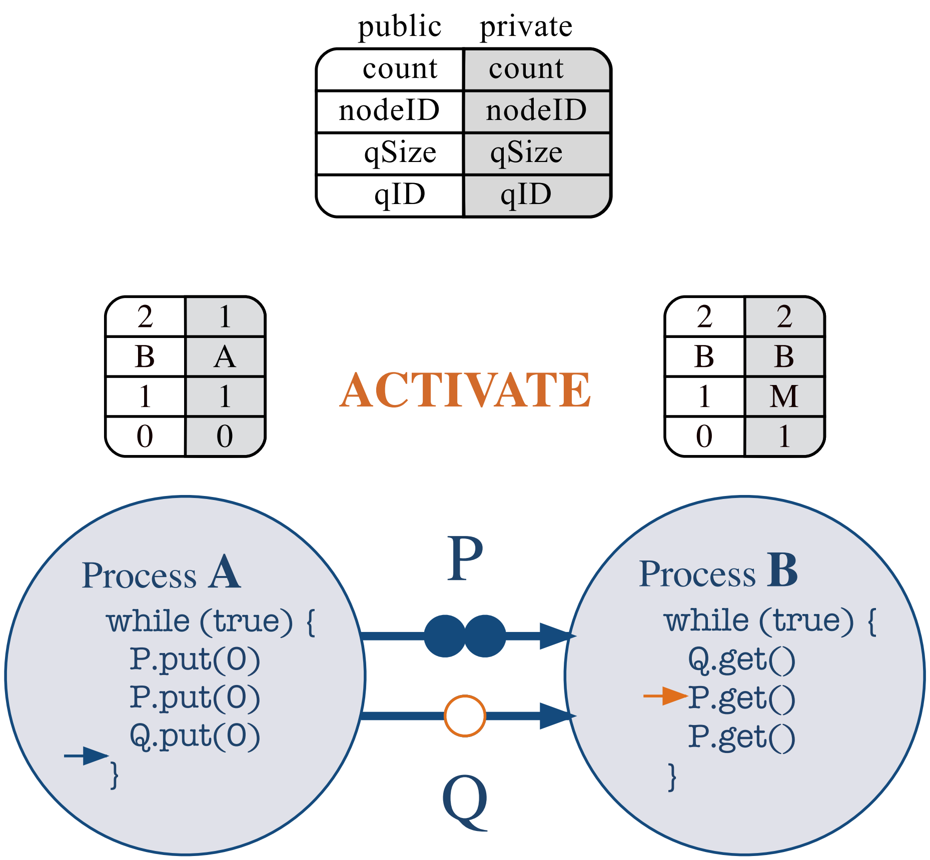
dependency removed

Example: Artificial Deadlock Detection & Resolution with D4R



1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**
5. **A** *transmits* to **B**
6. **A** *detects* deadlock
7. deadlock resolved
8. **A** activates, writes to **P**
9. **A** writes to **Q**

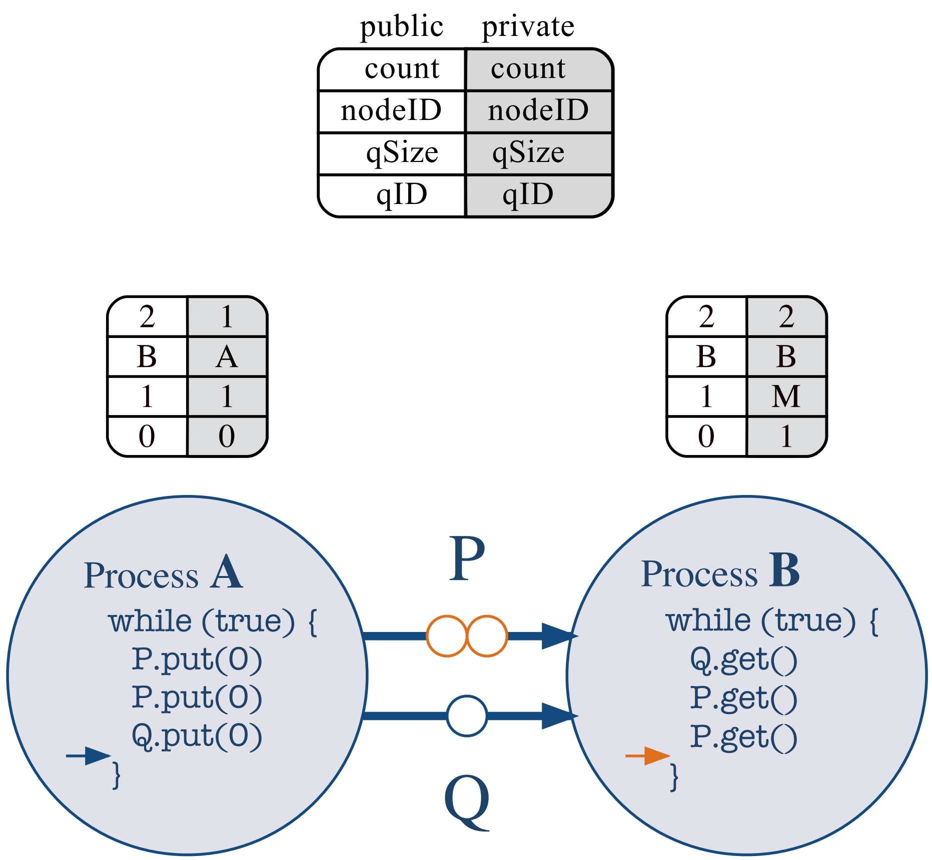
Example: Artificial Deadlock Detection & Resolution with D4R



1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**
5. **A** *transmits* to **B**
6. **A** *detects* deadlock
7. deadlock resolved
8. **A** activates, writes to **P**
9. **A** writes to **Q**
10. **B** activates, reads from **Q**

dependency removed

Example: Artificial Deadlock Detection & Resolution with D4R



1. **A** writes to **P**
2. **A** *blocks* writing to **P**
3. **B** *blocks* reading from **Q**
4. **B** *transmits* to **A**
5. **A** *transmits* to **B**
6. **A** *detects* deadlock
7. deadlock resolved
8. **A** activates, writes to **P**
9. **A** writes to **Q**
10. **B** activates, reads from **Q**
11. **B** reads (twice) from **P**

Contribution #2: CPN Model

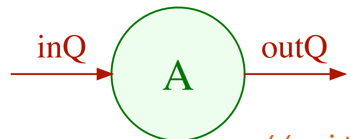
- Preserves KPN determinism, scalability, and composability
- Reduces operations for common signal processing algorithms
- Bounded memory when possible with D4R (Contribution #1)
- Enhancements for streaming data
  - Multi-token queue transactions to reduce overhead ← LabVIEW's "G" language traditionally used single-token transactions
  - Multi-channel queues for multi-dimensional synchronized data
  - Firing thresholds for both consumers and producers ← Computation Graphs have only consumer firing thresholds
  - Zero-copy queue transactions
- Enables high-throughput signal processing

CPN Queue Semantics

- Bounded queue sizes and blocking reads and writes
- Producer and consumer firing thresholds
- CPN semantics use two steps each for *read* or *write*
  - GetDequeuePtr(*threshold*, *channel*) blocks until sufficient tokens are readable in input queue, returns contiguous token array for consumption
  - Dequeue(*count*) dequeues tokens from head of input queue
  - GetEnqueuePtr(*threshold*, *channel*) blocks until sufficient free space is available in output queue, returns contiguous token array for writing
  - Enqueue(*count*) enqueues tokens from array head into output queue
- These semantics provide a zero-copy interface for queue I/O

CPN vs. KPN Semantics

FIR filter in the frequency domain using 50% overlap-and-save FFT




```
// with (extended) bounded KPN semantics
typedef complex<float> T;
const int nfft = 1024;
T filter[nfft];
T workBuf[nfft];
while (true) {
    // manage sliding window state
    memcpy(workBuf, workBuf+nfft/2, nfft/2*sizeof(T));
    // blocking call to copy in new data
    inQ.read(workBuf+nfft/2, nfft/2);

    // execute one step of filter
    fft(workBuf, workBuf, nfft);
    cpx_multiply(filter, workBuf, workBuf, nfft);
    ifft(workBuf, workBuf, nfft);

    // blocking call to copy out the results
    outQ.write(workBuf, nfft/2);
}
```

memory for overlap state

data copies

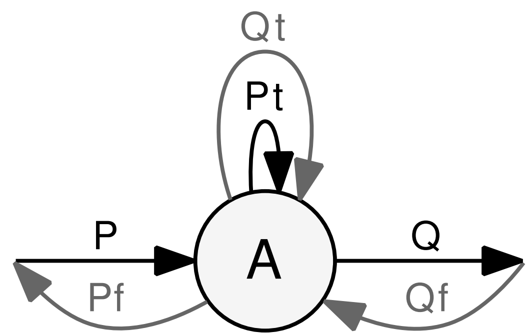
```
// with CPN semantics
typedef complex<float> T;
const int nfft = 1024;
T filter[nfft];

while (true) {
    // blocking calls for in/out buffer pointers
    const T* inPtr = inQ.GetDequeuePtr(nfft);
    T*       outPtr = outQ.GetEnqueuePtr(nfft);

    // execute one step of filter
    fft(inPtr, outPtr, nfft);
    cpx_multiply(filter, outPtr, outPtr, nfft);
    ifft(outPtr, outPtr, nfft);

    // complete the queue transactions
    inQ.Dequeue(nfft/2);
    outQ.Enqueue(nfft/2);
}
```

Preserving KPN's Properties



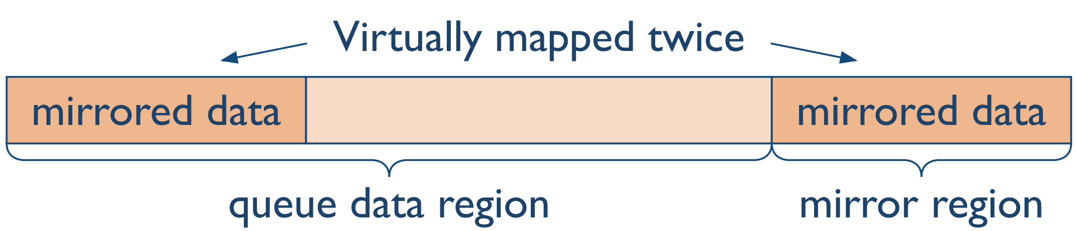
- Any CPN program can be transformed to KPN
  - Adding queues and modifying each node
  - Feedback queues (P_f and Q_f) are for boundedness
  - Self-loop queues (P_t and Q_t) are for managing firing thresholds
  - Grayed queues carry placeholder feedback tokens (value is unimportant)
- All tokens entering a process pass through self-loop queue
  - *GetDequeuePtr* ensures self-loop contains at least threshold number of tokens
  - *Dequeue* discards requested number of tokens from self-loop queue
  - *GetEnqueuePtr* and *Enqueue* behave similarly but with feedback tokens
- Same mathematical representation, formal properties preserved

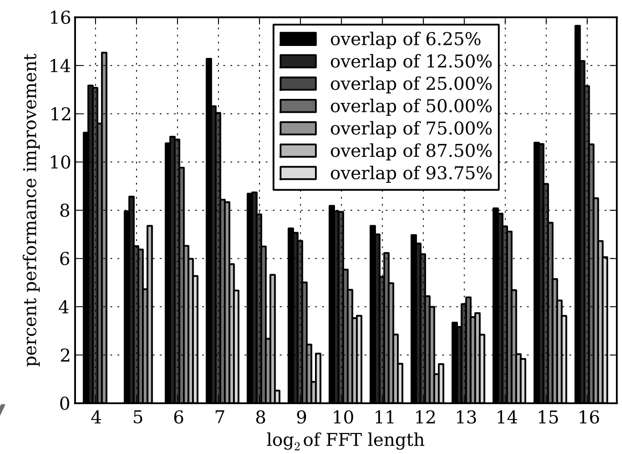
Contribution #3: CPN Framework

- High-performance implementation of the CPN model
- Scalable framework in C++, targeting POSIX (Unix) systems
  - Released as an open source library under the GNU LGPL license
  - More than 4 work-years of development effort, 26K lines of code
  - D4R algorithm for bounded scheduling
  - Unit tests and 72 hour tests for robustness and stability
- Developers can build high-throughput distributed systems from deterministic, composable components
- Case studies on both multi-core and distributed platforms

CPN Nodes and Queues

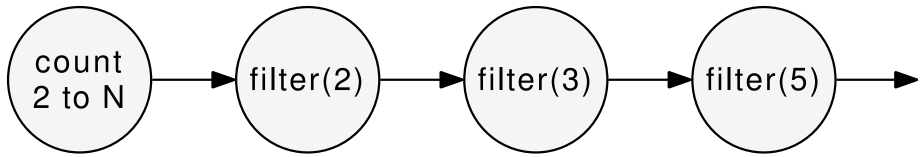
- Each CPN Node maps onto a single POSIX thread (Pthread)
- CPN Queues have firing thresholds and zero-copy interface
  - Nodes operate directly on queue memory to avoid unnecessary copies
- CPN Queues use mirroring for contiguous data [Allen *et al.* 2006]



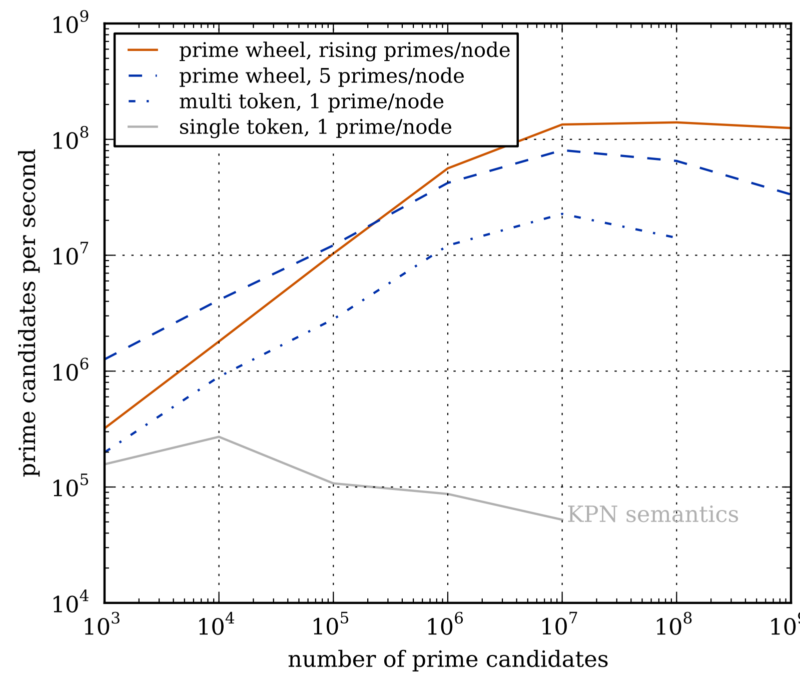


  - Circular buffers similar to modulo addressing
  - Virtual memory manager maintains data circularity
- OS dynamically schedules and load balances nodes (threads)

Prime Sieve Case Study

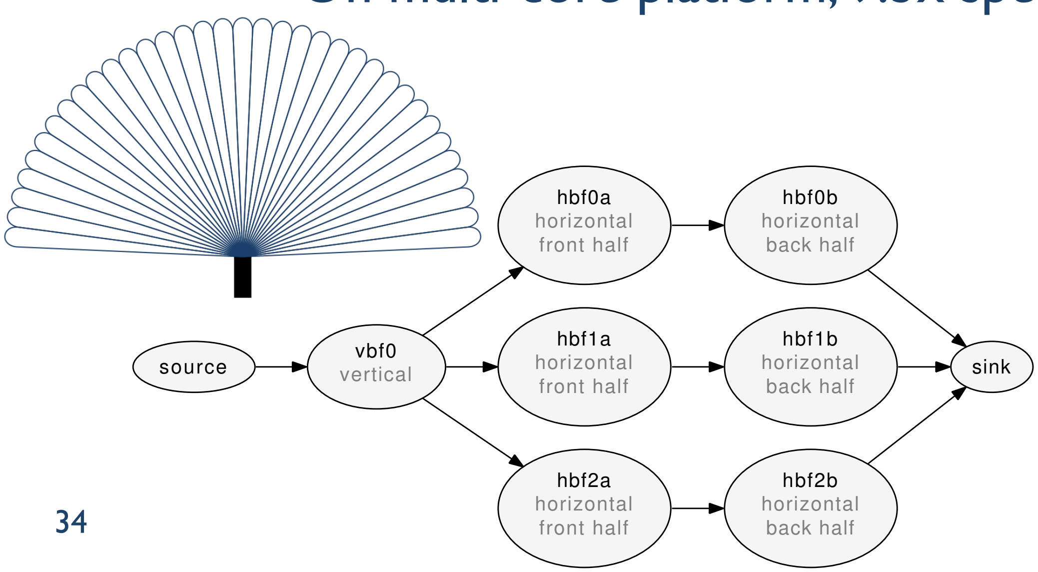


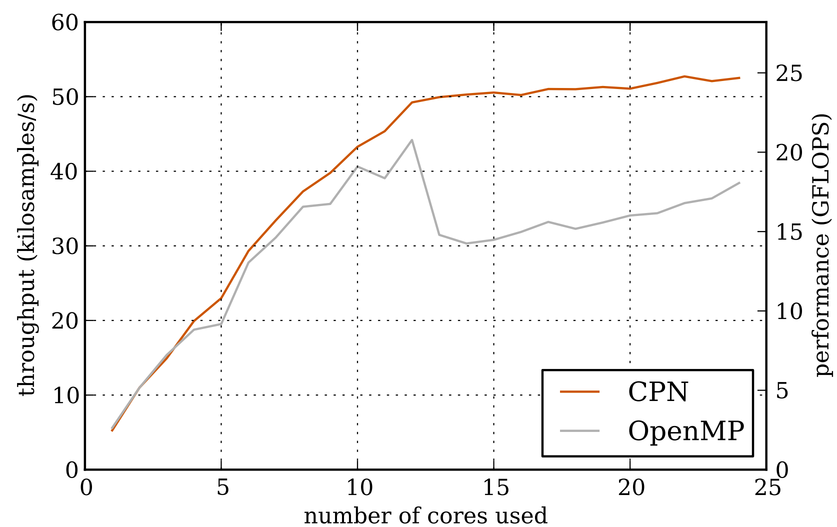
- Simple algorithm for finding prime numbers [Eratosthenes 250BCE]
  - First real example for KPN, requires dynamic creation & recursion
- Multi-core platform: 12x 2.66 GHz Intel Xeon with Hyper-Threads
- Guidelines for high performance
  - Multi-token firings reduce overhead
  - Node granularity vs. context switch
  - Load balancing of nodes
- 2500x speedup for 10^7 candidates



Beamformer Case Study

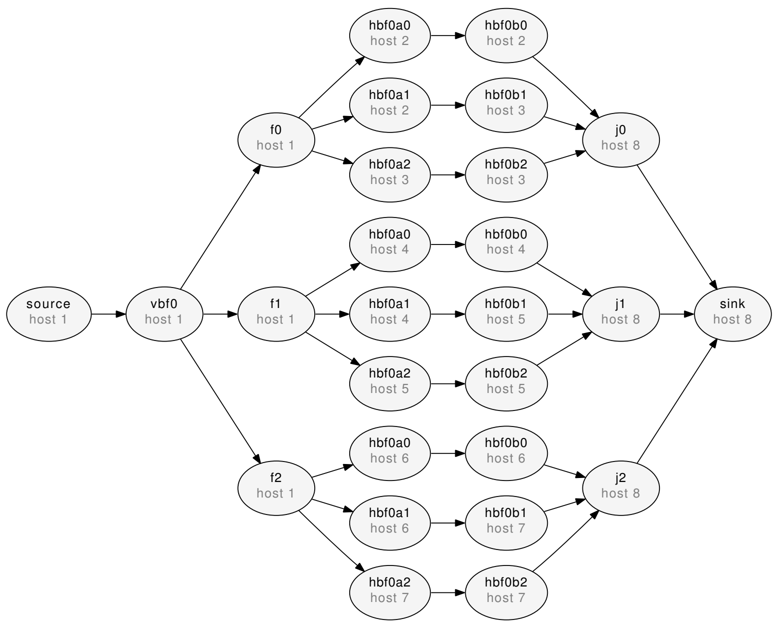
- Multiple beams formed from cylindrical sensor array outputs
  - Decomposed into horizontal and vertical components
  - Optimized kernels use SIMD and OpenMP loop parallelism
  - Horizontal beamformer uses FFTs and performs matched filtering
- At 50 ks/s target: 614 MB/s in, 672 MB/s out, 24 GFLOPS
- On multi-core platform, 9.3x speedup at 12 cores

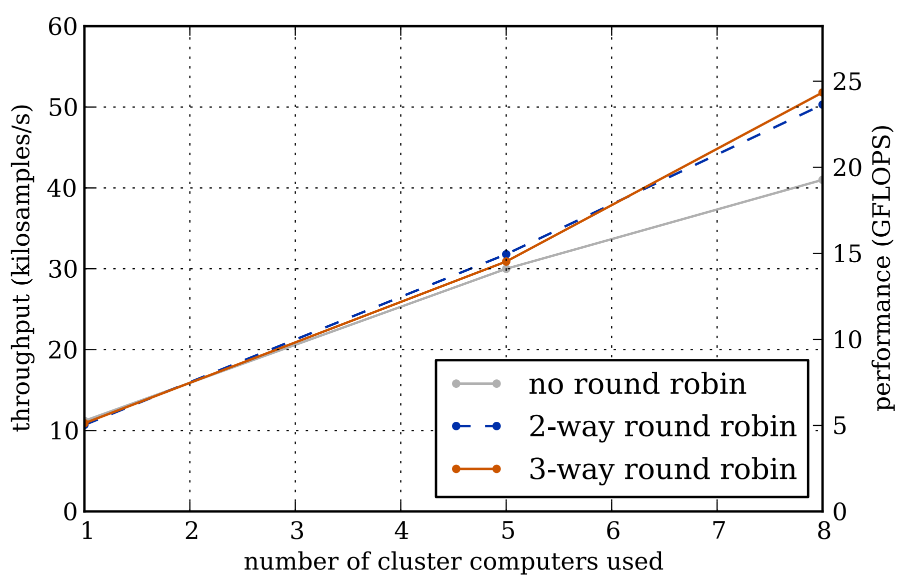




Beamformer Case Study

- On 8-host cluster connected by 8 Gigabit Infiniband network
  - Each host with four 2.33 GHz Intel Xeon processors
- Mapping file to distribute and load balance CPN program
- Increase data parallelism of horizontal with time multiplexing
- 4.6x speedup on 8 hosts w/ Infiniband output at 70% of peak





Conclusion

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

* Execution of *fair* KPN and CPN in bounded memory with D4R

CPN preserves the formal properties of KPN and reduces operations to implement common signal processing algorithms.

Future Work

- Improve D4R algorithm
  - Artificial deadlocks can occur without cycles [Basten&Hoogerbrugge 2001]
  - A similar edge-chasing algorithm could detect these deadlocks
- CPN Node migration and distributed scheduling
  - Automated load balancing on cluster computers
- CPN Queues with Remote Direct Memory Access (RDMA)
  - Higher throughput, reduced overhead on cluster systems
- Integrate into design automation tools (graphical programming)
- Additional targets and applications