Computational Process Networks:

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

Gregory E. Allen <gallen@arlut.utexas.edu>
Supervising Professor: Brian L. Evans <bevans@ece.utexas.edu>

Embedded Signal Processing Laboratory
ESPL
The University of Texas at Austin

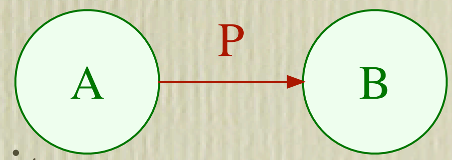
ARL
The University of Texas at Austin

WNCG
Wireless Networking & Communications Group

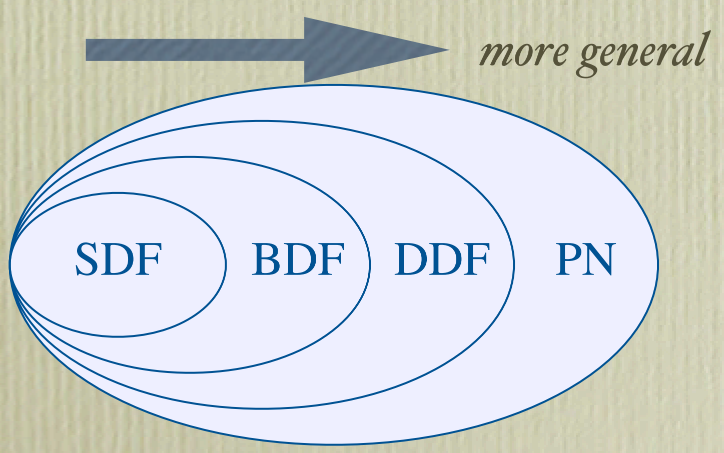
Motivation

- Signal processing systems are growing in size & complexity
- Parallel & distributed implementations for high throughput
- Commodity HW/SW reduce development time & cost
- *Problem:* Effective parallel programming is difficult
    - Non-determinate (unpredictable) execution
    - Hard to predict and prevent deadlock
    - Difficult to make software scalable (e.g. rendezvous models)
- Current approaches typically lack formal underpinnings

Dataflow Models



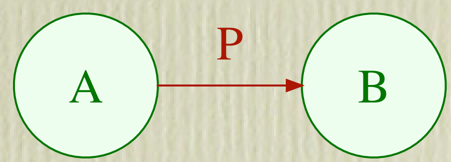
- Model programs as directed graphs
  - Each node represents a computational unit
  - Each edge represents a one-way FIFO queue
- Communicate data solely on the edges of the graph, and a node may have any number of input or output edges
- Models functional/data parallelism in systems



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

Process Networks (PN)

- *Solution:* A formal model -- Process Networks [Kahn 74]
- A networked set of Turing Machines
- Mathematically provable properties
    - Guarantees determinate execution
    - Allows concurrent execution



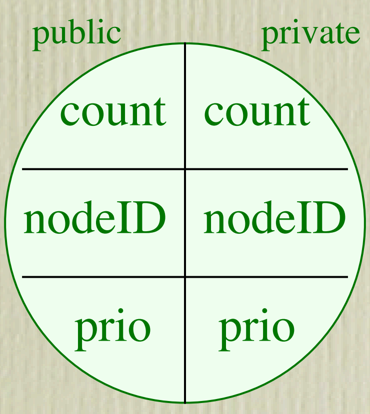
- Dynamic firing rules at each node
    - *Blocking reads*: suspend a node's execution when it attempts to consume data from an empty queue
    - *Non-blocking writes*: never suspend a node for producing data (so queues can grow without bound)
- Not directly implementable -- requires infinite memory

Bounded Scheduling of PN

- Clever dynamic scheduling of the nodes allows execution in bounded memory, if it is possible [Parks, 1995]
  - May introduce *artificial deadlocks* due to queue bounds
  - Relies on a global deadlock detector -- no local deadlock proposed
  - This can result in an incomplete execution -- violates PN model
- Local deadlock detection is required, and preserves the formal properties of the PN model [Geilen & Basten, 2003]
- Distributed deadlock detection algorithm [Mitchell & Merritt, 1984] can be applied to Bounded PN [Olson & Evans, 2005]
- *New result*: A prioritizing algorithm can also be applied to:
  - Determine whether a deadlock is real or artificial
  - Localize where to resolve artificial deadlocks

Distributed Dynamic Deadlock Detection and Resolution (D4R)

- Each node contains public and private sets of three fields:
  - A **count** field which is non-decreasing over time
  - A **nodeID** field which is unique over the entire program
  - A **priority** field containing the queue size, description to follow
- All algorithm states and transitions are based on fields
- No global knowledge required -- all transactions are local
- Algorithm is distributable and scalable



Distributed Dynamic Deadlock Detection and Resolution (D4R)

- Modification of Mitchell & Merritt [1984] priority algorithm
  - When a process blocks on read, its priority is set to negative one
  - When a process blocks on write, its priority is set to queue size
  - Modified algorithm find smallest *non-negative* priority process
- If the algorithm detects a deadlock, then:
  - A negative priority indicates a true deadlock
  - A non-negative priority indicates artificial deadlock;
the associated queue must grow

Computational Process Networks (CPN)

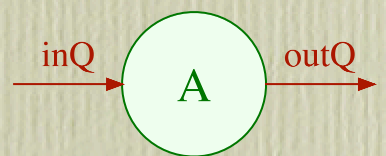
- New model for of high-performance parallel computation
  - Formal underpinnings, but implementable, scalable, and efficient
- Begin with the Process Network model [Kahn, 1974]
  - Provides formal determinism with parallel/distributed execution
- Utilize bounded scheduling and distributed deadlock detection and resolution [from previous]
  - 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]

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 queue transactions are broken into two functions




```
// 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);
}
```

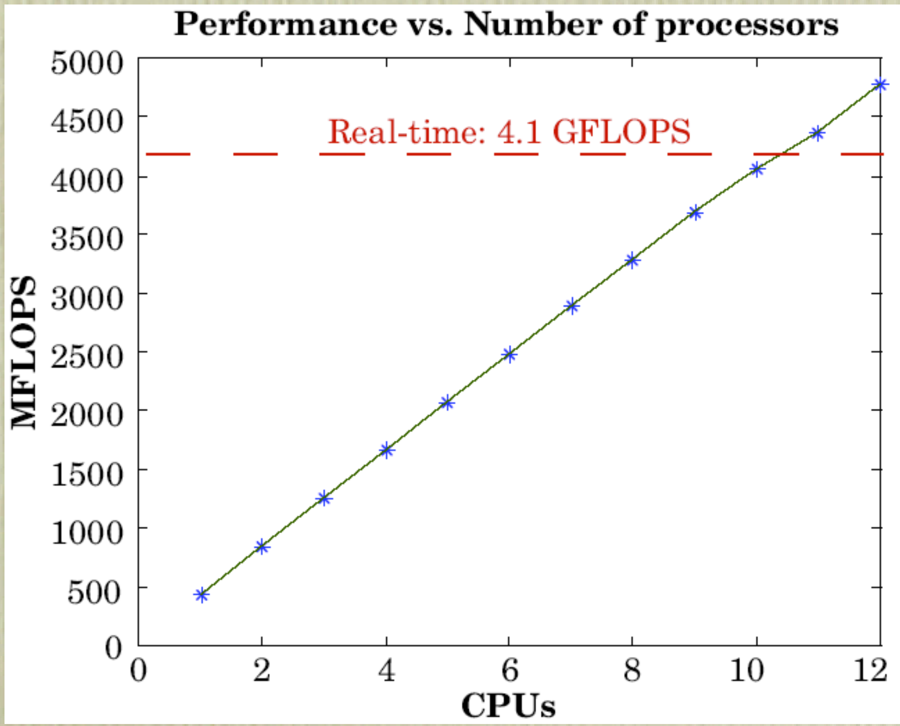
```
// PN code
typedef complex<float> T;
T filter[1024];
T tmpData[1024];
while (true) {
    // do overlap-save, get new data
    memcpy(tmpData, tmpData+512, 512*sizeof(T));
    inQ.get(tmpData+512, 512);

    // 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 threads
- Original implementation:
  - Limited to shared memory (SMP) system
  - No deadlock detection (artificial or otherwise)
  - Transactions Article in 2000: Sonar Beamformer on a 12-CPU Sun
- New implementation underway:
  - Distributed networked systems (especially workstation clusters)
  - Scalable distributed deadlock detection and resolution
  - Dynamic construction of a CPN program from an XML file
- Source code available: `www.ece.utexas.edu/~allen/CPN/`

Previous Results
(IEEE Trans. on Sig. Proc. 2000)

- Sonar Beamformer using Process Network framework
- 12-way SMP UltraSPARC-II Sun at 336 MHz
- CPN framework vs. sequential case and thread pools



- On one CPU, slowdown < 0.5%
- On 8 CPUs vs. thread pools
  - 7% faster
  - 20% less memory
- On 12 CPUs
  - Speedup is 11.28
  - Efficiency is 94%
  - Runs at real-time plus 14%