# Phaser Beams: Integrating Stream Parallelism with Task Parallelism

X10 Workshop

June 4th, 2011

Jun Shirako, David M. Peixotto,

Dragos-Dumitru Sbirlea and Vivek Sarkar

Rice University

# Introduction

- **Stream Languages**
  - Natures to explicitly specify streaming parallelism in a stream graph
    - Filter (node): Computation unit
    - Stream (edge): Flow of data among filters
  - Lack of dynamic parallelism
    - Fixed stream graphs w/o dynamic reconfiguration
- **Task Parallel Languages**
  - Support of dynamic task parallelism
    - Task: Dynamically created/terminated lightweight thread
      - Chapel, Cilk, Fortress, Habanero-Java/C, Intel Threading Building Blocks, Java Concurrency Utilities, Microsoft Task Parallel Library, OpenMP 3.0 and X10
  - Lack of support for efficient streaming communication among tasks
- **Address the gap between two paradigms**
  - Phaser beams: Integrating Stream and Dynamic Task parallel models

# Introduction

- **Habanero-Java**
  - Task parallel language based on X10 v1.5
  - http://habanero.rice.edu/hj
- **Phasers in HJ**
  - Extension of X10 clocks
  - Synchronization for dynamic task parallel model
  - Various synchronization patterns
    - Collective barriers, point-to-point synchronizations
  - Java 7 Phasers
- **Streaming extensions to phasers**
  - Streaming communication among tasks
  - Adaptive batch optimization
    - Runtime cycle detection for efficient execution of acyclic stream graphs

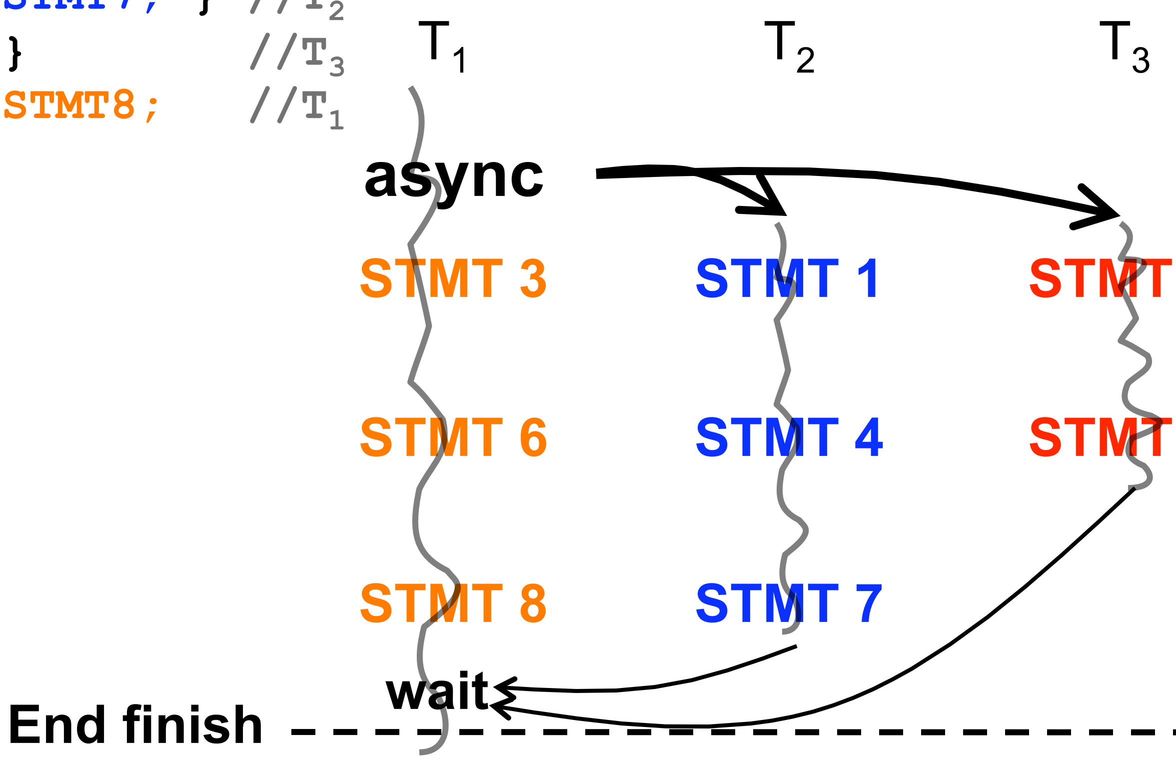
# Outline

- Introduction
- **Habanero-Java parallel constructs**
  - async, finish, phasers and accumulator
- **Extensions for streaming with dynamic parallelism**
  - Phaser beams
  - Expressed streaming patterns
- **Adaptive batch optimization**
  - Runtime cycle detection
  - Adaptive batching to avoid deadlock
- **Experimental results**
- **Conclusions**

# Task Creation & Termination

- **Async: Lightweight task creation**
- **Finish: Task-set termination**

```
finish {// Start finish

  // T1 creates T2 and T3
  async { STMT1; STMT4; STMT7; } //T2
  async { STMT2; STMT5; }        //T3
          STMT3; STMT6; STMT8;   //T1

} // End finish
```



# Phasers

- **Phaser allocation**
  - **phaser ph = new phaser(mode)**
    - Phaser **ph** is allocated with **registration mode**
    - Mode: **SIG_WAIT_SINGLE (default)**

**SIG_WAIT_SINGLE (default)**
|
**SIG_WAIT**
/ \
**SIG** **WAIT**

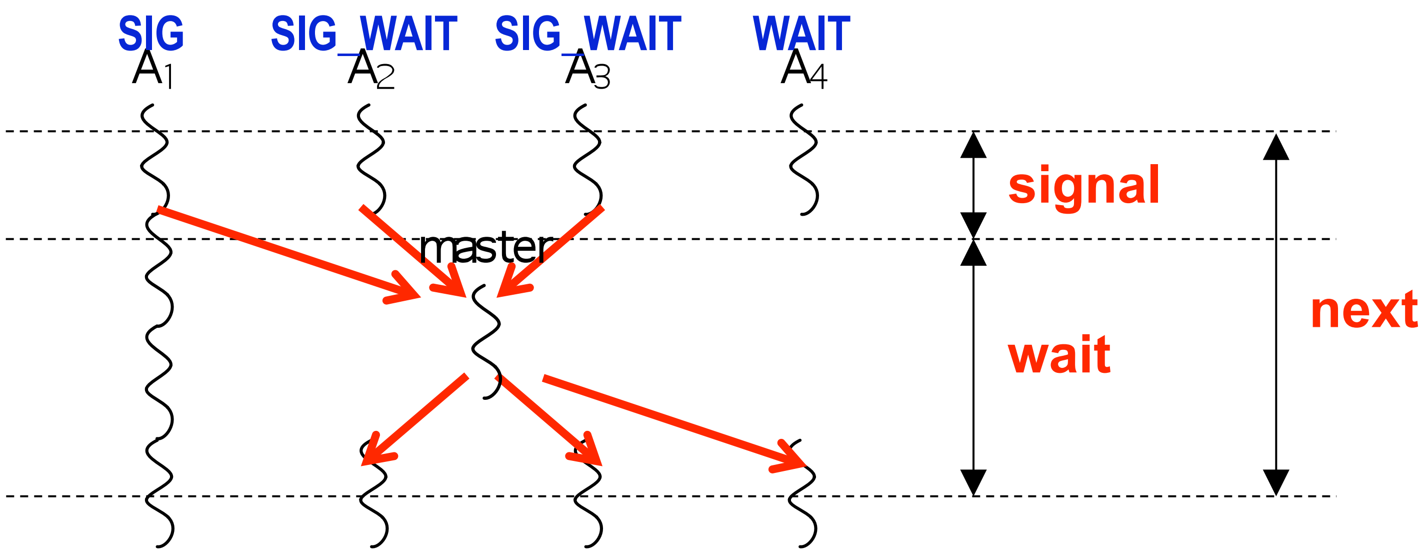
- Registration mode defines capability
- There is a lattice ordering of capabilities

- **Task registration**
  - **async phased (ph$_1$<mode$_1$>, ph$_2$<mode$_2$>, … ) {STMT}**
    - Created task is registered with **ph$_1$** in **mode$_1$**, **ph$_2$** in **mode$_2$**, …
    - Capability rule: Child task's registration mode must be subset of parent's
- **Synchronization**
  - **next:** Equivalent to **signal** followed by **wait**
    - Deadlock-free execution semantics
  - **signal:** Non-blocking operation to notify "I reached the sync point"
  - **wait:** Blocking operation to wait for other tasks' notification

# next / signal / wait

next = 
- Notify "I reached next" = signal / ph.signal()
- Wait for others to notify = wait / ph.wait()

- Synchronization semantics depends on mode
  - SIG_WAIT: next = signal + wait
  - SIG: next = signal + **no-op** (Don't wait for any task)
  - WAIT: next = **no-op** + wait (Don't signal any task)




- A master task is selected in tasks w/ wait capability
- It receives all signals and broadcasts a barrier completion notice

# Accumulators

- **Constructs for reduction combined with phaser barrier**
- **Allocation (constructor)**
  - accumulator(Phaser ph, accumulator.Operation op, Class type);
    - ph: Host phaser upon which the accumulator will rest
    - op: Reduction operation
      - sum, product, min, max, any
    - type: Data type
      - byte, short, int, long, float, double, Object (only for any)
- **Send a data to accumulator in current phase**
  - void put(Number data);
- **Retrieve the reduction result from previous phase**
  - Number get();
  - Eager vs. lazy accumulation implementations

# Phaser Accumulators for Reduction

```
phaser ph = new phaser(SIG_WAIT);
accumulator a = new accumulator(ph, accumulator.SUM, int.class);
accumulator b = new accumulator(ph, accumulator.MIN, double.class);


// foreach creates one task per iteration
foreach (point [i] : [0:n-1]) phased (ph<SIG_WAIT>) {
   int iv = 2*i + j;
   double dv = -1.5*i + j;
   a.put(iv);
   b.put(dv);

   next;

   int sum = a.get().intValue();
   double min = b.get().doubleValue();
   …
}
```

Must be SIG_WAIT / SIG_WAIT_SINGLE

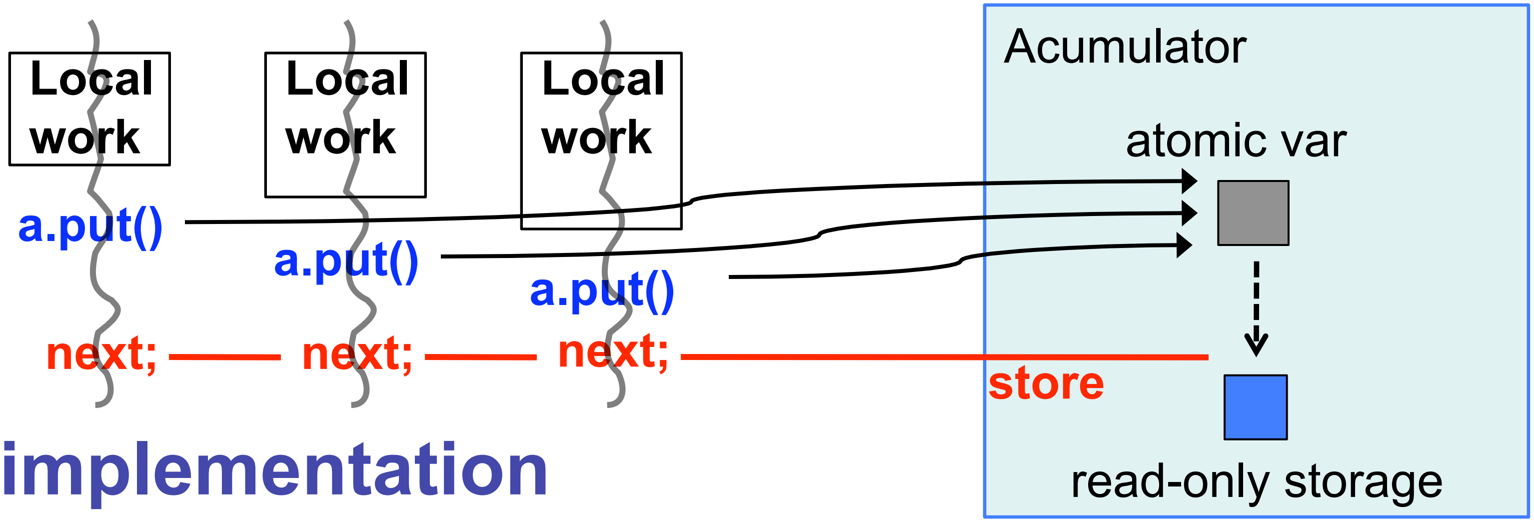
Send a value to accumulator

Barrier to advance the phase

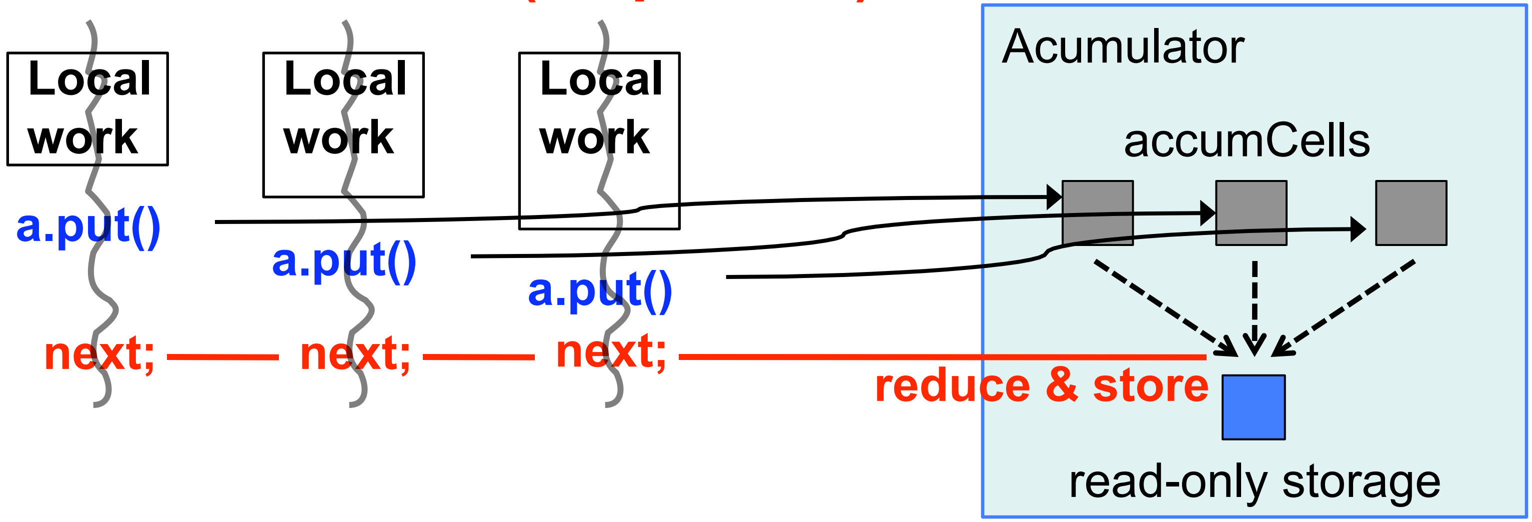
Get the result from *previous* phase (no race condition)

# Different implementations for Accumulation

- **Eager implementation**
  - Accumulation at **send (concurrent)**




- **Lazy implementation**
  - Accumulation at **next (sequential)**



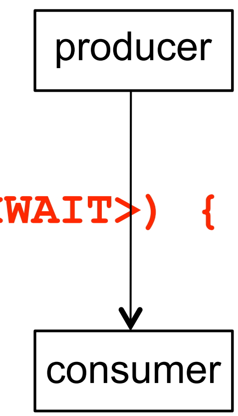
# Outline

- **Introduction**
- **Habanero-Java parallel constructs**
  - async, finish, phasers and accumulator
- **Extensions for streaming with dynamic parallelism**
  - Phaser beams
  - Expressed streaming patterns
- **Adaptive batch optimization**
  - Runtime cycle detection
  - Adaptive batching to avoid deadlock
- **Experimental results**
- **Conclusions**

# Streaming Communications

- **Producer tasks**
  - Put data on stream
  - Should go ahead of consumers
  - Tasks on phaser in SIG mode
- **Consumer tasks**
  - Get data from stream
  - Must wait for producers
  - Tasks on phaser in WAIT mode
- **Streams**
  - Manage communication among tasks
    - Keep data from producers until consumers are done
    - Limited size buffer to keep data
  - Accumulator to implement stream
    - Lock-step execution
      - Keep only a single data element
      - Tasks must be in SIG_WAIT

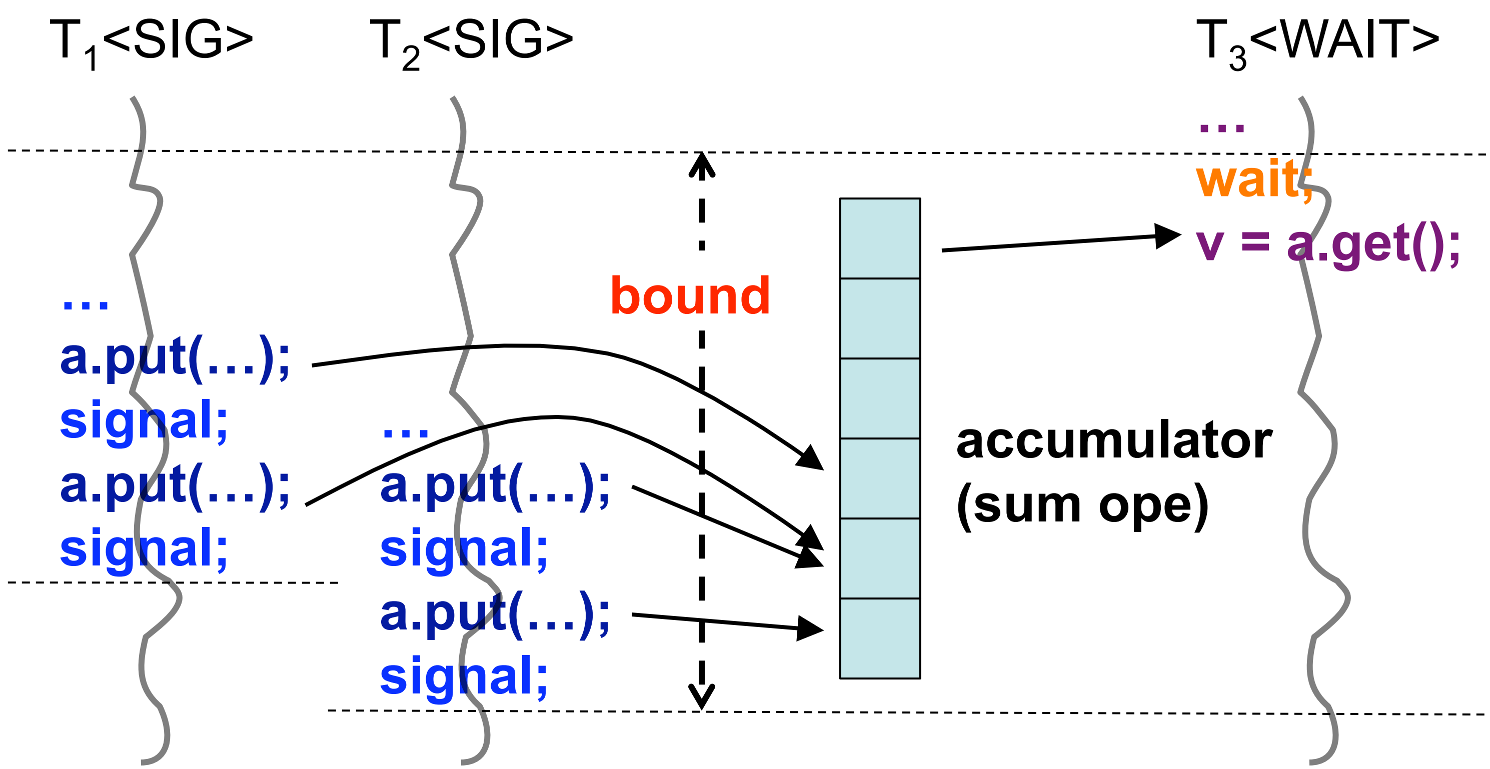
```
phaser ph = new phaser();
async phased (ph<SIG>) {
  while(...) {
    ...
    next;
    ...
} }
async phased (ph<WAIT>) {
while(...) {
    ...
    next;
    ...
} }
```



# Bounded Phaser Extensions

```
phaser ph      = new phaser(SIG_WAIT, bound);
accumulator a = new accumulator(ph, SUM, double.class);
```

- **Internal buffer to accumulator**
  - Keep multiple results from bounded number of previous phases
- **Bound constraint**
  - # wait ops ≤ # signal ops ≤ # wait ops + bound size



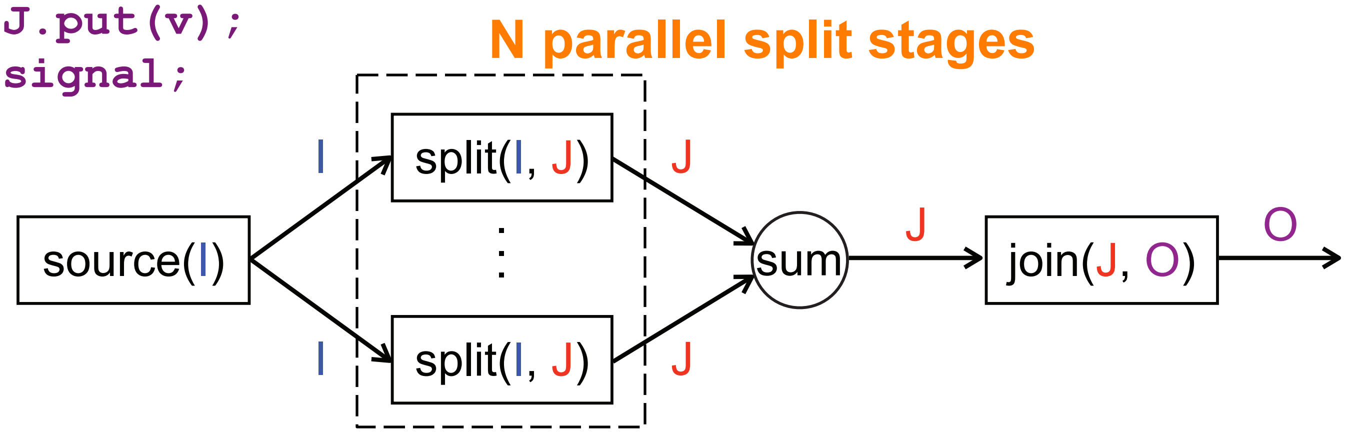
# Streaming Patterns: Pipeline

```
void Pipeline() {
    phaser phI     = new phaser(SIG_WAIT, bnd);
    accumulator I = new accumulator(phI, accumulator.ANY);
    phaser phM     = new phaser(SIG_WAIT, bnd);
    accumulator M = new accumulator(phM, accumulator.ANY);
    phaser phO     = new phaser(SIG_WAIT, bnd);
    accumulator O = new accumulator(phO, accumulator.ANY);
    async phased (phI<SIG>)                source(I);
    async phased (phI<WAIT>, phM<SIG>) avg(I,M);
    async phased (phM<WAIT>, phO<SIG>) abs(M,O);
    async phased (phO<WAIT>)               sink(O);
}
void avg(accumulator I, accumulator M) {
    while(...) {
        wait; wait;          // wait for two elements on I
        v1 = I.get(0);       // read first element
        v2 = I.get(-1);      // read second element (offset = -1)
        M.put((v1+v2)/2); // put result on M
        signal;
} }
```

$\text{source(I)} \xrightarrow{I} \text{avg(I, M)} \xrightarrow{M} \text{abs(M, O)} \xrightarrow{O} \text{sink(O)}$

# Streaming Patterns: Split-join

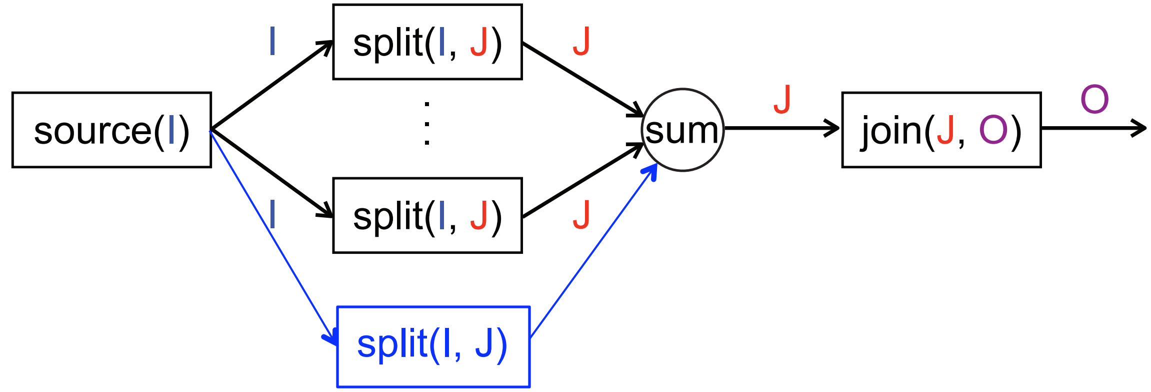
```
void Splitjoin() {
    phaser phI         = new phaser(SIG_WAIT, bnd);
    accumulator I      = new accumulator(phI, accumulator.ANY);
    phaser phJ         = new phaser(SIG_WAIT, bnd);
    accumulator J      = new accumulator(phJ, accumulator.SUM);

    async phased (phI<SIG>)                  source(I);
    foreach (point [s] : [0:N-1])
        phased (phI<WAIT>, phJ<SIG>) split(I, J);
    async phased (phJ<WAIT>)                 join(J);
}
split(I, J) {
    while(...) {
        wait;
        v = foo(I.get());
        J.put(v);
        signal;
} }
```



# General Streaming Graphs with Dynamic Parallelism

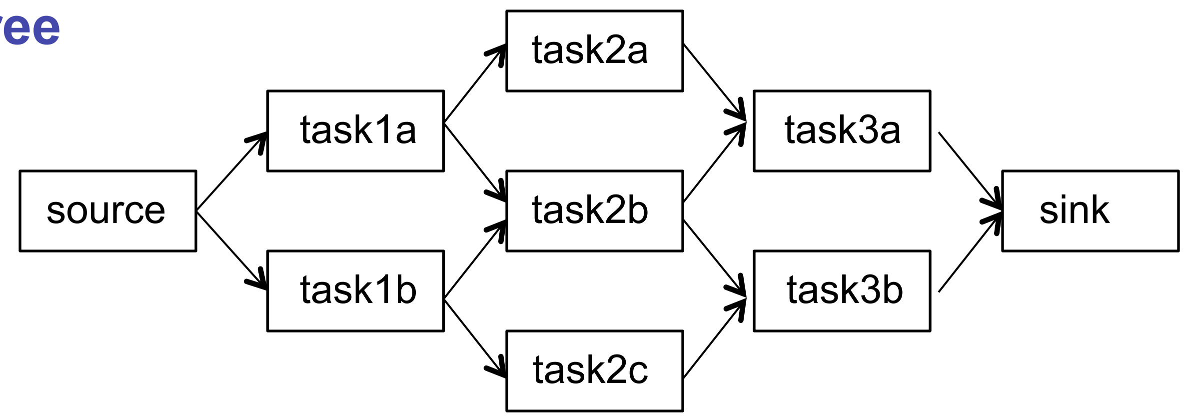
- **Dynamic split-join**

```
dynamicSplit(I, J) {
  while(...) {
    if (spawnNewNode()) async phased dynamicSplit(I, J);
    if (terminate())    break;
    wait; ...
} }
```




**stages are spawned/terminated dynamically**

- **Dynamic pipeline**
- **Tree**



# Outline

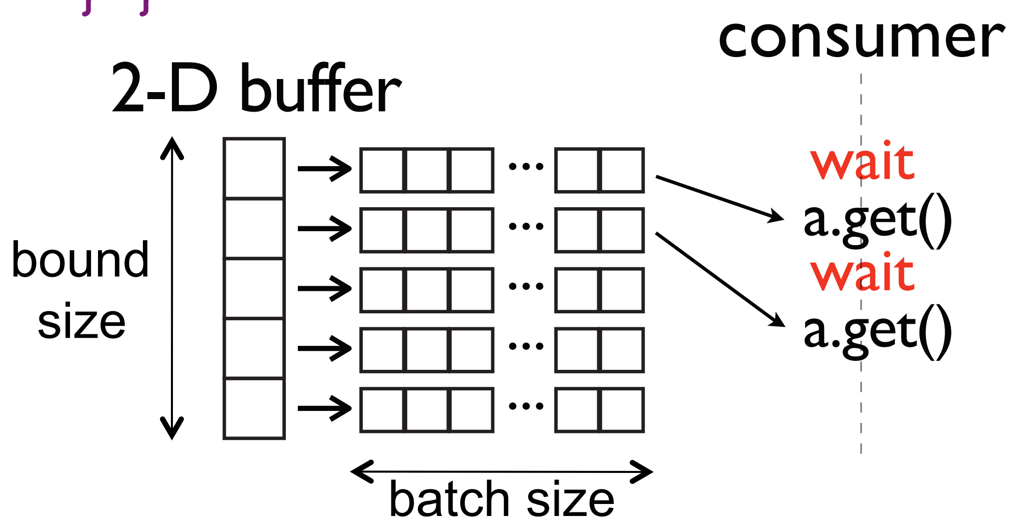
- **Introduction**
- **Habanero-Java parallel constructs**
  - async, finish, phasers and accumulator
- **Extensions for streaming with dynamic parallelism**
  - Phaser beams
  - Expressed streaming patterns
- **Adaptive batch optimization**
  - Runtime cycle detection
  - Adaptive batching to avoid deadlock
- **Experimental results**
- **Conclusions**

# Batch Optimization for Acyclic Graph

- Reduce communication overhead by factor of batch size
- Deadlock due to producer-consumer cycle

```
// Non-batched code
async phased
(ph1<WAIT>, ph2<SIG>) {
  while(...) {
    wait;
    v = foo(a1.get());
    a2.put(v);
    signal;
} }
```

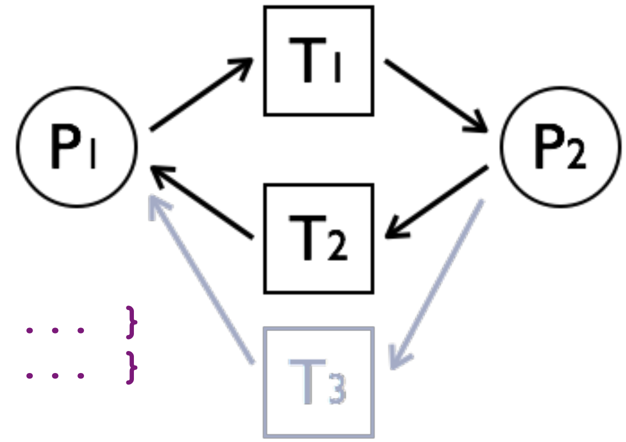
```
// Batched code
async phased
(ph1<WAIT>, ph2<SIG>) {
  while(...) {
    if (batch1.empty()) {
      wait;
      batch1 = a1.get();
    }
    v = foo(batch1.pop());
    batch2.push(v);
    if (batch2.full()) {
      a2.put(batch2);
      signal;
    }
  }
}
```



# Adaptive Batch Optimization

- **Simple cycle example**

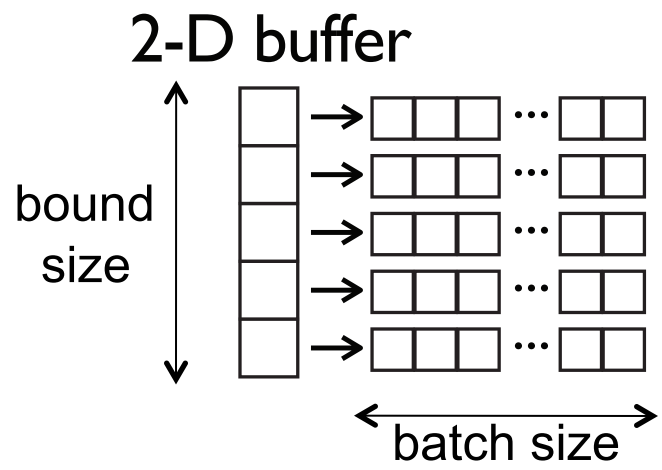
```
finish {
    // Parent (root) task create phasers
    phaser P1 = new phaser(SIG_WAIT);
    phaser P2 = new phaser(SIG_WAIT);
    async phased (P1<WAIT>, P2<SIG>) { // T1 ... }
    async phased (P2<WAIT>, P1<SIG>) { // T2 ... }
}
```




- **Adaptive batching**
  - Provide batched code and non-batched code (defined in macro)
  - Runtime cycle detection
    - D. Yellin, "Speeding up dynamic transitive closure for bounded degree graphs" , Acta Informatica, 30:369–384, 1993
  - Switch to non-batched code if cycle is detected
- **Capability rule in registration mode**
  - Child task's mode must be subset of parent task's mode
    - Child task doesn't introduce new cycle, trace only parent

# Experimental Setup

- **Platforms**
  - Intel Xeon E7330
    - 2.4GHz 16-core (4 Core-2-Quad)
  - Sun UltraSPARC T2
    - 1.2GHz 64-thread (8-core x 8-thread/core)
  - IBM Power7
    - 3.55GHz 32-core (SMT turned off)




- **Experimental variants**
  - MIT StreamIt compiler & runtime 2.1.1
    - C-based implementation
    - Always apply batch optimization (assumes acyclic stream graph)
    - Batch size = 10,000, bound = unlimited (std::queue)
  - Habanero-Java phasers
    - Java-based implementation
    - Adaptive batching (no constraint on stream graph structure)
    - Batch size = 10,000, bound = 8

# Experimental Setup

- **Microbenchmarks**
  - Push/pop microbenchmark
    - Single-producer / single-consumer
    - Throughput of streaming communication
  - Thread-ring (the Computer Language Benchmarks Game)
    - Threads are linked in a ring (cycle structure)
    - A token is passed around
    - Efficiency of runtime cycle detection
- **Application benchmarks**
  - Filterbank, FMRadio, BeamFormer (StreamIt benchmarks)
    - Acyclic graph structure
    - Static stream graph w/o dynamic parallelism
  - Sieve of Eratosthenes
    - Find prime numbers from input stream (increasing integers)
    - Dynamic pipeline / dynamic split-join

# Microbenchmarking Results

- **Push/pop: 1-producer / 1-consumer**
  - # operations per second
  - Busywait-based phaser vs. lock-based StreamIt

| | Xeon | T2 | Power7 |
|---|---|---|---|
| StreamIt (batch) | $114.0 \times 10^6$ | $21.7 \times 10^6$ | $33.1 \times 10^6$ |
| Phaser (non-batch) | $11.0 \times 10^6$ | $2.7 \times 10^6$ | $8.4 \times 10^6$ |
| Phaser (adaptive batch) | $148.2 \times 10^6$ | $24.5 \times 10^6$ | $299.4 \times 10^6$ |

- **Thread-ring: Cyclic streaming graph**
  - Average time per hop [microseconds]
  - Small overhead for adaptive batching

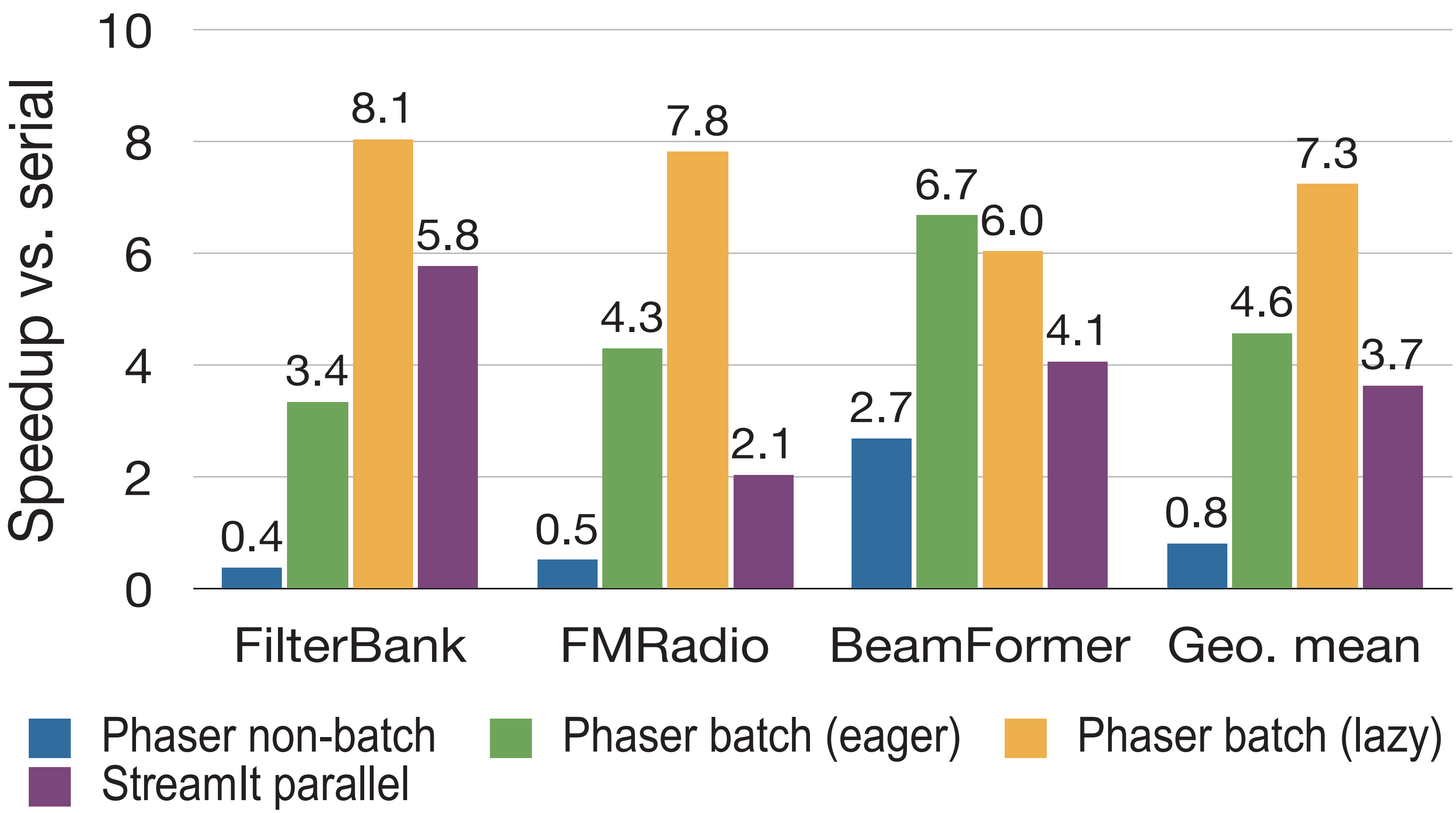
| | Xeon | T2 | Power7 |
|---|---|---|---|
| Java original | 9.4 μs | 16.3 μs | 11.9 μs |
| StreamIt (batch) | N/A | N/A | N/A |
| Phaser (non-batch) | 2.2 μs | 2.7 μs | 2.9 μs |
| Phaser (adaptive batch) | 2.2 μs | 2.7 μs | 3.0 μs |

# Summary for StreamIt Benchmarks

| Benchmark | variant | Xeon | T2 | Power7 |
|---|---|---|---|---|
| FilterBank | Java serial | 11.4 sec | 175.6 sec | 15.1 sec |
| | HJ parallel (phaser) | 1.4 sec | 23.9 sec | 3.4 sec |
| | StreamIt serial | 8.9 sec | 41.2 sec | 1.9 sec |
| | StreamIt parallel | 1.5 sec | 6.7 sec | 5.4 sec |
| FMRadio | Java serial | 25.3 sec | 288.1 sec | 26.6 sec |
| | HJ parallel (phaser) | 3.2 sec | 20.7 sec | 4.8 sec |
| | StreamIt serial | 7.6 sec | 470.3 sec | 5.9 sec |
| | StreamIt parallel | 3.7 sec | 21.2 sec | 8.0 sec |
| BeamFormer | Java serial | 19.1 sec | 258.7 sec | 20.7 sec |
| | HJ parallel (phaser) | 3.2 sec | 35.2 sec | 6.0 sec |
| | StreamIt serial | 6.4 sec | 86.8 sec | 8.9 sec |
| | StreamIt parallel | 1.6 sec | 13.4 sec | 3.5 sec |
| Geo-mean (speedup vs. Java serial) | HJ parallel (phaser) | 7.3× | 9.1× | 4.4× |
| | StreamIt serial | 2.3× | 2.0× | 4.4× |
| | StreamIt parallel | 8.5× | 19.0× | 3.8× |

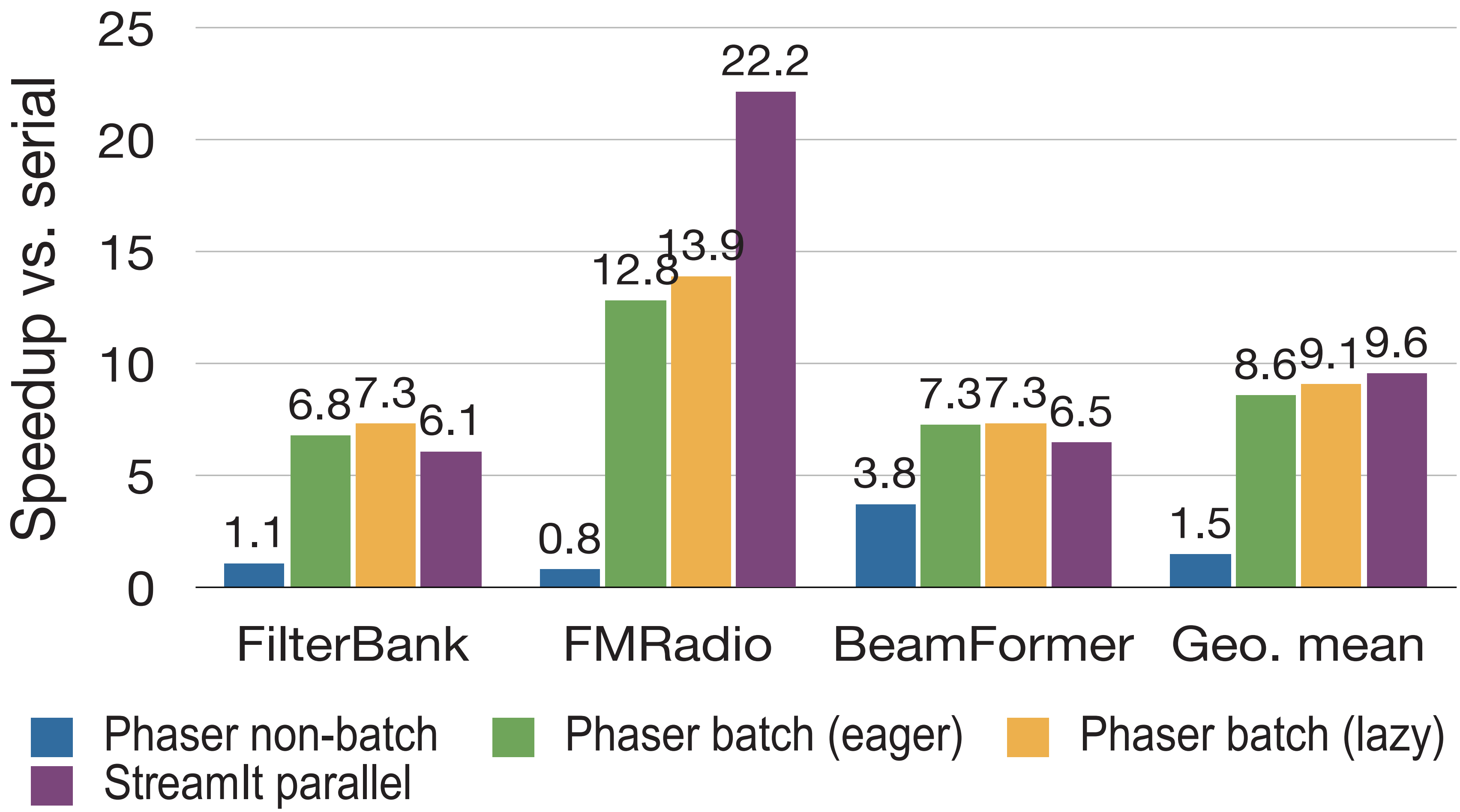
- HJ parallel: Lazy implementation policy for accumulator
- StreamIt serial (C-based): 2.0x – 4.4x faster Java serial

# Scalability (vs. each sequential base lang.) 2.4GHz 16-core Intel Xeon



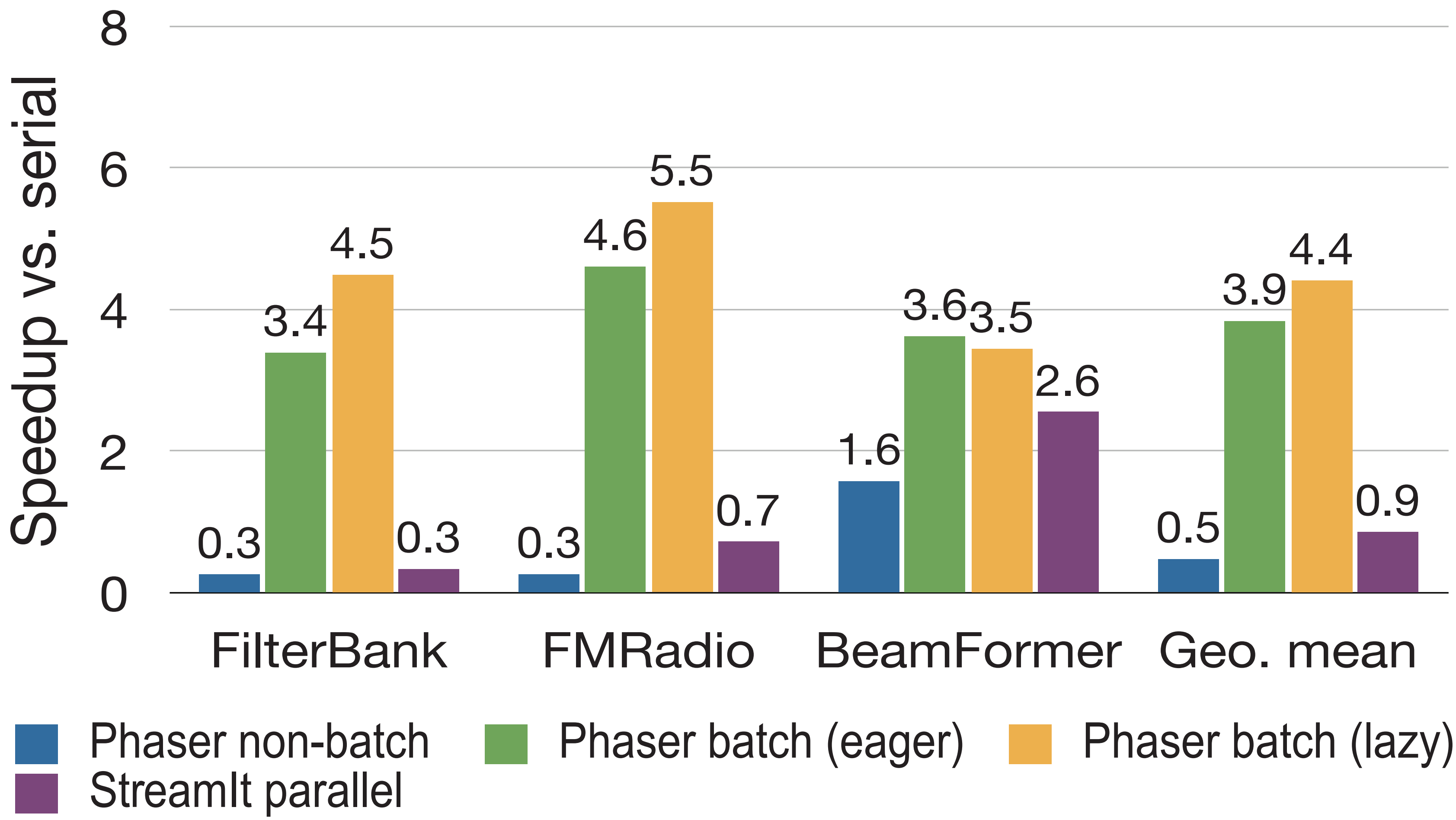

- Better scalability due to synchronization efficiency of phasers
- Accumulator implementation: Lazy policy > Eager policy

# Scalability (vs. each sequential base lang.) 1.2GHz 8-core x 8-thread/core Sun T2



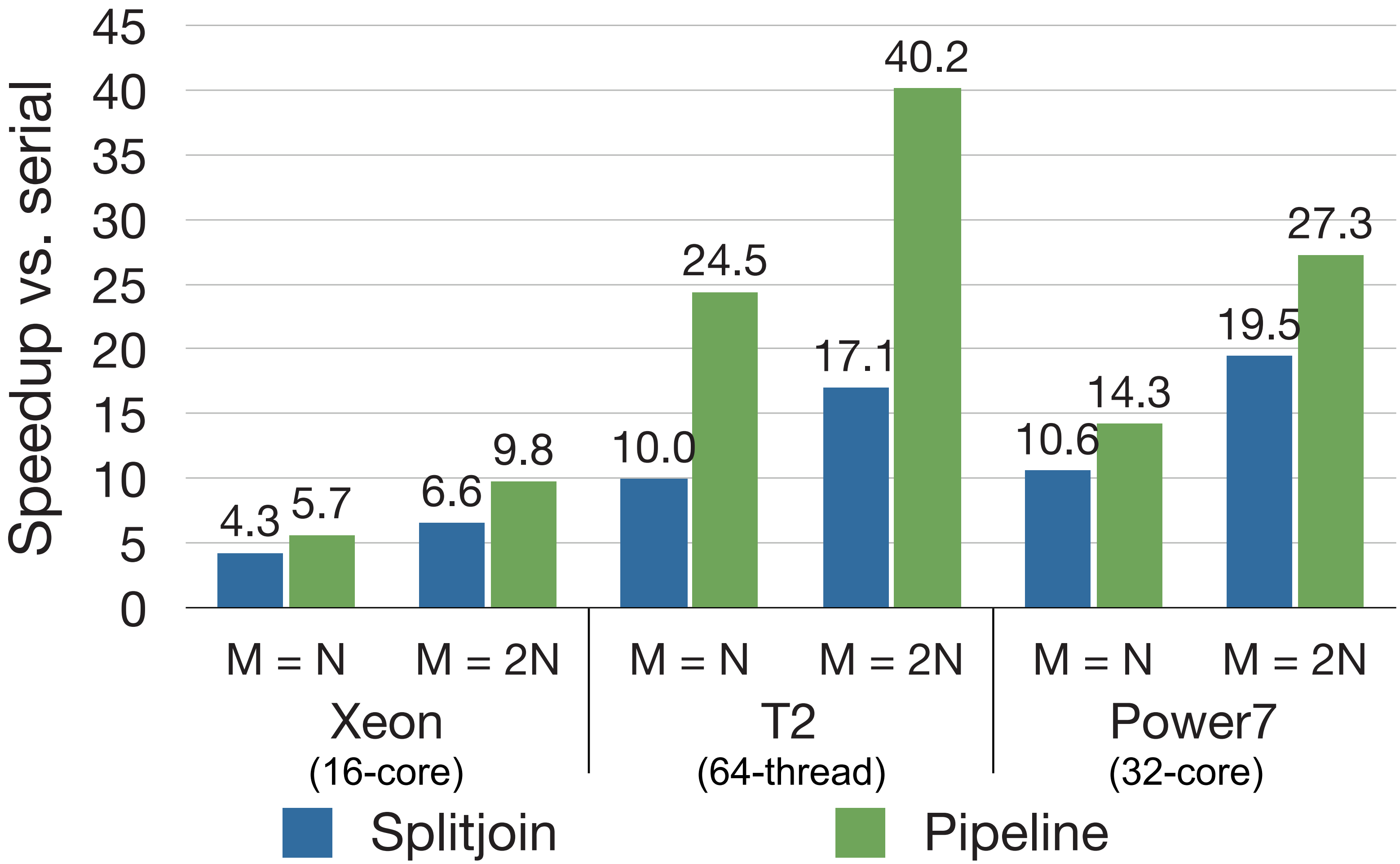

- Scalability of StreamIt is better than phasers
- Accumulator implementation: Lazy policy ≈ Eager policy

# Scalability (vs. each sequential base lang.) 3.55GHz 32-core IBM Power7




- Better scalability due to synchronization efficiency of phasers
- Accumulator implementation: Lazy policy > Eager policy

# Sieve of Eratosthenes
# (Integration of Dynamic Task and Stream Parallelism)

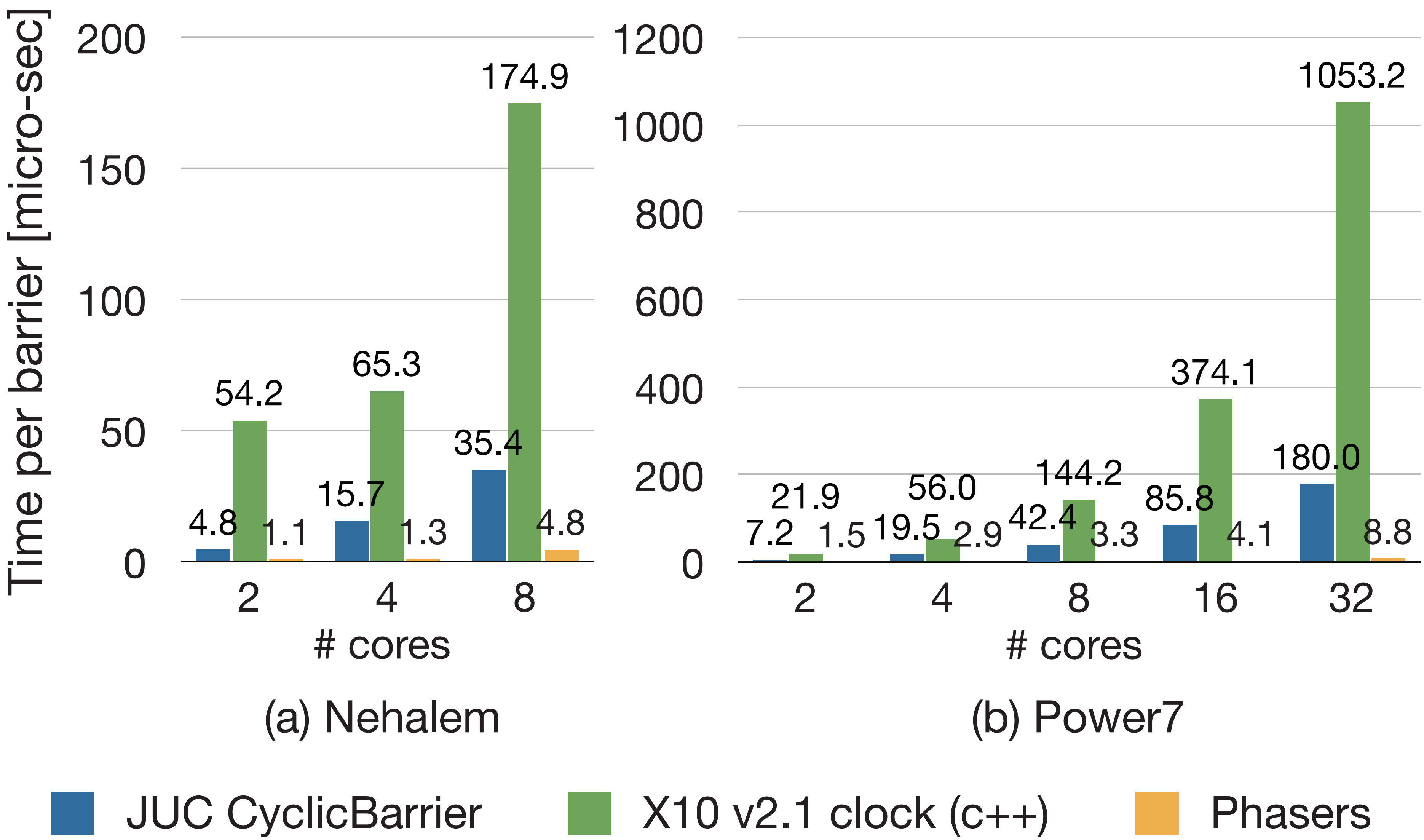



- M: Upper bound of integer in input stream
- N: Upper bound of prime number

# Conclusion

- **Phaser beams for streaming computation**
  - Integrating task and stream parallelism in a programming model
  - Adaptive batching with cycle detection
- **Experimental results on three platforms**
  - Push/pop microbenchmark (vs. C-based StreamIt)
    - 1.3x faster on Xeon, 1.1x on T2 and 9.0x on Power7
  - StreamIt benchmarks (vs. each sequential base lang.)
    - HJ phasers: 7.3x on Xeon, 9.1x on T2, and 4.4x faster on Power7
    - StreamIt: 3.7x on Xeon, 9.6x on T2, and 0.9x on Power7
  - Sieve of Eratosthenes (vs. sequential Java)
    - Up to 9.8x on Xeon, up to 40.2x on T2, and up to 27.3x on Power7
- **Future work**
  - Dynamic selection of eager or lazy policy
  - Static compiler optimizations, e.g., batch code generation and graph partitioning
  - Support of phaser functionality in X10 programming language

# Barrier Performance of CyclicBarrier, Clocks and Phasers




(a) Nehalem

(b) Power7

- Nehalem: Intel Corei7 2.4GHz 2 quad-core processor
- Power7: IBM Power7 3.55GHz