SCnC: Efficient Unification of Streaming with Dynamic Task Parallelism

Dragoș Sbîrlea, Jun Shirako, Ryan Newton*, Vivek Sarkar

Rice University

* Indiana University
Stream-parallel and task-parallel paradigms can be integrated in a single programming model with dataflow as a foundation, and this integration can be achieved with high programmability and high resource efficiency.
The long term target of the project:
a runtime that would integrate the best of both worlds: \textit{streaming parallelism} where possible and advantageous as well as \textit{task based parallelism}. 
Motivation

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

• Streaming languages
  – Lack the expressivity of dataflow model
  – No dynamic parallelism
Contributions

• Streaming runtime for restricted class of CnC programs (SCnC)

• Algorithm for identification of SCnC programs

• Extensions to the streaming model to better exploit dynamic parallelism

• Experimental results showing up to 40x throughput increase
1. Background:
   - The Streaming Model
   - The Concurrent Collections (CnC) model
   - Habanero Java

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Related work

5. 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. Related work

5. Conclusions and future work
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

* S. Rixner et al., “A bandwidth-efficient architecture for media processing”, MICRO 31
What is streaming

• 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
1. Background:
   - The Streaming Model
   - The Concurrent Collections (CnC) model
   - Habanero Java
2. Streaming Concurrent Collections (SCnC)
3. Dynamic parallelism for SCnC
4. Related work
5. Conclusions and future work
Intel Concurrent Collections (CnC)

• Macro-dataflow programming model for parallel applications

• Can express arbitrary task graph with arbitrary data dependencies
void step(
  Object controlTag,
  InputItemCollection ic1,
  InputItemCollection ic2
  OutputItemCollection oc) {
  Key1 = foo(controlTag);
  Key2 = bar(controlTag);
  item1 = ic1.get(key1);
  item2 = ic2.get(key2);
  result = foobar(item1, item2);
  Oc.put(key3, result);
CnC Application Graph

Environment

Shared Variables AB

Task of type A

Environment Variables

Shared Variables AC

Tasks of type B

Shared Variables BD

Tasks of type C

Shared Variables CE

Tasks of type D

Shared Variables DE

Tasks of type E

- Item Collection
- Step Collection
- Control Collection
- Item Put edge
- Item Get edge
- Control Put edge
- Spawn edge
Habanero CnC Runtime

• Most CnC runtimes are task based
  – Intel: C++, Haskell
  – Rice: Habanero Java(HJ), .NET, Habanero C

• HJ CnC
  – User written step functions that plug into auto-generated code for item and control collections
  – Based on Habanero Java
    • Each step instance is an `async` (task)
    • `ConcurrentHashMap` for item collections
    • `Locking` for extra synchronization, if needed
CnC Summary

• High level, textual specifications language
  – Orthogonal to the programming language used
  – Makes tasks explicit
    => coordination language
  – Makes values explicit, driven by data
    => dataflow language
    => can still express any arbitrary task graph with arbitrary data dependencies

• Single assignment items
  – Determinism
Outline

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

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Related work

5. Conclusions and future work
Habanero Java Constructs

• Habanero Java (HJ) was preferred because of
  – The work stealing/work sharing scheduler
    • Async and finish for task management
  
  – Primitives for synchronization and communication
    • Phasers and accumulators for synchronization and data communication
HJ constructs 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 extension:
• #wait operations ≤ #signal operations ≤ #wait operations + k
• bounded buffer of size k
```
Outline

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

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Related work

5. Conclusions and future work
Streaming CnC

Streaming CnC – subset of CnC 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>Item collections</th>
<th>Steps Collections</th>
<th>CnC</th>
<th>Streaming</th>
<th>SCnC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producers</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N (streaming)</td>
<td>1</td>
</tr>
<tr>
<td>Consumers</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N (streaming)</td>
<td>1</td>
</tr>
<tr>
<td>Sources</td>
<td>N</td>
<td></td>
<td>1</td>
<td>N (streaming)</td>
<td>N (streaming)</td>
</tr>
<tr>
<td>Destinations</td>
<td>N</td>
<td></td>
<td>1</td>
<td>N (streaming)</td>
<td>N (streaming)</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

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

```java
public void Put (Object p) {
    a.send(p);
    ph.signal();
}
```
Results without dynamic parallelism

- CnC versus SCnC on StreamIt benchmarks
- Tests run on Xeon 16 core system
- CnC runs out of memory on input size 1/10 of the size used for SCnC
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.

- Dynamic parallelism is controlled through place tags.
  - Tags placed in control collections have 2 dimensions <place, value>:
    - place = 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
  - But there is support for dynamic stateful filters too

- Results are deterministic, outputs appearing in the order given by the controlTags
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

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

- Additional benefit from dynamic parallelization
  - Normalized to static parallel SCnC from previous slide
  - Variable delay added to simulate real statistics computations
Outline

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

2. Streaming Concurrent Collections (SCnC)

3. Dynamic parallelism for SCnC

4. Related work

5. Conclusions and future work
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());
    }
}
# 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 a subset of CnC that can be implemented through streaming.

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

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

• Proposed an algorithm to convert a CnC application to a SCnC application
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 collocated 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
Examples taken from StreamIt programs, and re-drawn for this presentation
class StepB {
    void step(Object tag, InputItemCollection ab, OutputItemCollection bd, ControlCollection d) {
        int tagVal = ((Integer)(tag)).intValue();
        int item = ab.get(tag);
        if (tagVal%2==0) {
            bd.put(tag, item*2);
        } else {
            bd.put(tag+1, item);
        }
        d.put(tag);
    }
}
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();
    }
}
```

Streaming phasers and accumulators:

- #wait operations ≤ #signal operations ≤ #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
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
Make environment explicit
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
SCnC Implementation: Streaming phasers

- 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
Phasers with dynamic parallelism

Problem:

• Multiple phaser producers and consumers if using more than one place:
  – Accumulator reduction operation is used to produce a single element instead of one for each producer
  – All consumers consume the same items

• We want:
  – Produce different items per place, put them all in the accumulator buffer
  – Consume different items

• Non-phaser approaches may not guarantee determinism
Phasers with dynamic parallelism

Solution:

• For each async of the same step collection:
  – If (tag.placeId == async.placeId), run user code
  – If not, a modified version of the code is run, that:
    • Does NOT do the intensive work to compute the put values
    • Does fake puts with null values so the accumulation result is the one coming from the user code.

• Next slide illustrates the runtime behavior
Control Tags (placeId, value)

<table>
<thead>
<tr>
<th>Place 0 Async</th>
<th>Place 1 Async</th>
<th>Produced Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,x)</td>
<td>WaitForTag() // (1,x)</td>
<td></td>
</tr>
<tr>
<td>Is tag.place==0? (F)</td>
<td>Is tag.place==1? (T)</td>
<td></td>
</tr>
<tr>
<td>Start fakeStep</td>
<td>Do Work(tag.value)</td>
<td></td>
</tr>
<tr>
<td>FakePut()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WaitForTag() // (0,x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get (tag.place=0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do Work(tag.value)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A)</td>
</tr>
</tbody>
</table>

Parallel Work

Put(A)
WaitForTag() // (0,x)
Get (tag.place!=0)
Start fakeStep
FakePut()
Generating the fake step code

Start with user step code:

• **Keep** the item get calls
• **Keep** the control flow leading to a put
• **Replace** puts with fake puts (signal, but no accumulate)
• **Remove** code that computes the values that were put by the user code (remove the work)
Generating the fake step code

```java
tag t = accT.get();
if (t.spaceTag == MySpacetag) {
    // *** USER CODE ***
    accI.doWait();
    item i1 = phI.get();
    result r = DoCondition(i1);
    if (cond(r)) {
        DoWork();
        accO.put();
        phO.signal();
    }
    // *** END USER CODE ***
}
else {  // *** GENERATED CODE ***
    // DO the reads
    acl.doWait();
    item i1 = phI.get();
    result r = DoCondition(i1);
    if (cond(r)) {
        // DO NOT do work
        // DO NOT put anything
        phO.signal();
    }
    // *** END GENERATED CODE ***
}
```

Dynamic Split Join

S1
place 0

S1
place 1

ItemCollection
0

S2

Control Collection

phT<SIG>, accT

<place0, _>

phT<WAIT>, accT

<place1, _>

phT<WAIT>, accT

phO<SIG>, accO

phO<WAIT>, accO
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: Multiple producers

Before:

Producer Step 1
Producer Step 2
Producer Step N

... 

Item Collection

After:

Producer Step 1
Producer Step 2
Producer Step N

... 

Item Collection 1
Item Collection 2
Item Collection N

Join Control Collection
Join Step

... 

Item Collection
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:

- Prescribing Step Collection
  - Control Collection 1
    - Prescribed Step Collection 1
  - Control Collection 2
    - Prescribed Step Collection 2
  - Control Collection 3
    - Prescribed Step Collection 3
Graph Shape Transformation: Multiple consumers

Before:

Producer Step -> Item Collection

Consumer Step 1

Consumer Step 2

... 

Consumer Step N

After:

Producer Step

Item Collection 1 -> Consumer Step 1

Item Collection 2 -> Consumer Step 2

... 

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
- Control Collection 2
- Item Collection 3
- Environment

After:

- Step 1
- Item Collection 1
- Step 3
- Step 2
- Item Collection 2
- Step 4
- Control Collection 2
- Item Collection 3
- Environment
- Entry Step
- Entry Control Collection
CnC to Streaming CnC

• Graph shape analysis
  – Needed because of the runtime requirement (single producer)

• Streaming access patterns checking

• Mapping on Streaming CnC
  – Transformation of item and control tags from CnC indexes to SCnC offsets
Transformation workflow

1. CnC application
2. Is the graph well formed?
   - No: Convert graph to well formed shape
   - Yes: Success in converting?
     - Yes: Ok
     - No: Check streaming access pattern
6. Fail: Map to HJ & streaming phasers
7. Working Streaming Application
8. Application is not streaming
Streaming access pattern checking

1. Require the graph edges to be labeled with put and get function, mapping the step control tag to the item tags of accessed items

2. Mathematically model the relationship between the environment and item collections through producer and consumer functions \( f_c \) and \( f_p \)

3. Symbolically check a small number of “streaming rules”

4. Use the same rules to size buffers for deadlock-freedom

\[
\begin{align*}
\text{producer} &= f_{\text{ip}1} \circ f_2 \circ f_{\text{cp}2} \circ f_1 \circ f_{\text{cp}1} \circ f_0 \\
\text{consumer} &= f_{\text{ic}1} \circ f_3 \circ f_{\text{cp}3} \circ f_0
\end{align*}
\]
Streaming rules

- **Producer precedence**: for any \( y \) and any producer and consumer function of the same item collection
  \[ f_p^{-1}(y) \leq f_c^{-1}(y) \]

- **Sliding window**:
  \[ f_{c\,\text{min}}(x) \leq f_{c\,\text{min}}(x+1) \]

Define minimum consumer function as:
\[ f_{c\,\text{min}}(x) = \min(f_{c\,1}(x), f_{c\,2}(x), \ldots), \text{ for any } x \]
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)