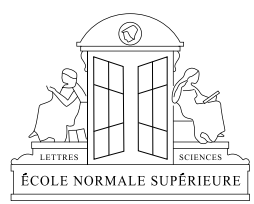
Dynamic Streaming: Applications, Constructs and Challenges

Albert Cohen

PARKAS Team
INRIA and ENS Paris

April 14th, 2014

Funded by: TERAFLUX & PHARAON FP7, and ManycoreLabs "investissements d'avenir" grants



Stream Processing?

"A model that uses sequences of data and computation kernels to expose and exploit concurrency and locality for efficiency [of execution and programmability]."

Two Workshops on Streaming Systems

- WSS03: `http://groups.csail.mit.edu/cag/wss03`
- WSS08: `http://people.csail.mit.edu/rabbah/conferences/08/micro/wss`

Stream Processing – In This Talk

Application domains

1. High-productivity computing systems
2. Efficient runtimes, task-level pipelines
3. Embedded control, safety-critical, certified systems

Some influential languages

- Block-diagram: StateCharts, Scade, Simulink, LabVIEW
- Synchronous: Lustre, Esterel, Signal
- Data-flow: Lucid, Linda, SISAL, pH, SAC, CnC
- Kahn network APIs: YAPI, CAL
- Cyclo-static data flow: StreamIt, SigmaC

Stream Processing – Missing Something?

Bill Dally's streaming processors

- Bulk-Synchronous Parallelism? Vector processing pipelines?
- Brook, Cg? CUDA, OpenCL?

3-phase, "hardware-centric" decoupling: *load*⟶*compute*⟶*store*

- Special case of Kahn networks
- Implicitely relies on chaining/fusion to save on memory transfers

Foundations: Kahn Process Networks

Kahn networks, 1974

Gilles Kahn (1946–2006)

Denotational: least fixpoint of a *system of equations* over *continuous* functions, for the Scott topology lifted to unbounded *streams*

$$s \sqsubseteq s' \Longrightarrow f(s) \sqsubseteq f(s') \qquad + \text{ lifted to the limit}$$

Operational: communicating processes over *unbounded FIFOs* with *blocking reads*

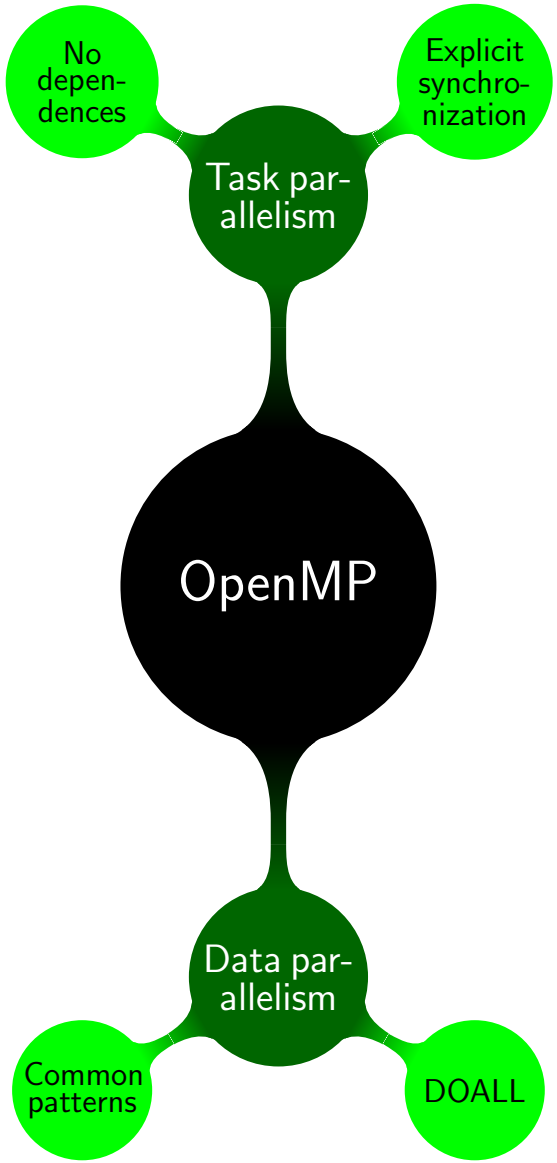
- → *Deterministic* by design
- → General recursion (dynamic process creation), *parallel composition*, *reactive systems*
- → Distribute and decouple computations from communications

1. Task-Level Data-Flow and Streaming Languages

Example: Compilation of the OpenStream Language

OpenMP extension

- `http://openstream.info`
- GCC-based
- maximize *productivity*



Example: Compilation of the OpenStream Language

OpenMP extension

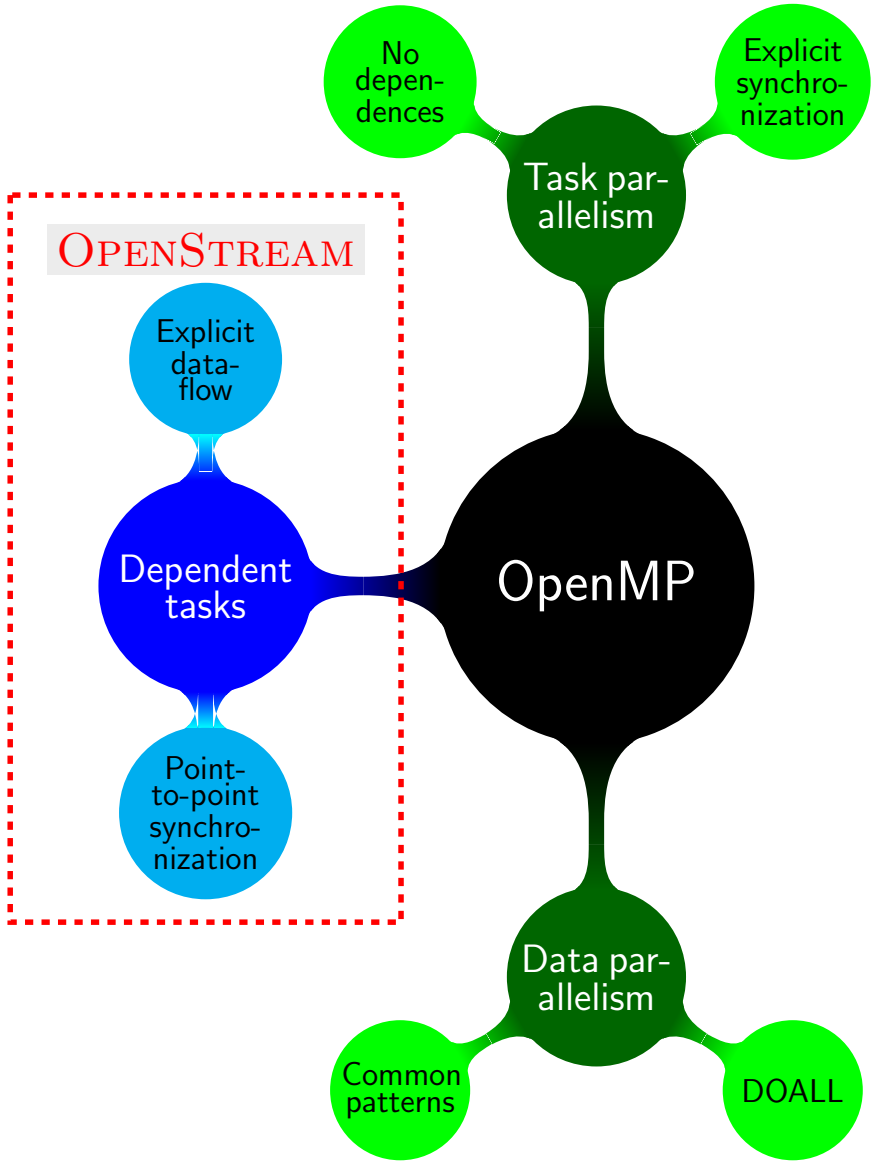
- `http://openstream.info`
- GCC-based
- maximize *productivity*

Parallelize irregular codes

- no static/periodic restriction
- maximize *expressiveness*

Efficient execution model

- save precious local memory
- avoid communications



OpenStream Introductory Example

```
for (i = 0; i < N; ++i) {

  #pragma omp task firstprivate (i) output (x) // T1
    x = foo (i);

  #pragma omp task input (x)                   // T2
    print (x);
}
```

Control program sequentially creates ***N*** instances of ***T1*** and of ***T2***

Firstprivate clause privatizes variable ***i*** with initialization at task creation

Output clause gives write access to stream ***x***

Input clause gives read access to stream ***x***

Stream ***x*** has FIFO semantics

Generalization: Control-Driven Data Flow

Formal semantics of imperative programming languages with dynamic creation of dependent tasks

- *Control program*
  - *control flow*: dynamic construction of a task graph
  - model: graph of *activation points*, each generating a *task activation*
- *Tasks*
  - imperative program with a (dynamic) stream access signature
  - becomes executable once its dependences are satisfied
  - recursively becomes the control program for tasks created within
    - work in progress
    - link with synchronous languages
  - model: *task activation* defined as a set of *stream accesses*
- *Streams*
  - Kahn-style unbounded, indexed channels
  - multiple producers and/or consumers
  - specify dependences and/or communication
  - model: indexed set of memory locations, defined on a finite subset

Control-Driven Data Flow – Results

- Deadlock classification
  - insufficiency deadlock: missing producer before a barrier or control program termination
  - functional deadlock: dependence cycle
  - spurious deadlock: deadlock induced by CDDF semantics on dependence enforcement (Kahn prefixes)
- Conditions on program state allowing to prove
  - deadlock freedom
  - compile-time serializability
  - functional and deadlock determinism
- Conditions are weaker than Kahn continuity
- Relaxed form of strictness (Cilk)

Impact on Runtime Design and Implementation

Two common approaches, so far not reconciled

- *Lightweight scheduling of data-flow tasks*
- *FIFO-based synchronization with pressure and back-pressure*

2. Lightweight Scheduling

Work-Stealing for a Relaxed Memory Model

A relaxed lock-free work-stealing algorithm

Two implementations: C11 and ARM inline assembly

Based on state-of-the-art sequentially consistent algorithm (Chase and Lev, 2005)

Proven for the POWER/ARM relaxed memory model (axiomatic POWER model by Mador-Haim et al., 2012). POWER and ARM have the same memory model

[Lê et al., PPoPP 2013]

Throughput results

Experiments with work-stealing throughput: *task taken per second*

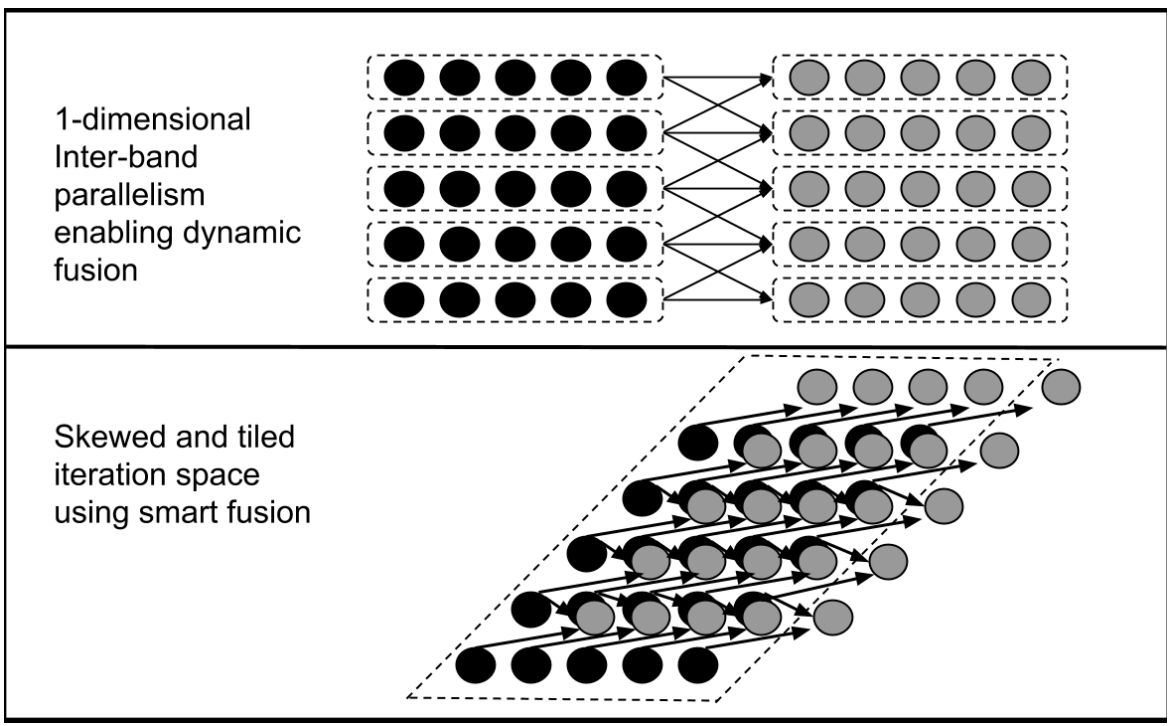
	Comb: $b = 1; d = 10^7$	Tree: $b = 3; d = 15$
Core i7 (2 threads)	$\mathbf{4.87862 \times 10^8}$	$\mathbf{3.60838 \times 10^8}$
Tegra 3 (2 threads)	$\mathbf{5.47223 \times 10^7}$	$\mathbf{4.12112 \times 10^7}$

Pretty good, but note that *dependent* task scheduling can be *much* slower: producer-consumer synchronizations accumulate (coherence, communication) delays on the critical path

Application: data-flow vs. bulk-synchronous parallelism

Example: parallelizing the Ring-Roberts filter

```
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N - 1; j++) // Gaussian blur
S1    B[i][j] = (A[i][j] + A[i][j-1] + A[i][1+j] +
      A[1+i][j] + A[i-1][j] + A[i-1][j-1] +
      A[i-1][j+1] + A[i+1][j-1] + A[i+1][j+1])/8;
for (i = 1; i < N-2; i++)
  for (j = 2; j < N-1; j++) // Edge detection
S2    A[i][j] = abs(B[i][j]-B[i+1][j-1]) +
      abs(B[i+1][j] - B[i][j-1]);
```



Application: data-flow vs. bulk-synchronous parallelism

Example: parallelizing the Ring-Roberts filter

```
parfor (ti=0; ti < ubti; ti++)
  for (tj=0; tj < ubtj; tj++)
   { /* tile body of S1 */ }
/* barrier */
parfor (ti=0; ti < ubti; ti++)
  for (tj=0; tj <= ubtj; tj++)
   { /* tile body of S2 */ }
/* barrier */
```

```
for (ti = 0; ti < ubti; ti++) {
  parfor (tj=0; tj < ubtj; tj++) {
   { /* tile body */ }
  /* barrier */
}
```

GFLOPS on AMD Opteron 6274 (opt) and Intel i7-2600 (i7)

processor-cores	ref ICC	pluto minfuse	pluto maxfuse	data-flow
opt-1	1.25	0.4	0.7	0.9
opt-8	1.25	2.7	3.9	4.7
opt-16	1.25	2.0	0.7	6.8
i7-1	3.4	2.6	2.3	2.8
i7-2	4.2	3.6	4.0	5.4
i7-4	4.1	3.5	4.3	10.1

3. FIFO-Based Synchronization

Alternative: Using Explicit FIFOs

- Good:
  - Does *not* depend on a task scheduler to enforce dependences
  - *Amortize* synchronization cost over multiple operations
- Bad:
  - Load balancing issues
  - Hard to bound a FIFO automatically
  - Cumbersome interaction with lightweight scheduling: blocking a task blocks the underlying worker (POSIX) thread!

Fast Implementations of SPSC Streams

- Simplest and efficient solution for powers of two: absolute indices (deal with buffer wrap-around and 32/64-bit overflows)
  - Lamport's seminal FIFO queue
- Index-cached FIFO: manual/software caching of the last index
  - Lee et al.: MCRB (Chinese U. of Hong Kong, Bell Labs)
  - Giacomoni et al.: FastForward (U. Colorado)
  - Aldinucci et al.: FastFlow (U. Pisa)
  - *[Lê et al. SBAC-PAD'13]: WeakRB*

Lamport's Lock-Free FIFO Queue in C11

```
atomic_size_t front;
atomic_size_t back;
T data[SIZE];

void init(void) {
  atomic_init(front, 0);
  atomic_init(back, 0);
}

bool push(T elem) {
  size_t b, f;
  b = load_explicit(&back, seq_cst);
  f = load_explicit(&front, seq_cst);
  if ((b + 1) % SIZE == f)
    return false;
  data[b] = elem;
  store_explicit(&back, (b+1)%SIZE, seq_cst);
  return true;
}
```

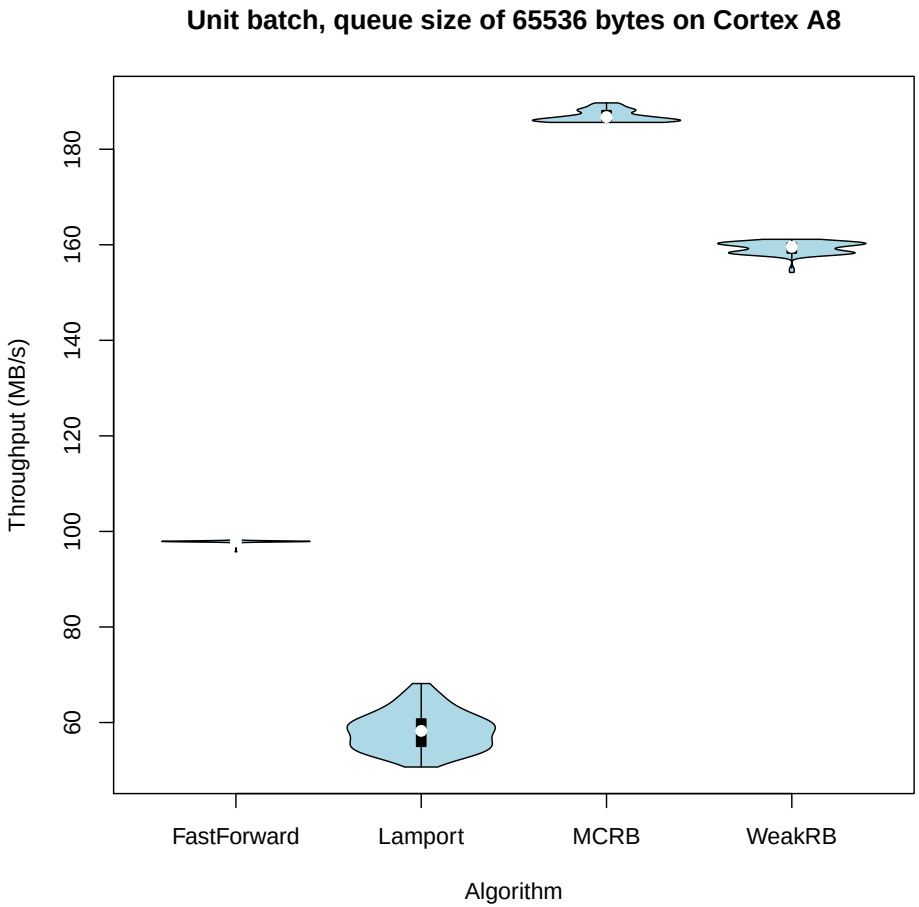
```
bool pop(T *elem) {
  size_t b, f;
  b = load_explicit(&back, seq_cst);
  f = load_explicit(&front, seq_cst);
  if (b == f)
    return false;
  *elem = data[b];
  store_explicit(&front, (f+1)%SIZE, seq_cst);
  return true;
}
```

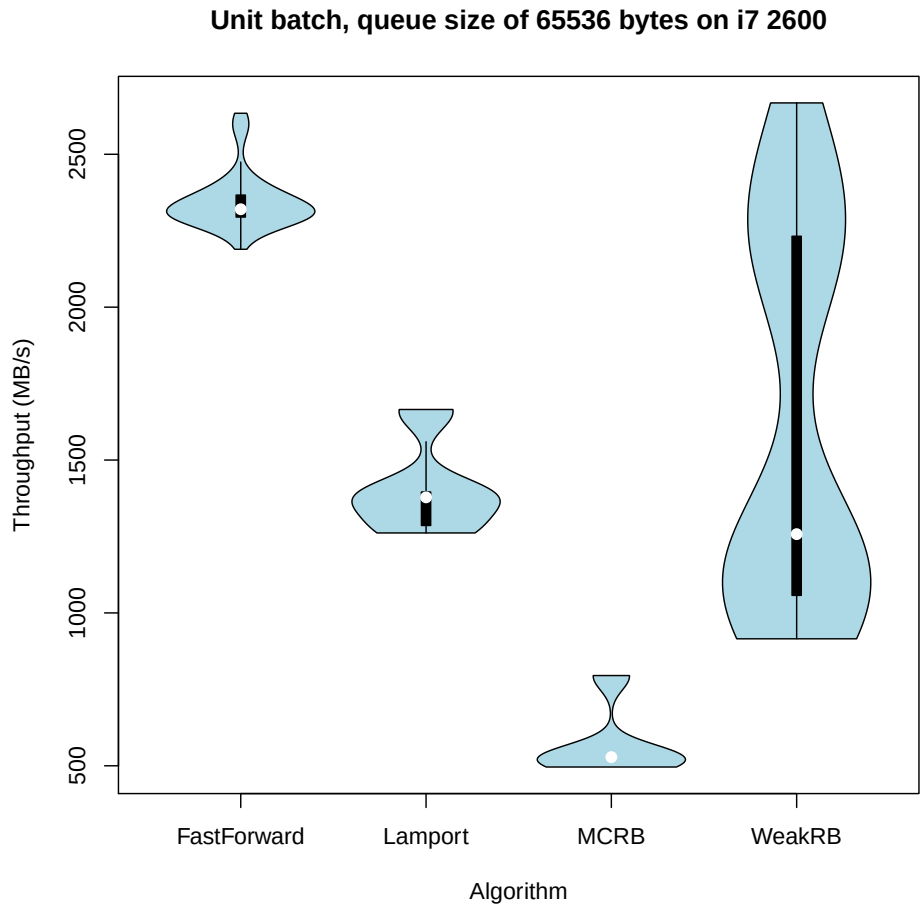
Optimization: WeakRB

```
atomic_size_t front;
size_t pfront;
atomic_size_t back;
size_t cback;

_Static_assert(SIZE_MAX % SIZE == 0,
  "SIZE div SIZE_MAX");
T data[SIZE];

void init(void) {
  atomic_init(front, 0);
  atomic_init(back, 0);
}

bool push(const T *elems, size_t n) {
  size_t b, f;
  b = load_explicit(&back, relaxed);
  if (pfront + SIZE - b < n) {
    pfront = load_explicit(&front, acquire);
    if (pfront + SIZE - b < n)
      return false;
  }
  for (size_t i = 0; i < n; i++)
    data[(b+i) % SIZE] = elems[i];
  store_explicit(&back, b + n, release);
  return true;
}
```

```
bool pop(T *elems, size_t n) {
  size_t b, f;
  f = load_explicit(&front, relaxed);
  if (cback - f < n) {
    cback = load_explicit(&back, acquire);
    if (cback - f < n)
      return false;
  }
  for (size_t i = 0; i < n; i++)
    elems[i] = data[(f+i) % SIZE];
  store_explicit(&front, f + n, release);
  return true;
}
```

Unit Push/Pop Throughput

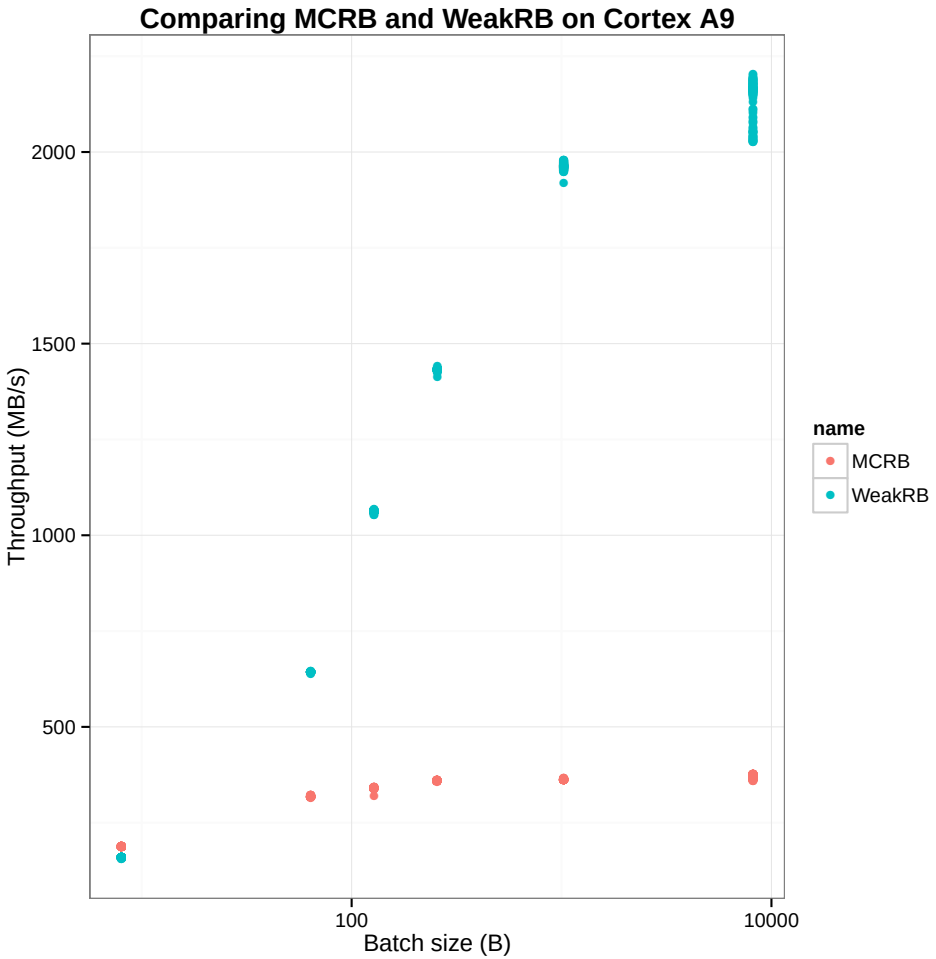


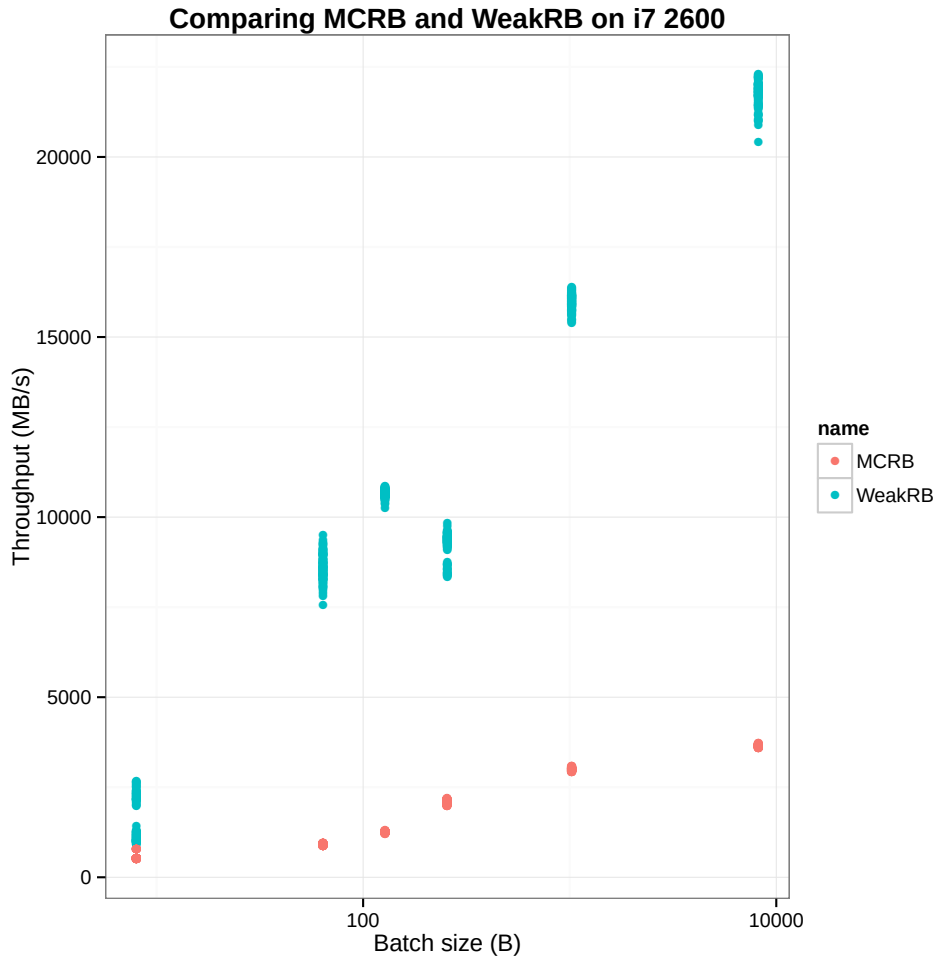


FIFOs of size 64kB, operating on 4 byte words

Very similar to the cost work-stealing, and scales linearly with the number of incoming dependences per task

Batched Synchronization Results





Ongoing work: *NK

Goal: runtime system for Kahn networks

- Solution 1: lightweight scheduling with feed-forward data flow execution model
- Solution 2: FIFOs for scalable and efficient dynamic dependence resolution and task scheduling
- Best-of-both-worlds? **NK*

  Batched FIFOs with lightweight task suspension, lock-free wake-up algorithm, and transparent caching of indices

  Formally proven relaxed memory (C11) implementation

*NK in a Nutshell

Pushing data to an SPSC FIFO with task suspension and wake-up
$\longrightarrow$ naive algorithm

```
push(Queue *q, Elem x) {
    atomic {
        if (!try_to_push_data(q, x)) { // stall
            q->pstall = true;
            return_to_scheduler();
        } else if (q->cstall)) {
            wakeup(other_end(q));
        }
    }
}
```

*NK in a Nutshell

Pushing data to an SPSC FIFO with task suspension and wake-up
$\longrightarrow$ fine-grain concurrency

```
push(Queue *q, Elem x) {
    atomic {
        if (is_full(q)) {
            q->pstall = true;
            return_to_scheduler();
        }
    }
    push_data(q, x); // always succeed
    atomic {
        if (q->cstall)
            wakeup(other_end(q));
    }
}
```

*NK in a Nutshell

Pushing data to an SPSC FIFO with task suspension and wake-up
$\longrightarrow$ sequentially consistent implementation

```
push(Queue *q, Elem x) {
    if (is_full(q)) {
        q->pstall = true;
        if (is_full(q) || CAS(q->pstall, 1 -> 0))
            return_to_scheduler();
    }
    push_data(q, x);
    if (q->cstall && CAS(q->cstall, 1 -> 0))
        wakeup(other_end(q));
}
```

*NK in a Nutshell

Pushing data to an SPSC FIFO with task suspension and wake-up
→ pseudo-code of "strict" algorithm

```
push(Queue *q, Elem x) {
    if (is_full(q)) {
        q->pstall = true;
        fence();
        if (is_full(q) || CAS(q->pstall, 1 -> 0))
            return_to_scheduler();
    }
    push_data(q, x);
    fence();
    if (q->cstall && CAS(q->cstall, 1 -> 0))
        wakeup(other_end(q));
}
```

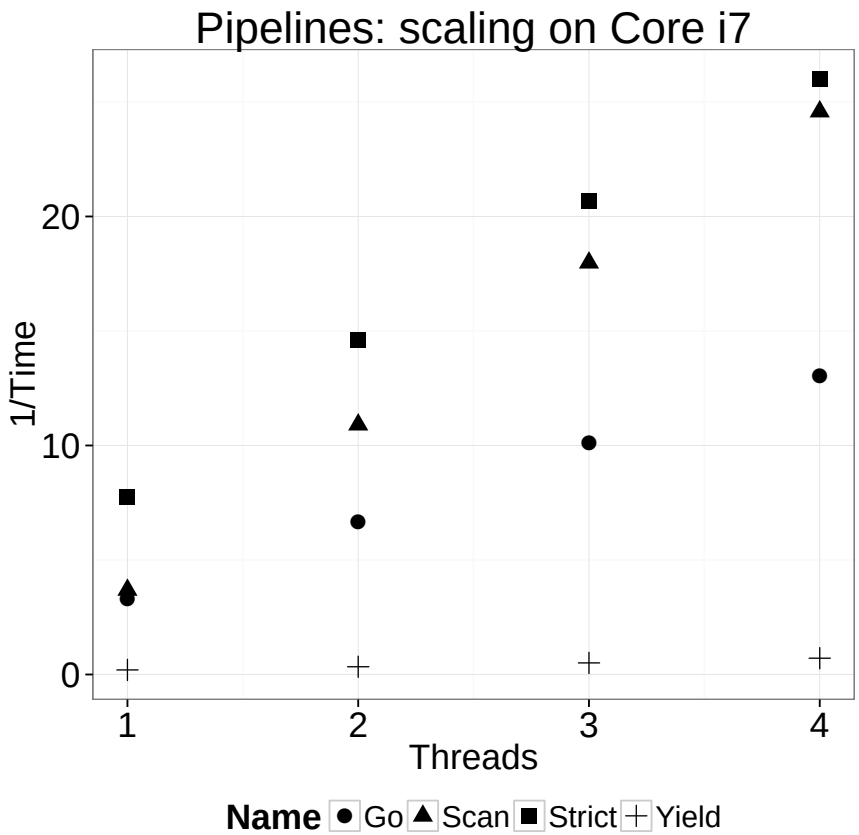
*NK in a Nutshell

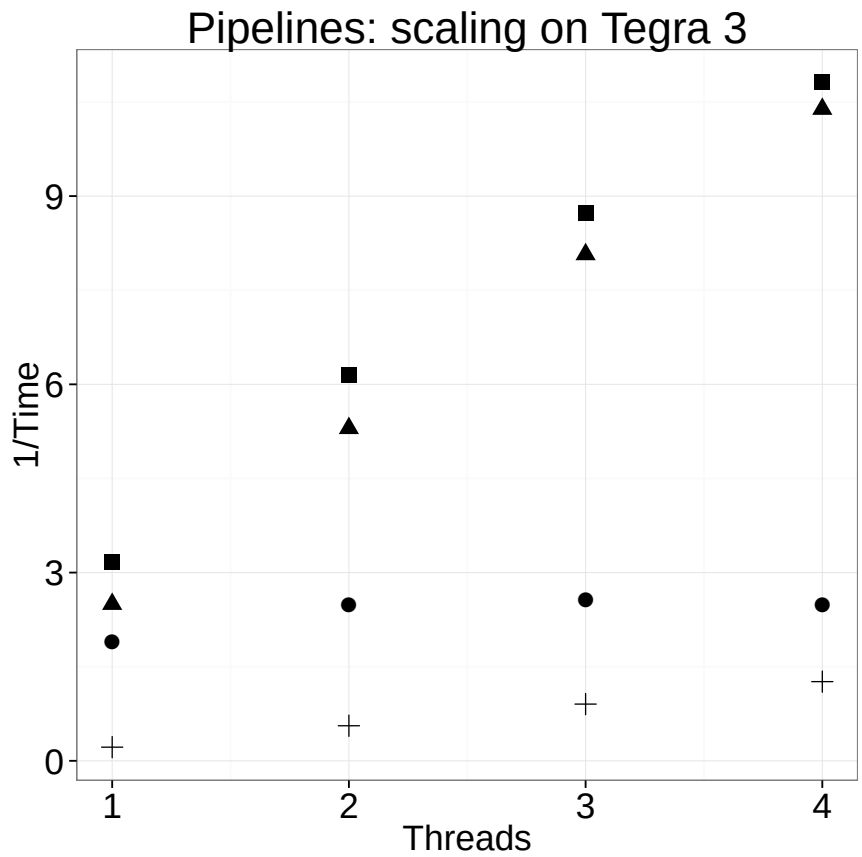
Pushing data to an SPSC FIFO with task suspension and wake-up
$\longrightarrow$ pseudo-code of "scan" algorithm

```
push(Queue *q, Elem x) {
    if (is_full(q)) {
        atomic {
            q->pstall = true;
            fence();
            foreach(Queue *oq in stalled_queues()) {
                wakeup(other_end(oq));
            }
            return_to_scheduler();
        }
    }
    push_data(q, x);
    if (q->cstall)
        wakeup(other_end(q));
}
```

*NK: Preliminary Experiments

Comparison with Go: goroutines and channels, locking on critical path
Comparison with a plain "yield" to the scheduler





4. Challenges

Challenges: Task-Level Optimizations

- *Conversion to persistent streaming processes*
  - Parallel execution of data-flow tasks with streaming constructs
- *Task data-parallelization*
  - Parallel iteration of independent activations of a task
  - Thread-level and vector parallelism
- *Task fusion and scheduling*
  - Static code generation, clock-directed
- *Task coarsening (batching)*
  - Loop nest transformation analog: strip-mining
- *Synchronization optimization*
  - Elimination of redundant push/pop
- *Copy optimization*
  - In-place operations for sliding windows, direct access to ring buffers
- *Static versus dynamic streaming*
  - Compilation for polyhedral/CSDF programs (comparisons wanted)
  - Compiler and runtime techniques for dynamic streaming programs

→ See Antoniu Pop's thesis (dynamic streaming and CDDF)
→ See Cupertino Miranda's thesis (Erbium IR and runtime)

Challenges: Dynamic Streaming Runtime

- **NK*
  - Generalization: MPMC, index-based/Erbium, split-phase, etc.
  - Port to manycore and distributed memory architectures
- *Scalability*
  - Hierarchical scheduling, topology-awareness → streaming? [Drebes et al. TACO 2014]
- *Performance debugging*
  - Andi Drebes' Aftermath: `http://openstream.info` Streaming extension?

Message

Streaming as a Language Construct

- Kahn networks: *deterministic parallelism*, task graphs, stream equations, differential equations, state machines, sliding windows, etc...
- The *compiler* plays a central role: restrict to executable specifications, task-level optimization, exposing task-level parallelism, in-place memory management
- Study the classes of synchronous and "strict" Kahn networks for parallel computing
- Intermediate representation based on streaming and synchronous language constructs (Nhat Minh Lê and Adrien Guatto)

Message

Streaming as a general-purpose execution model

- Conversion of task-level data flow to persistent streaming processes
- Index-cached FIFOs and batched communications
- Combination of compiler and runtime optimizations
- Lock-free, relaxed memory implementation of index-cached, batched FIFO communications and suspendable tasks

Many thanks to...

Andi Drebes
Adrien Guatto
Karine Heydemann
Martin Kong
Nhat Minh Lê
Robin Morisset
Antoniu Pop