Bounded Stream Scheduling in Polyhedral OpenStream

Nuno Miguel Nobre | nunomiguel.nobre@manchester.ac.uk
Andi Drebes | andi.drebes@inria.fr
Graham Riley | graham.riley@manchester.ac.uk
Antoniu Pop | antoniu.pop@manchester.ac.uk

IMPACT 2020: January 22, 2020 | Bologna, Italy
The case for streaming dataflow languages
The case for streaming dataflow languages

Instead of barrier synchronization
The case for streaming dataflow languages

Instead of barrier synchronization

Point-to-point synchronization:
  Hide latency
  More opportunities for parallelism
The case for streaming dataflow languages

Instead of barrier synchronization

Point-to-point synchronization:
  Hide latency
  More opportunities for parallelism
  Task
The case for streaming dataflow languages

Instead of barrier synchronization

Point-to-point synchronization:
  Hide latency
  More opportunities for parallelism
    Task
    Data
The case for streaming dataflow languages

Instead of barrier synchronization
Point-to-point synchronization:
- Hide latency
- More opportunities for parallelism
  - Task
  - Data
  - Pipeline
The case for streaming dataflow languages

Instead of barrier synchronization

Point-to-point synchronization:
  Hide latency
  More opportunities for parallelism
    Task
    Data
    Pipeline

Scheduling is the runtime’s job

Provide functional determinism

No in-place writes:
  Fewer dependencies
The case for streaming dataflow languages

Instead of barrier synchronization

Point-to-point synchronization:
  - Hide latency
  - More opportunities for parallelism
    - Task
    - Data
    - Pipeline

Scheduling is the runtime’s job

Provide functional determinism

No in-place writes:
  - Fewer dependencies
  - Memory footprint
Outline

1) OpenStream
   • Overview & polyhedral subset
   • Computing dependencies and schedules

2) Stream bounding
   • Basic strategy & limitations
   • Usage guidelines
OpenStream: a short overview

Data-flow extension to OpenMP

- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

Tasks access stream elements through **windows**: created dynamically at runtime
OpenStream: a short overview

Data-flow extension to OpenMP

- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

Tasks access stream elements through **windows**:

```
stream s;
```

Control program          Accesses on stream s          Task dependencies: overlapping windows
OpenStream: a short overview

Data-flow extension to OpenMP

- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

Tasks access stream elements through **windows**:

```c
stream s;

task p1 {
    write three times to s;
}
```

- **Control program**
- **Accesses on stream s**
- **Task dependencies**: overlapping windows

Tasks spawn as concurrent coroutines, and streams are unbounded channels for communication between tasks. Streams are created dynamically at runtime.
OpenStream: a short overview

Data-flow extension to OpenMP

- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

Tasks access stream elements through **windows**:

```plaintext
stream s;

task p1 {
    write three times to s;
}

task p2 {
    write two times to s;
}
```

Control program  Accesses on stream s  Task dependencies: overlapping windows
OpenStream: a short overview

Data-flow extension to OpenMP

- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

Tasks access stream elements through **windows**:

```c
stream s;

task p1 {
    write three times to s;
}

task p2 {
    write two times to s;
}

task r {
    peek three times from s;
}
```

Control program                     Accesses on stream s                     Task dependencies: overlapping windows

![Diagram showing task dependencies and window access on stream s]
OpenStream: a short overview

Data-flow extension to OpenMP

- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

Tasks access stream elements through **windows**:

```plaintext
stream s;

task p1 {
    write three times to s;
}
task p2 {
    write two times to s;
}
task r {
    peek three times from s;
}
task c {
    read five times from s;
}
```

Control program

Accesses on stream s

Task dependencies: overlapping windows
Polyhedral OpenStream: computing dependencies

```plaintext
stream s;
parameter N;

for(i = 0; i < N; ++i)
  task tw {
    write two times to s;
  }

for(j = 0; j < N/2; ++j)
  task tc {
    read four times from s;
  }
```

Polyhedral control program:

- No nested task creation
- Affine control statements
Polyhedral OpenStream: computing dependencies

stream s;
parameter N;

for(i = 0; i < N; ++i)
  task tw {
    write two times to s;  \( W_s(t_w, i) = 2i \)  window: [2i, 2i + 1]
  }

for(j = 0; j < N/2; ++j)
  task tc {
    read four times from s;  \( R_s(t_c, j) = 4j \)  window: [4j, 4j + 3]
  }

Polyhedral control program:

- No nested task creation
- Affine control statements

Can statically count \( W_s \) and \( R_s \) and obtain access windows:

- Ehrhart polynomials
- Brion generating functions
Polyhedral OpenStream: computing dependencies

stream s;
parameter N;

for (i = 0; i < N; ++i)
  task tw {
    write two times to s;
    \[ W_s(t_w,i) = 2i \text{ window: } [2i, 2i + 1] \]
  }

for (j = 0; j < N/2; ++j)
  task tc {
    read four times from s;
    \[ R_s(t_c,j) = 4j \text{ window: } [4j, 4j + 3] \]
  }

Polyhedral control program:
- No nested task creation
- Affine control statements

Can statically count \( W_s \) and \( R_s \) and obtain access windows:
- Ehrhart polynomials
- Brion generating functions

Compute dependencies by intersecting windows

\[ 2i \leq 4j + 3 \land 4j \leq 2i + 1 \]
\[ 2j \leq i \leq 2j + 1 \]
Polyhedral OpenStream: scheduling

Dependencies: polynomial (in)equalities $p_i(x)$, semi-algebraic sets:

$$S = \{x \in \mathbb{R}^d \mid p_1(x) \geq 0, p_2(x) \geq 0, \ldots, p_n(x) \geq 0\}$$
Polyhedral OpenStream: scheduling

Dependencies: polynomial (in)equalities $p_i(x)$, semi-algebraic sets:

$$S = \{x \in \mathbb{R}^d \mid p_1(x) \geq 0, p_2(x) \geq 0, \ldots, p_n(x) \geq 0\}$$

A polynomial $P(x)$ is strictly positive in $S$ iff:

$$P(x) = \sum_{k \in \mathbb{N}^n} \lambda_k p_1^{k_1}(x)p_2^{k_2}(x) \ldots p_n^{k_n}(x) \quad \lambda_k \geq 0 \quad \sum \lambda_k > 0$$
Polyhedral OpenStream: scheduling

Dependencies: polynomial (in)equalities $p_i(x)$, semi-algebraic sets:

$$S = \{ x \in \mathbb{R}^d \mid p_1(x) \geq 0, p_2(x) \geq 0, \ldots, p_n(x) \geq 0 \}$$

A polynomial $P(x)$ is strictly positive in $S$ iff:

$$P(x) = \sum_{k \in \mathbb{N}^n} \lambda_k p_1^{k_1}(x)p_2^{k_2}(x)\cdots p_n^{k_n}(x) \quad \lambda_k \geq 0 \quad \sum \lambda_k > 0$$

Cannot possibly exhaust all $k$ in finite time:

• Semi-decidable (undecidable) problem
• In practice, ~ conservative ‘Farkas lemma’
stream s;
parameter N;

for(i = 0; i < N; ++i)
    task tw {
        write two times to s;
    }

for(j = 0; j < N/2; ++j)
    task tc {
        read four times from s;
    }
stream s;
parameter N;

for(i = 0; i < N; ++i)
  task tw {
    write two times to s;
  }

for(j = 0; j < N/2; ++j)
  task tc {
    read four times from s;
  }

Stream bound: 4 elements
stream s;
parameter N;

for(i = 0; i < N; ++i)
  task tw {
      write two times to s;
  }

for(j = 0; j < N/2; ++j)
  task tc {
      read four times from s;
  }

Stream bound: 4 elements

⚠️ ≤ \( W_s(t_{w,2}) + \text{(\# writes)} - \text{bound} - 1 \)

= 4 + 2 - 4 - 1 = 1

New back-pressure dependency:
  some parallelism is lost
stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsource {
    write once to s;
}
Stream bounding: the implications of (partial) causality

```
stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsource {
    write once to s;
}
```

Stream bound: 2 elements, **deadlock**
Stream bounding: the implications of (partial) causality

stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsource {
    write once to s;
}

Stream bound: 2 elements, **deadlock**

Caveat: the stream is 2-element bound
Stream bounding: the implications of (partial) causality

```c
stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsource {
    write once to s;
}

Indexing →

Stream bound: 2 elements, deadlock
Caveat: the stream is 2-element bound
```

Filling ←
Stream bounding: the implications of (partial) causality

stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsourc { 
    write once to s;
}

Stream bound: 2 elements, deadlock
Caveat: the stream is 2-element bound
Stream bounding: the implications of (partial) causality

```c++
stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsource {
    write once to s;
}
```

Stream bound: 2 elements, **deadlock**

Caveat: the stream is a 2-element bound
stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsource {
    write once to s;
}

Stream bound: 2 elements, **deadlock**

Caveat: the stream is 2-element bound
Stream binding: the implications of (partial) causality

```
stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
    task tw {
        read once from s;
        write once to s;
    }

task tsource {
    write once to s;
}

Stream bound: 2 elements, deadlock
Caveat: the stream is 2-element bound
```
Stream bounding: the implications of (partial) causality

stream s;
parameter N;

task tsink {
    read once from s;
}

for(k = 1; k < N; ++k)
task tw {
    read once from s;
    write once to s;
}

task tsource {
    write once to s;
}

Stream bound: 2 elements, *deadlock*

Caveat: the stream is 2-element bound
stream s1, s2;

task tw1 {
    write two times to s1;
}
task tw2 {
    write three times to s2;
}
task ta {
    write two times to s2;
    read two times from s1;
}
task tb {
    write once to s1;
    read three times from s2;
}
task tc1 {
    read once from s1;
}
task tc2 {
    read two times from s2;
}
stream s1, s2;

task tw1 {
    write two times to s1;
}

task tw2 {
    write three times to s2;
}

task ta {
    write two times to s2;
    read two times from s1;
}

task tb {
    write once to s1;
    read three times from s2;
}

task tc1 {
    read once from s1;
}

task tc2 {
    read two times from s2;
}
Stream bounding: global surface minimization

Minimum bounds:

\[ s_1: 2 \text{ elements} \]
\[ s_2: 3 \text{ elements} \]

```
stream s1, s2;

task tw1 {
    write two times to s1;
}
task tw2 {
    write three times to s2;
}
task ta {
    write two times to s2;
    read two times from s1;
}
task tb {
    write once to s1;
    read three times from s2;
}
task tc1 {
    read once from s1;
}
task tc2 {
    read two times from s2;
}
```
Stream bounding: global surface minimization

Minimum bounds:

- **s₁**: 2 elements
- **s₂**: 3 elements

```c
stream s1, s2;

task tw1 {
    write two times to s1;
}

task tw2 {
    write three times to s2;
}

task ta {
    write two times to s2;
    read two times from s1;
}

task tb {
    write once to s1;
    read three times from s2;
}

task tc1 {
    read once from s1;
}

task tc2 {
    read two times from s2;
}
```
Stream bounding: global surface minimization

Minimum bounds:

\[ s_1: 2 \text{ elements} \]

\[ s_2: 3 \text{ elements} \]

\begin{align*}
\text{stream } & s_1, s_2; \\
\text{task } & tw1 \{ \\
& \quad \text{write two times to } s_1; \\
& \} \\
\text{task } & tw2 \{ \\
& \quad \text{write three times to } s_2; \\
& \} \\
\text{task } & ta \{ \\
& \quad \text{write two times to } s_2; \\
& \quad \text{read two times from } s_1; \\
& \} \\
\text{task } & tb \{ \\
& \quad \text{write once to } s_1; \\
& \quad \text{read three times from } s_2; \\
& \} \\
\text{task } & tc1 \{ \\
& \quad \text{read once from } s_1; \\
& \} \\
\text{task } & tc2 \{ \\
& \quad \text{read two times from } s_2; \\
& \}
\end{align*}

We can have one, but \textit{not both}:

\[ s_1: 2 \text{ elements \& } s_2: \geq 5 \text{ elements} \]

\[ s_1: \geq 3 \text{ elements \& } s_2: 3 \text{ elements} \]
Stream bounding: global surface minimization

Minimum bounds:

- \( s_1 \): 2 elements
- \( s_2 \): 3 elements

```java
stream s1, s2;

task tw1 {
    write two times to s1;
}
task tw2 {
    write three times to s2;
}
task ta {
    write two times to s2;
    read two times from s1;
}
task tb {
    write once to s1;
    read three times from s2;
}
task tc1 {
    read once from s1;
}
task tc2 {
    read two times from s2;
}
```

\( s_1 \): 0 elements
\( s_2 \): 0 elements

The return of the causality caveat, assume these bounds:

- \( s_1 \): 2 elements & \( s_2 \): 3 elements
Stream bounding: global surface minimization

Minimum bounds:

- \( s_1 \): 2 elements
- \( s_2 \): 3 elements

```plaintext
stream s1, s2;
task tw1 {
    write two times to s1;
}
task tw2 {
    write three times to s2;
}
task ta {
    write two times to s2;
    read two times from s1;
}
task tb {
    write once to s1;
    read three times from s2;
}
task tc1 {
    read once from s1;
}
task tc2 {
    read two times from s2;
}
```

The return of the causality caveat, assume these bounds:

- \( s_1 \): 2 elements & \( s_2 \): 3 elements
Stream bounding: global surface minimization

Minimum bounds:

stream s1, s2;

task tw1 {
    write two times to s1;
}

task tw2 {
    write three times to s2;
}

task ta {
    write two times to s2;
    read two times from s1;
}

task tb {
    write once to s1;
    read three times from s2;
}

task tc1 {
    read once from s1;
}

task tc2 {
    read two times from s2;
}

s_1: 0 elements
s_2: 2 elements

The return of the causality caveat, assume these bounds:

s_1: 2 elements & s_2: 3 elements
Stream bounding: global surface minimization

Minimum bounds:

\begin{equation}
\begin{align*}
\text{stream } s_1, s_2; \\
\text{task } tw1 \{ \\
& \text{write two times to } s_1; \\
\} \\
\text{task } tw2 \{ \\
& \text{write three times to } s_2; \\
\} \\
\text{task } ta \{ \\
& \text{write two times to } s_2; \\
& \text{read two times from } s_1; \\
\} \\
\text{task } tb \{ \\
& \text{write once to } s_1; \\
& \text{read three times from } s_2; \\
\} \\
\text{task } tc1 \{ \\
& \text{read once from } s_1; \\
\} \\
\text{task } tc2 \{ \\
& \text{read two times from } s_2; \\
\}
\end{align*}
\end{equation}

\begin{itemize}
\item $s_1$: 2 elements
\item $s_2$: 3 elements
\item $s_1$: 0 elements
\item $s_2$: 0 elements
\end{itemize}

The return of the causality caveat, assume these bounds:

\begin{itemize}
\item $s_1$: 2 elements & $s_2$: 3 elements
\end{itemize}
Stream bounding: global surface minimization

Minimum bounds:

\[ s_1 : 2 \text{ elements} \]
\[ s_2 : 3 \text{ elements} \]

```plaintext
stream s1, s2;

// Write twice to s1
task tw1 {
    write two times to s1;
}

// Write three times to s2
task tw2 {
    write three times to s2;
}

// Write twice to s2 and read twice from s1
task ta {
    write two times to s2;
    read two times from s1;
}

// Write once to s1 and read three times from s2
task tb {
    write once to s1;
    read three times from s2;
}

// Read once from s1
task tc1 {
    read once from s1;
}

// Read two times from s2
task tc2 {
    read two times from s2;
}
```

The return of the causality caveat, assume these bounds:

\[ s_1 : 0 \text{ elements} \]
\[ s_2 : 3 \text{ elements} \]
Stream bounding: global surface minimization

Minimum bounds:

```
stream s1, s2;
task tw1 {
    write two times to s1;
}
task tw2 {
    write three times to s2;
}
task ta {
    write two times to s2;
    read two times from s1;
}
task tb {
    write once to s1;
    read three times from s2;
}
task tc1 {
    read once from s1;
}
task tc2 {
    read two times from s2;
}
```

s₁: 1 element  
s₂: 0 elements

The return of the causality caveat, assume these bounds:

s₁: 2 elements & s₂: 3 elements
Stream bounding: global surface minimization

Minimum bounds:

```
stream s1, s2;

task tw1 {
    write two times to s1;
}
task tw2 {
    write three times to s2;
}
task ta {
    write two times to s2;
    read two times from s1;
}
task tb {
    write once to s1;
    read three times from s2;
}
task tc1 {
    read once from s1;
}
task tc2 {
    read two times from s2;
}
```

The return of the causality caveat, assume these bounds:

- $s_1$: 2 elements & $s_2$: 3 elements
Can we run a given program on a device with memory $M$?

1) Select stream bounds combination s.t. $\sum_s \text{bound}_s = M$

2) Add back-pressure dependencies for this combination

3) Look for schedule

4) If found: guaranteed execution
   If not found: if other combinations available, 1)
     if all exhausted, conservatively assume execution not possible
Summary

Back-pressure dependencies:
1) Bound streams
2) Statically, but conservatively, decide execution in limited memory
3) Limitations:
   • Causality-induced ‘spurious’ deadlocks
   • Non-independent stream minimization
   • Overestimation of actual memory usage
   • Deadlock detection undecidability