Integrating Stream and Task Parallelism in a Dataflow Programming Model

Dragoș Sbîrlea, Jun Shirako, Ryan Newton, Vivek Sarkar

Rice University, Indiana University
Motivation

• The dataflow Concurrent Collections model has no streaming optimizations:
  – Memory management (garbage collection)
  – Task management overhead
  – Data synchronization
  + Generality

• Streaming languages
  – Lack the expressivity of dataflow model
  – No dynamic parallelism
  + Performance
Motivation

• The long term target:
  – a CnC runtime for dataflow languages that would integrate the best of both worlds: *streaming parallelism* where possible and advantageous as well as *task based parallelism*. 
Contributions

• CnC API that allows efficient streaming
  – The corresponding runtime implementation

• Algorithm for automatic transformation of CnC programs to streaming CnC API

• Extensions to the streaming model to better exploit parallelism (dynamic split/join)

• Proof that performance can be obtained
  – Up to 40x performance improvement compared to task based parallel execution
Outline

1. Background:
   – The Streaming Model
   – The Concurrent Collections (CnC) model
   – Habanero Java

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Converting CnC programs to Streaming CnC

5. Related work

6. Conclusions and future work
Outline

1. Background:
   – The Streaming Model
   – The Concurrent Collections (CnC) model
   – Habanero Java
2. Streaming Concurrent Collections (SCnC)
3. Dynamic parallelism for SCnC
4. Converting CnC programs to Streaming CnC
5. Related work
6. Conclusions and future work
Streaming

- Increasingly prevalent
  - Streaming video: 39% of internet traffic
  - 90% of compute cycles to be spent on streaming by 2000

- Pattern of applications with:
  - **Streams**: unbounded, virtually infinite set of items
  - Each item is used for a **limited lifetime**
  - Independent **filters** process streams by using queue operations:
    - Pop
    - Peek
    - Push
Examples taken from StreamIt programs, and re-drawn for this presentation
Outline

1. Background:
   - The Streaming Model
   - The Concurrent Collections (CnC) model
   - Habanero Java

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Converting CnC programs to Streaming CnC

5. Related work

6. Conclusions and future work
Habanero Java Constructs

• Habanero constructs used
  
  – **Async** and **finish** for tasks management

  – **Streaming phasers** for streaming support
    • **Phasers** and **accumulators** for synchronization and data communication
Phaser & Accumulator example: producer-consumer

```java
finish {
    final phaser ph = new phaser(SIG_WAIT);
    final accumulator a = new accumulator(ph);

    async phased (ph<SIG>) { //Producer
        while(moreWorkToDo()) {
            a.put(doSomeWork());
            ph.signal();
        }
        a.put(doneWithWork());
        ph.signal();
    }
    a.put(doneWithWork());
    ph.signal();
}

async phased (ph<WAIT>) { //Consumer
    while(true) {
        ph.doWait();
        if(!doSomeOtherWork(a.get()))
            break;
    }
}
```

Streaming phasers and accumulators:

- #wait operations \(\leq\) #signal operations \(\leq\) #wait operations + k
- bounded buffer of size k
Streaming phasers restrictions

• Phaser registration restriction:
  – Registration only at task creation time
  – On subset of phasers owned by parent

• Multiple producers restriction:
  – Reduction operation used to obtain a single result
Outline

1. Background:
   - The Streaming Model
   - The Concurrent Collections (CnC) model
   - Habanero Java

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Converting CnC programs to Streaming CnC

5. Related work

6. Conclusions and future work
Implementation of Streaming CnC

Streaming CnC – CnC API variant that can be run with a streaming runtime, as opposed to the task based runtime.

Goals:
• Variable number of input/output items for each step
• Deterministic, just like CnC
• Efficient in execution time and memory usage

<table>
<thead>
<tr>
<th>Concurrent Collections</th>
<th>Streaming</th>
<th>Streaming CnC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item collection</td>
<td>Stream</td>
<td>(phaser, accumulator)</td>
</tr>
<tr>
<td>Control collection</td>
<td>Stream</td>
<td>(phaser, accumulator)</td>
</tr>
<tr>
<td>Step collection</td>
<td>Filter</td>
<td>async with a loop</td>
</tr>
</tbody>
</table>
SCnC’s Restrictions to the CnC model

- Number of producers and consumers in SCnC versus CnC

<table>
<thead>
<tr>
<th></th>
<th>CnC</th>
<th>SCnC</th>
<th>Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Item collections</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producers</td>
<td>N</td>
<td>1</td>
<td>N (streaming)</td>
</tr>
<tr>
<td>Consumers</td>
<td>N</td>
<td>N (streaming, in sync)</td>
<td>N (streaming)</td>
</tr>
<tr>
<td><strong>Steps Collections</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sources</td>
<td>N</td>
<td>N (streaming)</td>
<td>1</td>
</tr>
<tr>
<td>Destinations</td>
<td>N</td>
<td>N (streaming)</td>
<td>1</td>
</tr>
</tbody>
</table>
SCnC base class for a step

```java
public void start(WrappedInt tag) {
    async phased {
        prescribingControlCollection.ph<phaserMode.WAIT>,
        producedItemCollection1.ph<phaserMode.SIG>,
        producedControlCollection1.ph<phaserMode.SIG>
    }
    run(tag);
}

public void run(WrappedInt ptag) {
    WrappedInt tag = null;
    // get a new tag from the prescribing control collection
    tag = prescribingControlCollection.Get();
    while (tag.value != prescribingControlCollection.endStream) {
        // the step function is written by the user
        step(tag);
        // get the next control tag used in the next iteration
        // get call includes phaser operations
        tag = prescribingControlCollection.Get();
    }
}
```
Item Collection implementation

Base class for collections with a phaser and accumulator. Put and get semantics changed compared to CnC:

• Put
  • Adds to the top of the queue

• Get
  • Access history instead of future items

public void Put (Object p) {
  a.send(p);
  ph.signal();
}

public Object Get (int no) {
  Object value = null;
  if (no == 0) {
    ph.doWait();
    value = a.objResult();
  } else {
    value = a.objResult(no);
  }
  return value;

Results without dynamic parallelism

• Filterbank and Beamformer are 2 StreamIt benchmarks
• Tests run on Xeon 16 core system
• Results scaled to CnC throughput
• CnC
  – Runs out of memory on input size 1/10 of the size used for SCnC
  – Execution time dominated by garbage collection for large input sizes
Outline

1. Background:
   – The Streaming Model
   – The Concurrent Collections (CnC) model
   – Habanero Java

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Converting CnC programs to Streaming CnC

5. Related work

6. Conclusions and future work
Streaming with dynamic parallelism

• Parallelism to be exploited: between loop iterations of a single step collection
  – one step has to use more than one async.

• Programmer controls this through place tags.
  – Tags placed in control collections have 2 dimensions <place, value>:
    • A place tag, representing the id of the async on which that tag should be processed
      – New asyncs created on the fly (dynamic parallelism)
    • The actual tag value
  – The mechanism can be made completely transparent for stateless filters
Eratosthenes Sieve

- Same Xeon 16 core system
- Number of Filters dynamic, up to 16 on average
Outline

1. Background:
   – The Streaming Model
   – The Concurrent Collections (CnC) model
   – Habanero Java

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Converting CnC programs to Streaming CnC

5. Related work

6. Conclusions and future work
CnC to Streaming CnC

• Graph shape analysis and transformation
  – Needed for the streaming access pattern analysis
  – Needed because of the runtime requirement (single producer)
  – Additional shapes supported through transformations

• Access patterns checking
  – Streaming access pattern is restrictive

• Mapping on Streaming CnC
  – Transformation of tags from CnC to SCnC
Streaming access pattern checking

- Need to model the direct relationship between the environment and item collections
- Performed through producer and consumer functions $f_c$ and $f_p$
- Composition of control put functions ending with an item put or item consume function

$$f_{\text{producer}} = f_{\text{ip}} \circ f_2 \circ f_{\text{cp}} \circ f_1 \circ f_{\text{cp}} \circ f_0$$

$$f_{\text{consumer}} = f_{\text{lc}} \circ f_3 \circ f_{\text{cp}} \circ f_0$$
Streaming access patterns

- **Producer precedence:** for any $y$ and any producer and consumer function of the same item collection
  $$f^{-1}_p(y) \leq f^{-1}_c(y)$$

- **Sliding window:**
  $$f_{c\min}(x) \leq f_{c\min}(x+1)$$

minimum consumer function is:
$$f_{c\min}(x) = \min(f_{c\ 1}(x), f_{c\ 2}(x), ...),$$
for any $x$ and a particular item collection
Streaming access patterns

- **Bounded buffer**: for any two consumer/producer functions fc1 and fc2, for any $x$, there is an integer $N$ such that:

$$|(f_{c1} - f_{c2})(x)| < N$$
Analysis

• Legality
  – Need to extract or to express tag function
  – Need mathematical operations
  – Graph reshaping needs transformations to the step code in some cases

• Profitability
  – Depends on application size (number of filters)
  – Too many: context switching
  – Too few: need to use dynamic split join
  – Reshaping the graph to be well-formed adds additional overhead (space and synchronization)
### Related work:

#### StreamIt comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>SCnC</th>
<th>StreamIt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streams per filter</td>
<td>Unlimited number</td>
<td>1</td>
</tr>
<tr>
<td>Input rate</td>
<td>Variable</td>
<td>Fixed (v1.0)</td>
</tr>
<tr>
<td>Parallelism</td>
<td>Dynamic</td>
<td>Fixed</td>
</tr>
<tr>
<td>Custom split-joins</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Messaging</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Reinitialization</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Software pipelining</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Other optimizations</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Skill reuse</td>
<td>Yes*</td>
<td>No</td>
</tr>
</tbody>
</table>
Conclusion

• Proposed SCnC – CnC API variant that can be implemented through streaming.

• Proposed an algorithm to convert a CnC application to a SCnC API

• Shown proof of efficient execution and memory management through a streaming some CnC apps.

• Added features that improve the flexibility of the classic streaming model (dynamic parallelism)
Future work

• Tight integration with task based runtime

• Implement automatic transformation from CnC to SCnC

• Folding the dynamic single assignment memory space as generalization of streaming
  – Streaming = folding in space and computation
BACKUP SLIDES START HERE
Acknowledgements

• “Phaser beams: Integrating task and streaming parallelism”
  – Jun Shirako, David Peixotto, Dragos Sbirlea, Vivek Sarkar
  – X10 Workshop colocated with PLDI 2011

• “Phasers: a unified deadlock-free construct for collective and point-to-point synchronization”
  – J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer
  – ICS 2008

• “Phaser accumulators: A new reduction construct for dynamic parallelism”
  – J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer
  – IPDPS 2009

• Habanero group
The streaming domain

- Increasingly prevalent
  - Streaming video: 39% of internet traffic
  - 90% of compute cycles to be spent on streaming by 2000

- Widely applicable
  - Embedded systems
    - Cellphones, DSPs
  - Desktop applications
    - Real-time encryption
    - Compression
    - Graphics
  - Servers
    - Software routers
    - Cell phone base stations

- Summary:
  - Streaming research can have large impact
  - Being able to stream more programs might take advantage of a lot of existing stream-optimized infrastructure
Finish-Async example

```javascript
finish {
    async { // A
        ...
    }
    async { // B
        ...
        async { // B1
            ...
        }
        async { // B2
            ...
        }
    }
}
```
Phaser & Accumulator example: producer consumer

```java
finish {
    final phaser ph = new phaser(SIG_WAIT);
    final accumulator a = new accumulator(ph);

    async phased (ph<SIG>) { //Producer
        while(moreWorkToDo()) {
            a.put(doSomeWork());
            ph.signal();
        }
        a.put(doneWithWork());
        ph.signal();
    }

    async phased (ph<WAIT>) { //Consumer
        while(true) {
            ph.doWait();
            if(!doSomeOtherWork(a.get()))
                break;
        }
    }
}
```

Streaming phasers and accumulators:
- #wait operations ≤ #signal operations ≤ #wait operations + k
- bounded buffer of size k
Static Task Creation Graph

Task of type A

Tasks of type B → Tasks of type C

Tasks of type D → Tasks of type E
Make shared data explicit
Make control explicit

Shared Variables AB → Task of type A → Shared Variables AC

Tasks of type B → Tasks of type C

Shared Variables BD → Tasks of type D → Tasks of type E

Shared Variables DE

- Item Put edge
- Item Get edge
- Control Put edge
- Spawn edge
Make environment explicit
CnC Building Blocks

Environment

Task of type A

Tasks of type B

Tasks of type D

Tasks of type E

Shared Variables AB

Shared Variables BD

Shared Variables DE

Shared Variables AC

Shared Variables CE

Environment Variables
CnC Item Collections

• Collections of values called “items”
• Items are dynamic single assignment
  • Item tags are used to index into the item collections
  • No data races

• Two operations
  – Put(tag, item)
  – Get(tag)
CnC Control Collections

- Collections of values that control the execution of steps

- A Control collection is bound to one or more Step Collections through a prescription edge
  - Step Collection provides the implementation of the function to be executed
  - Control Collection provides the control tag that specifies the control flow through the code of the function
CnC Step Collections

- Step collections are functions

- Step instances are pairs (function, control tag)
  - The control tag is the parameter that dictates control flow through the function and thus data (item) access
  - A single function execution
The Streaming CnC (SCnC) runtime uses the Habanero streaming phasers primitive.

- Will dictate some of the restrictions needed for SCnC

- Phasers = collective synchronization construct
- Accumulators = reduction mechanism
Dynamic Split Join

- S1 place 0
  - phO<SIG>, accO
  - phT<WAIT>, accT
  - <place0, _>

- S1 place 1
  - phO<SIG>, accO
  - phT<WAIT>, accT
  - <place1, _>

- ItemCollection
  - phO<WAIT>, accO

- S2
Graph Shape Transformation: multiple prescriptions

Before:

- Prescribing Step Collection
  - Control Collection 1
    - Prescribed Step Collection 1
    - Prescribed Step Collection 2
    - Prescribed Step Collection 3

After:
Graph Shape Transformation: Multiple consumers

Before:

Producer Step → Item Collection → Consumer Step 1

Producer Step → Item Collection → Consumer Step 2

...

Producer Step → Item Collection → Consumer Step N

After:

Producer Step → Item Collection 1 → Consumer Step 1

Producer Step → Item Collection 2 → Consumer Step 2

...

Producer Step → Item Collection N → Consumer Step N
Graph Shape Transformation: Environment

Before:

Step 1 → Item Collection 1 → Step 3

Step 2

Item Collection 2 → Step 4

Item Collection 3

Control Collection 2

Control Collection 1

Environment

After:

Step 1 → Item Collection 1 → Step 3

Step 2

Item Collection 2 → Step 4

Item Collection 3

Control Collection 2

Control Collection 1

Entry Step

Entry Control Collection

Environment
Graph shape

• Restrictions on graph shape
  – Item and control collections are single producer
  – Single entry from environment
  – Item collections are single consumer
  – Control collections are single prescription

• Algorithm
  • Performs one transform per restriction
  • May fail some transforms
Graph Shape Transformation
Example: Multiple producers

Before:

Producer Step 1
Producer Step 2
... 
Producer Step N

Item Collection

After:

Producer Step 1 → Item Collection 1
Producer Step 2 → Item Collection 2
... 
Producer Step N → Item Collection N

Join Control Collection
Join Step

Item Collection
Case Study: FacilityLocation

- Facility Location is a streaming application that does clustering
- We implemented the online, randomized algorithm
- Static parallel version uses 4 cores only
Static parallelism version

- FacilityLocation Throughput graph
  - 16 core Xeon system
  - Streaming version without dynamic parallelism uses 4 cores
  - No dynamic parallelism
  - Normalized to CnC throughput

![FacilityLocation Throughput](chart.png)
Dynamic parallelism version

• Normalized to static parallel SCnC from previous slide

• Artificial, variable delay added to simulate real statistics computations

![Dynamic parallelism Throughput for FacilityLocation](image)

- Throughput (Scaled w.r.t. static SCnC with same delay)
- none
- 1ms every 50th
- 1ms every 25th
- 1ms every 12th

55
class FIRFilter extends Filter {
    float[] weights;
    void init(float[] weights) {
        SetInput(Float.TYPE);
        setOutput(Float.TYPE);
        setPush(N);
        setPop(1);
        setPeek(N);
        this.weights = weights;
    }
    void work() {
        float sum = 0;
        for (int i=0; i<N; i++)
            sum += input.peek(i)*weights[i];
        input.pop();
        output.push(sum);
    }
}

class Main extends Pipeline {
    void init() {
        add(new DataSource());
        add(new FIRFilter(N));
        add(new Display());
    }
}

Related Work: StreamIt
References
