The University of Manchester

Advanced Processor Technologies

Bounded Stream Scheduling in Polyhedral OpenStream

Nuno Miguel Nobre | nunomiguel.nobre@manchester.ac.uk

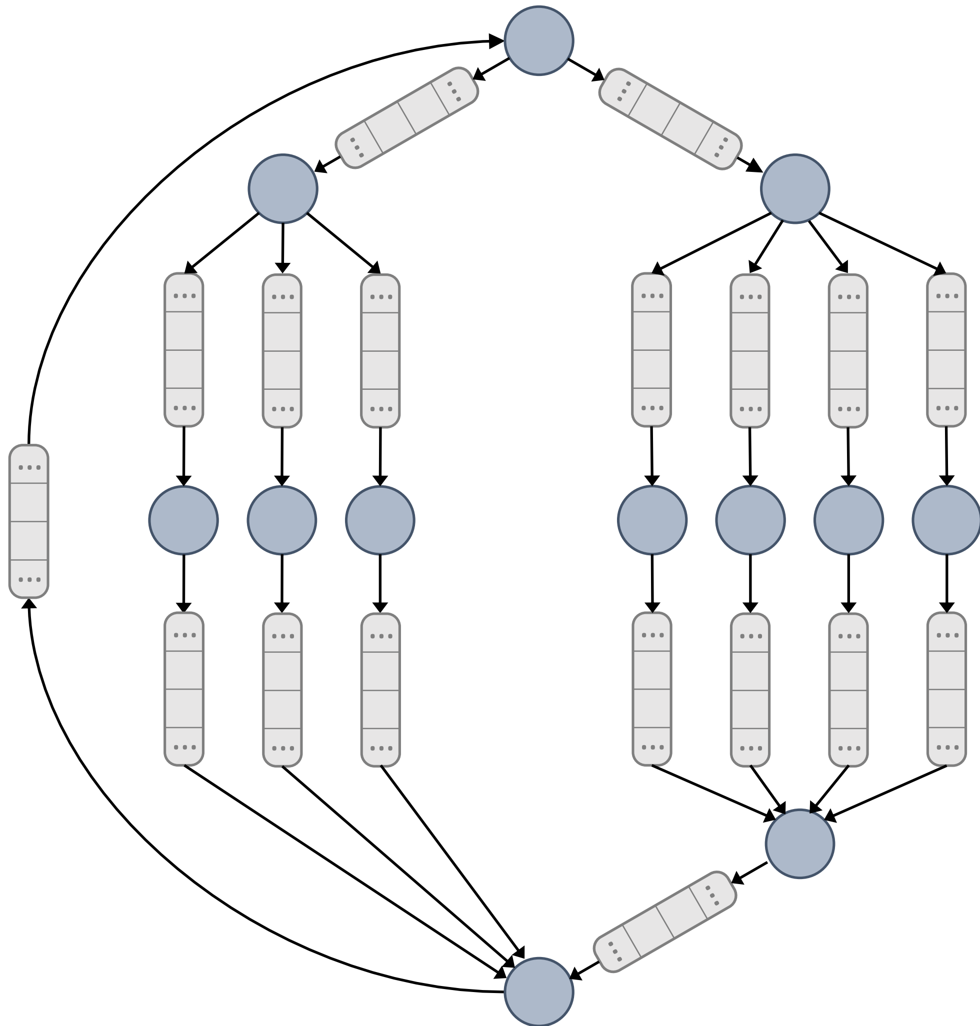
Andi Drebes | andi.drebes@inria.fr

Graham Riley | graham.riley@manchester.ac.uk

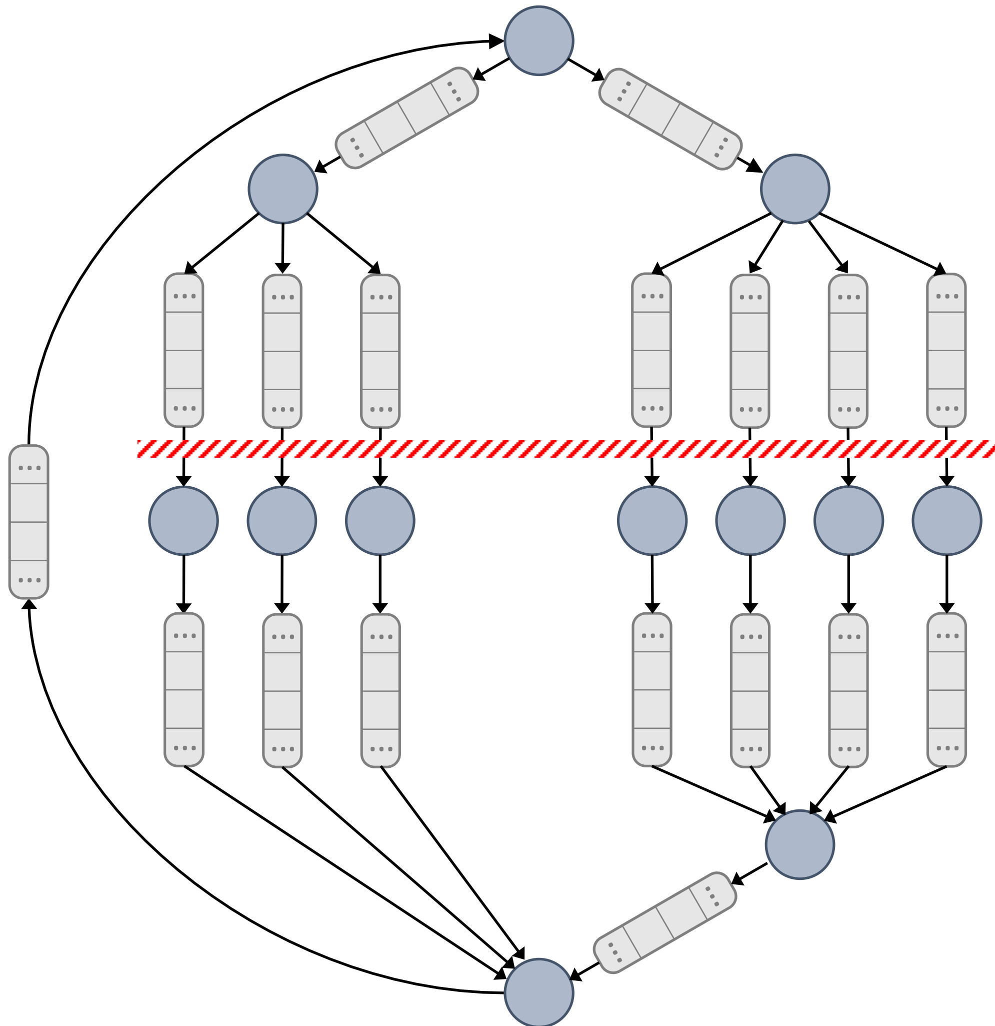
Antoniu Pop | antoniu.pop@manchester.ac.uk

IMPACT 2020: January 22, 2020 | Bologna, Italy

The case for streaming dataflow languages

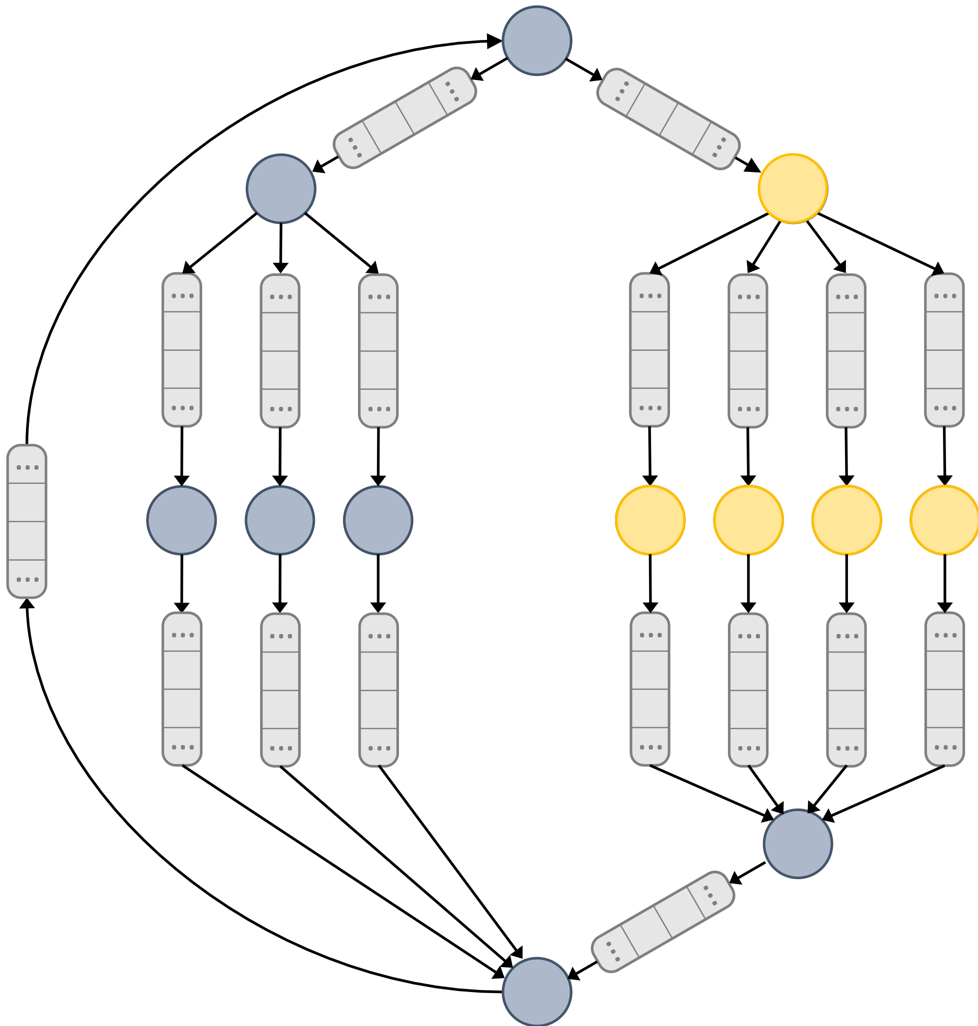


The case for streaming dataflow languages



Instead of barrier synchronization

The case for streaming dataflow languages

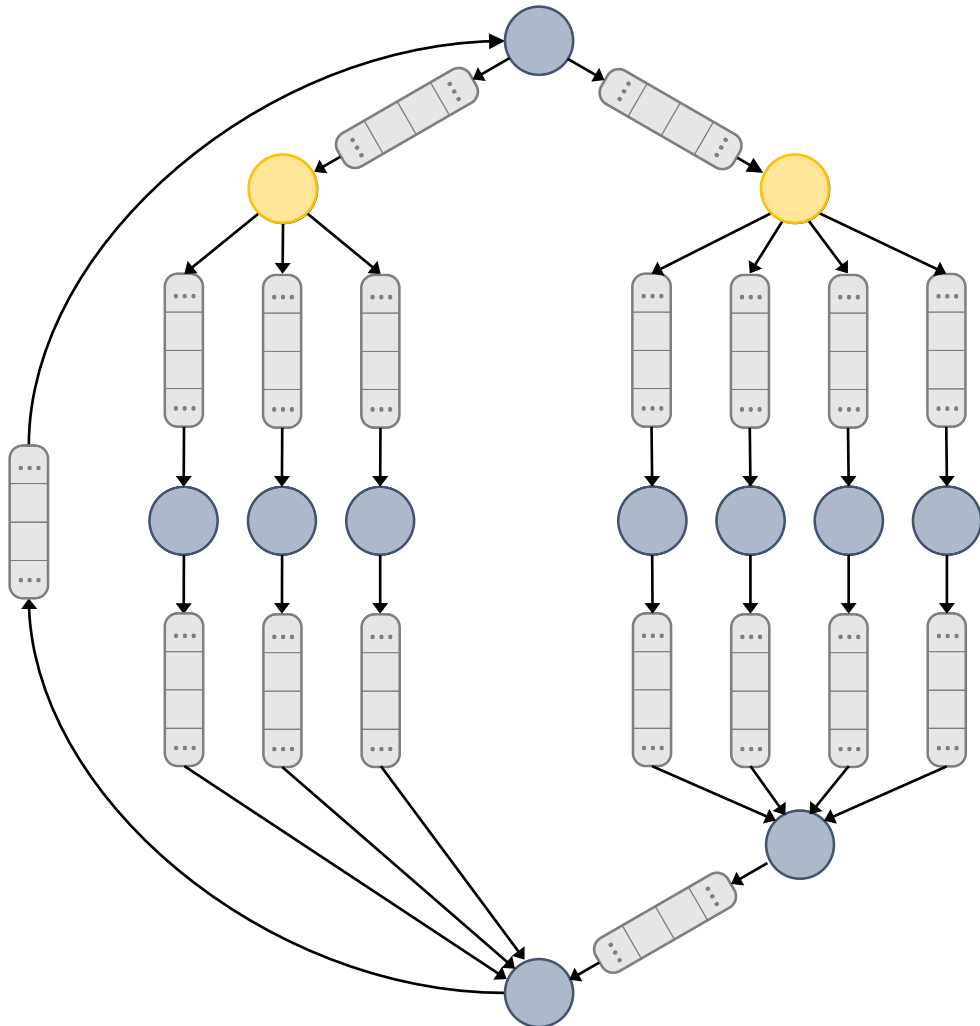


Instead of barrier synchronization

Point-to-point synchronization:

- Hide latency
- More opportunities for parallelism

The case for streaming dataflow languages

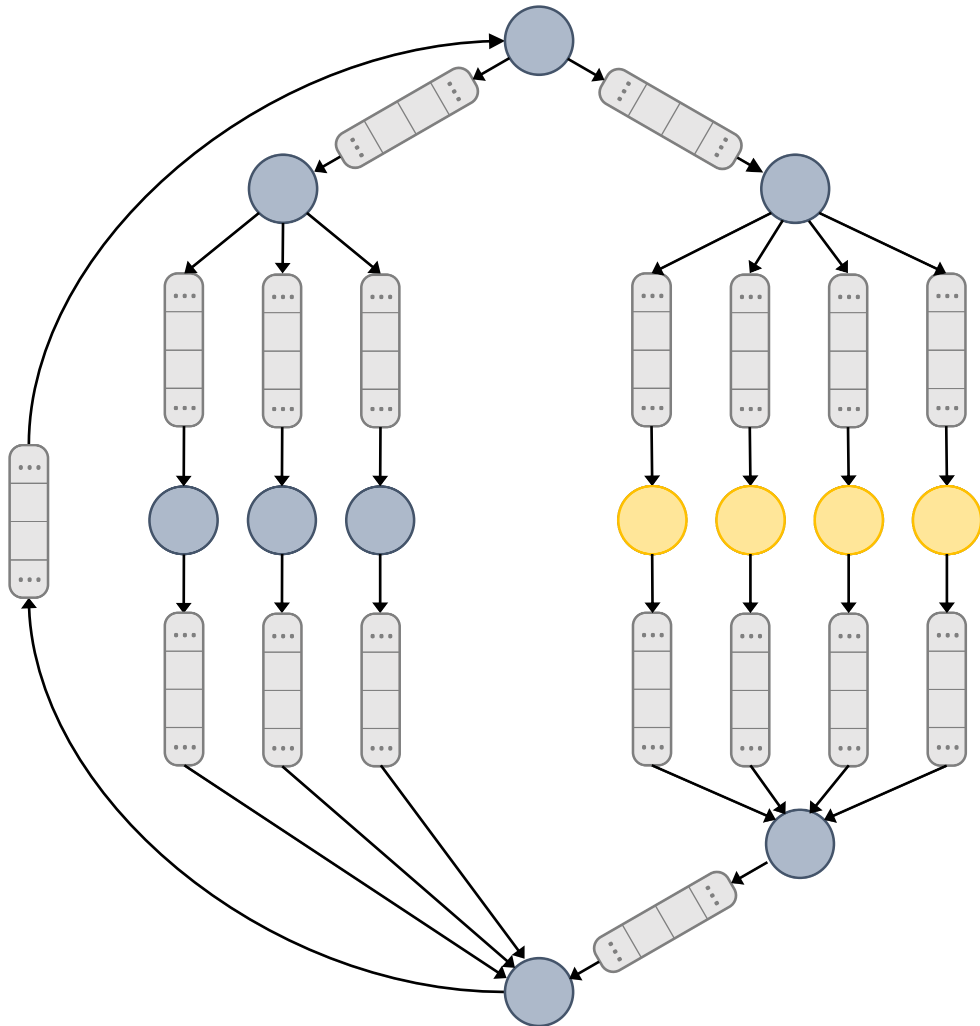


Instead of barrier synchronization

Point-to-point synchronization:

- Hide latency
- More opportunities for parallelism
  - Task

The case for streaming dataflow languages

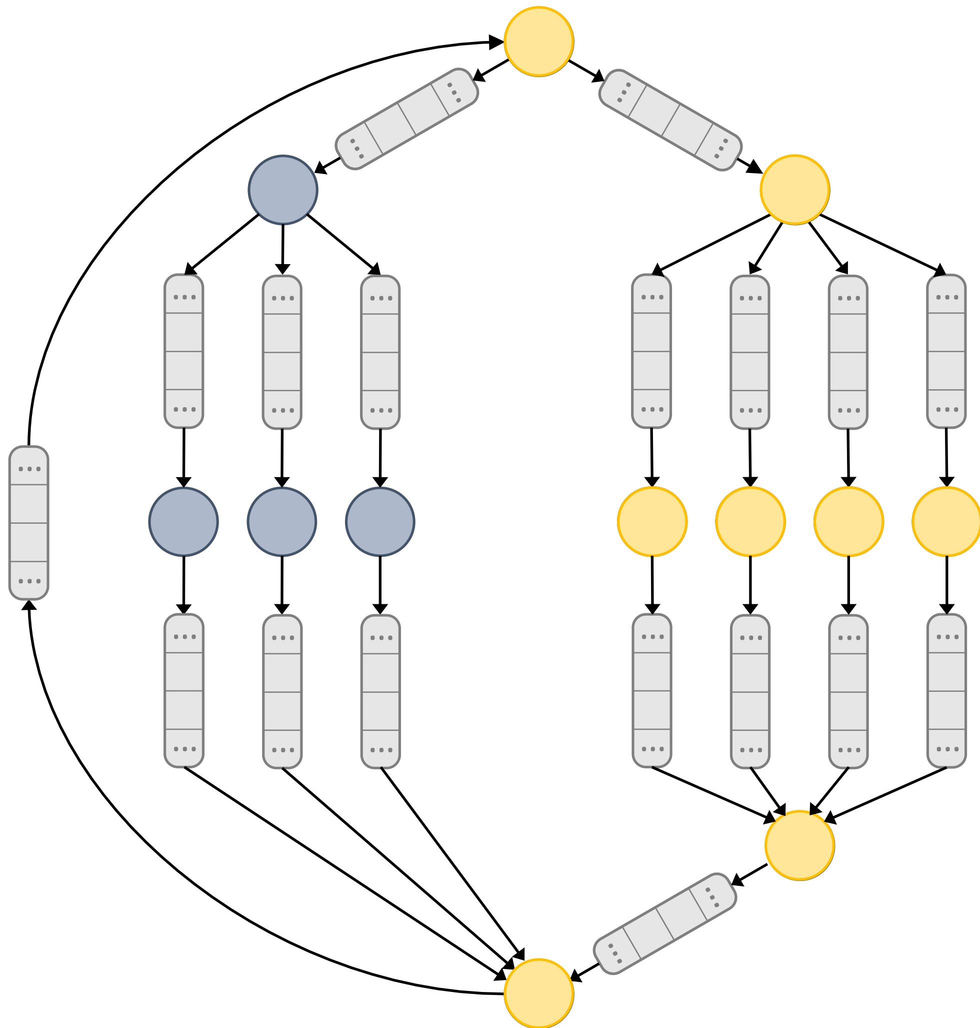


Instead of barrier synchronization

Point-to-point synchronization:

- Hide latency
- More opportunities for parallelism
  - Task
  - Data

The case for streaming dataflow languages

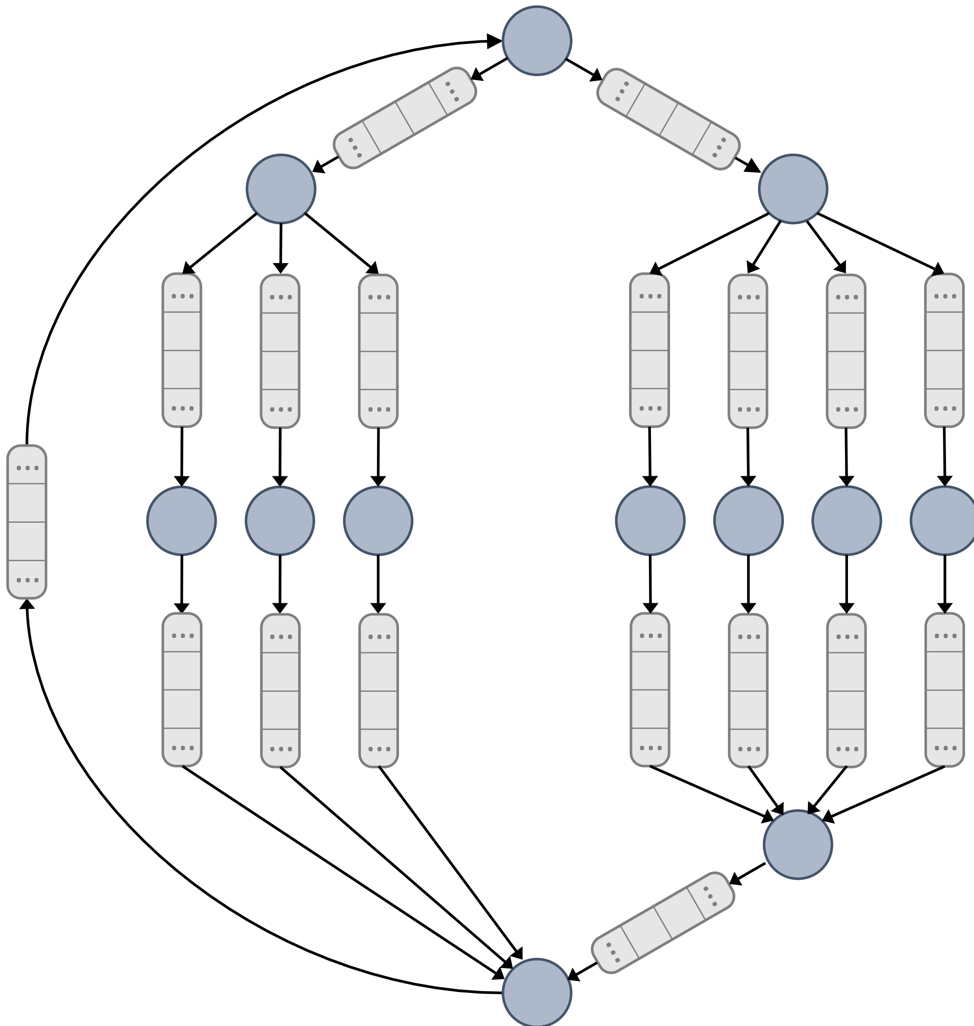


Instead of barrier synchronization

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

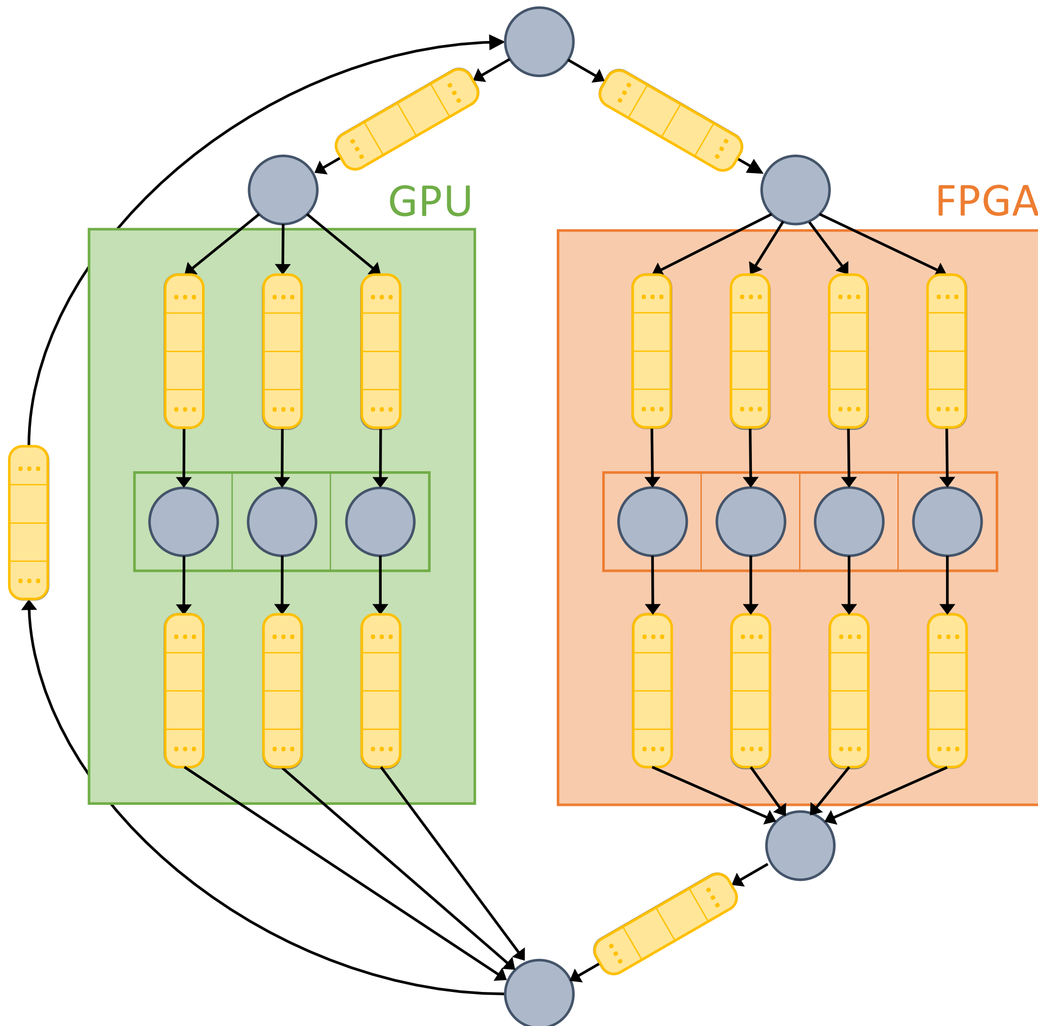
Scheduling is the runtime's job

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

} created dynamically at runtime

Tasks access stream elements through **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

} created dynamically at runtime

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

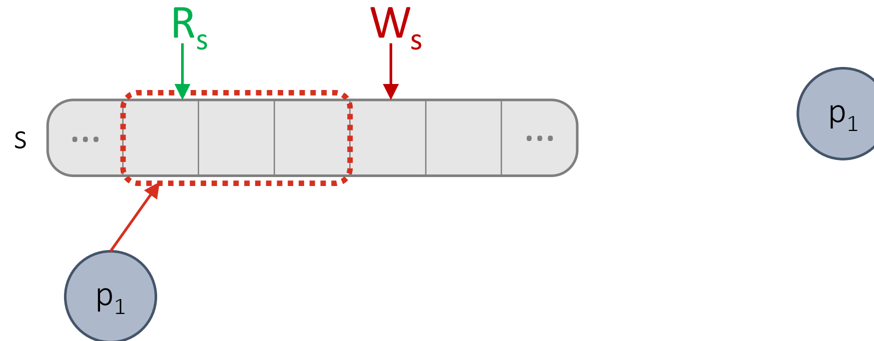
- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

} created dynamically at runtime

Tasks access stream elements through **windows**:

```
stream s;

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



Control program

Accesses on stream s

Task dependencies: overlapping windows

OpenStream: a short overview

Data-flow extension to OpenMP

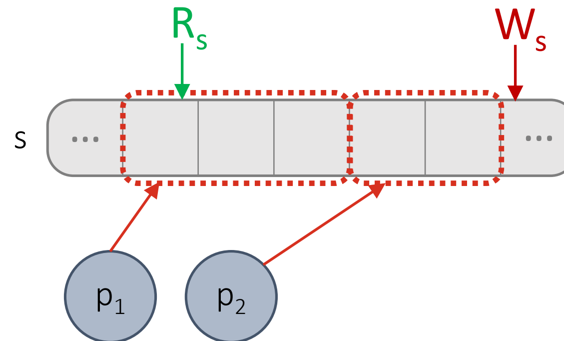
- **Tasks**: units of work spawned as concurrent coroutines
- **Streams**: *unbounded* channels for communication between tasks

} created dynamically at runtime

Tasks access stream elements through **windows**:

```
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

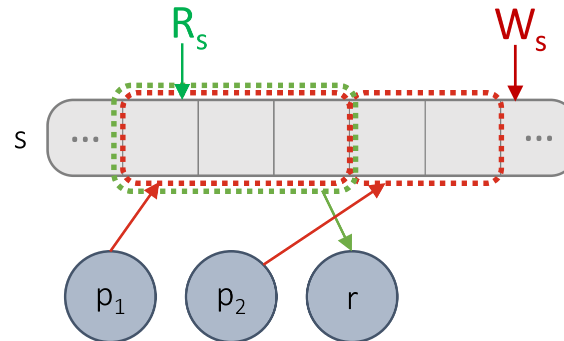
} created dynamically at runtime

Tasks access stream elements through **windows**:

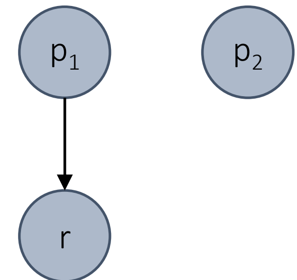
```
stream s;

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

OpenStream: a short overview

Data-flow extension to OpenMP

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

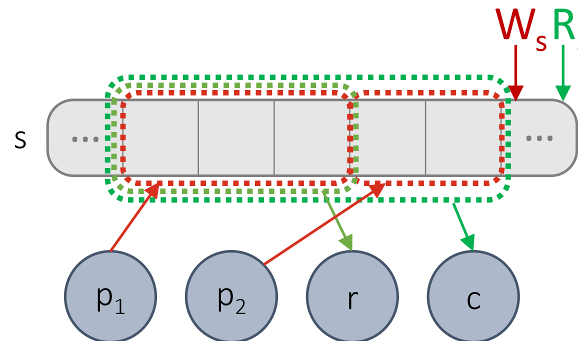
} created dynamically at runtime

Tasks access stream elements through **windows**:

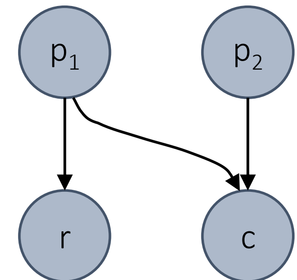
```
stream s;

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

```
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;
      }

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

$W_s(t_w, i) = 2i$ window: $[2i, 2i + 1]$

$R_s(t_c, j) = 4j$ 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;
      }

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

$W_s(t_w, i) = 2i$ window: $[2i, 2i + 1]$

$R_s(t_c, j) = 4j$ window: $[4j, 4j + 3]$

$2i \leq 4j + 3 \land 4j \leq 2i + 1$

$2j \leq i \leq 2j + 1$

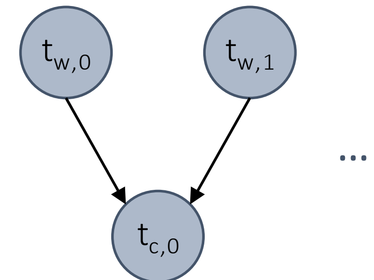
Polyhedral control program:

- 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



Polyhedral OpenStream: scheduling

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

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

Polyhedral OpenStream: scheduling

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

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

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

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

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:

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

Cannot possibly exhaust all k in finite time:

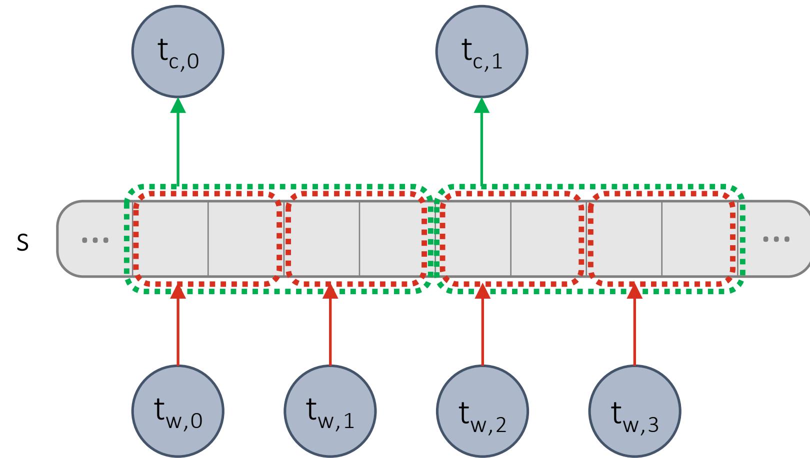
- Semi-decidable (undecidable) problem
- In practice, $\sim$ conservative ‘Farkas lemma’

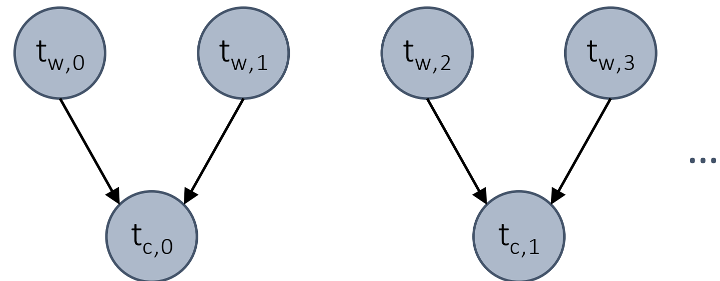
Stream bounding: back-pressure WaRs

```
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;
    }
```





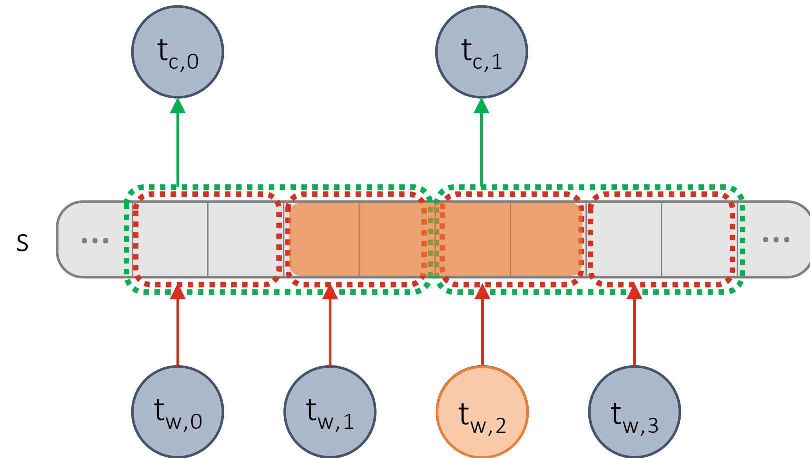
Stream bounding: back-pressure WaRs

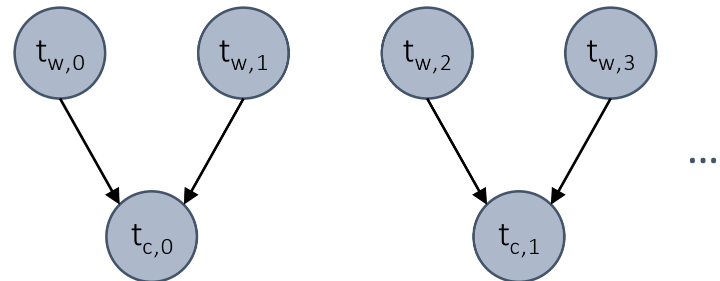
```
stream s;
parameter N;

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



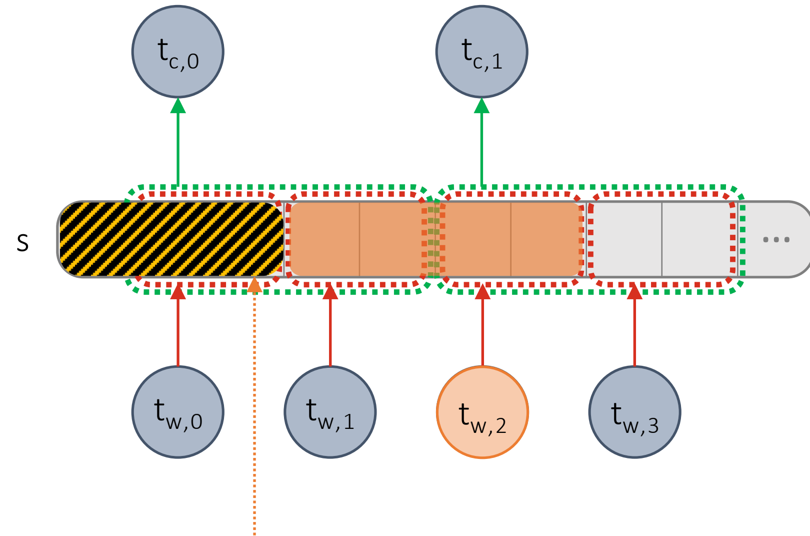


Stream bounding: back-pressure WaRs

```
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;
    }
```



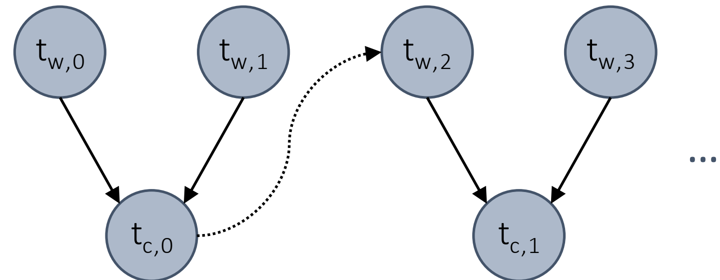
Stream bound: 4 elements

⚠ $\leq W_s(t_{w,2})$ + (# writes) – bound – 1

= 4 + 2 – 4 – 1 = 1

New back-pressure dependency:

some parallelism is lost



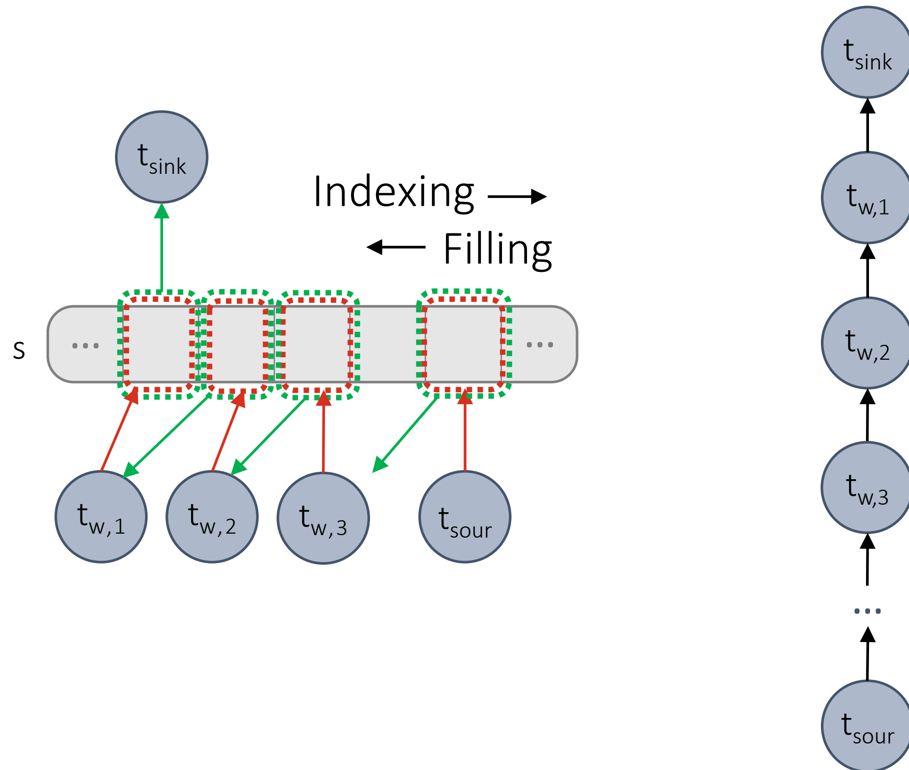
Stream bounding: the implications of (partial) causality

```
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;
}
```



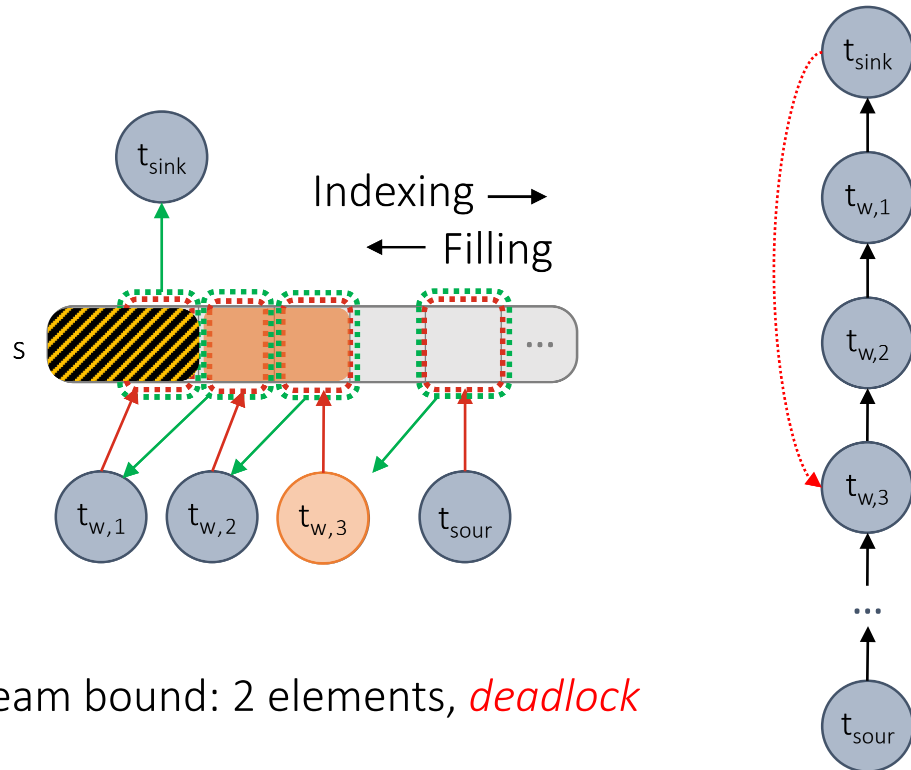
Stream bounding: the implications of (partial) causality

```
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 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;
}
```

t_{sink}

Indexing →

← Filling

s

…

$t_{w,1}$ $t_{w,2}$ $t_{w,3}$ t_{sour}

Stream bound: 2 elements, *deadlock*

Caveat: the stream ***is*** 2-element bound

t_{sink}

$t_{w,1}$

$t_{w,2}$

$t_{w,3}$

…

t_{sour}

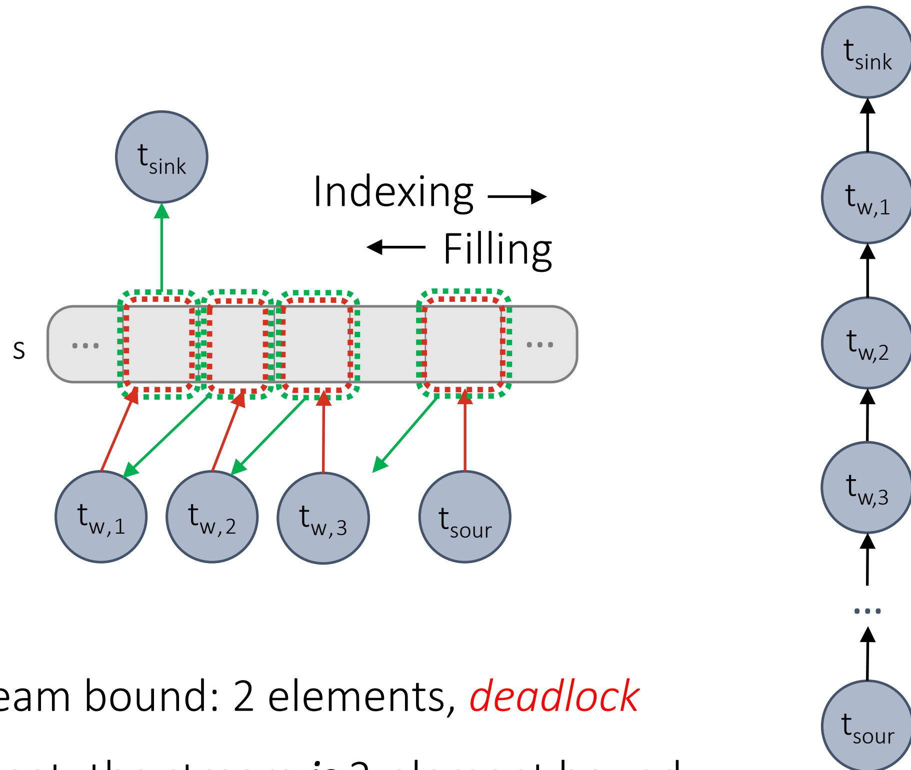
Stream bounding: the implications of (partial) causality

```
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

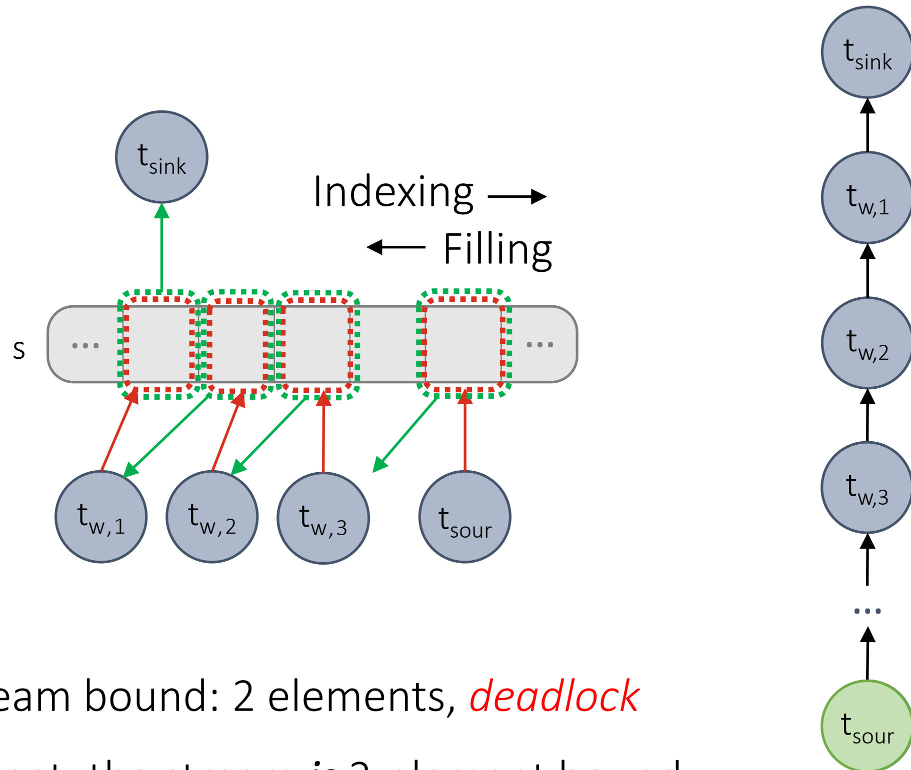
Stream bounding: the implications of (partial) causality

```
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

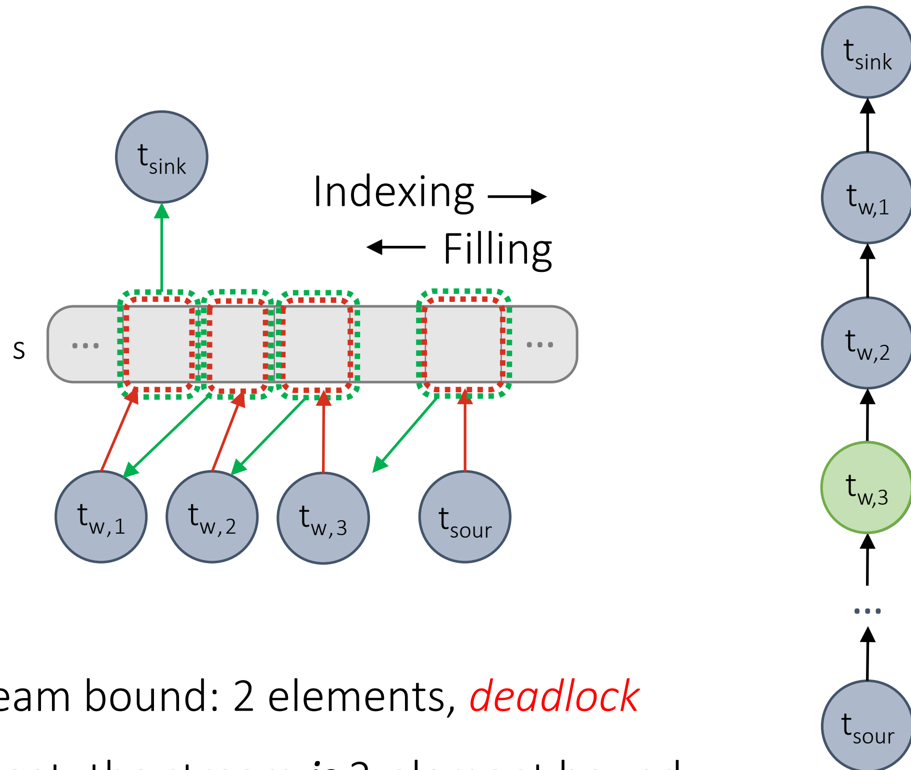
Stream bounding: the implications of (partial) causality

```
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

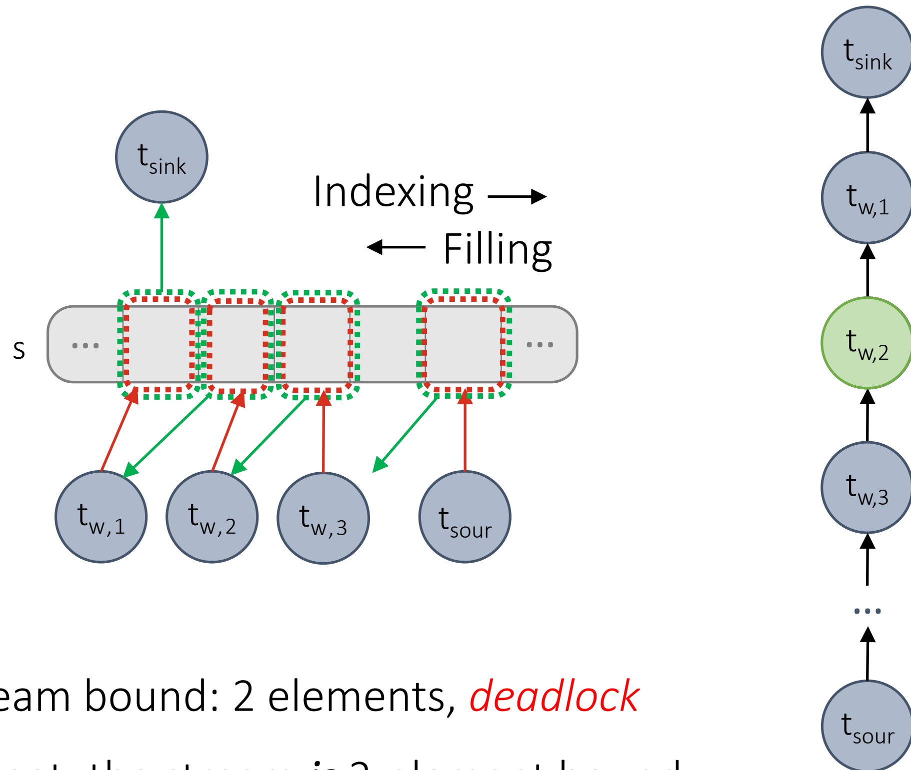
Stream bounding: the implications of (partial) causality

```
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

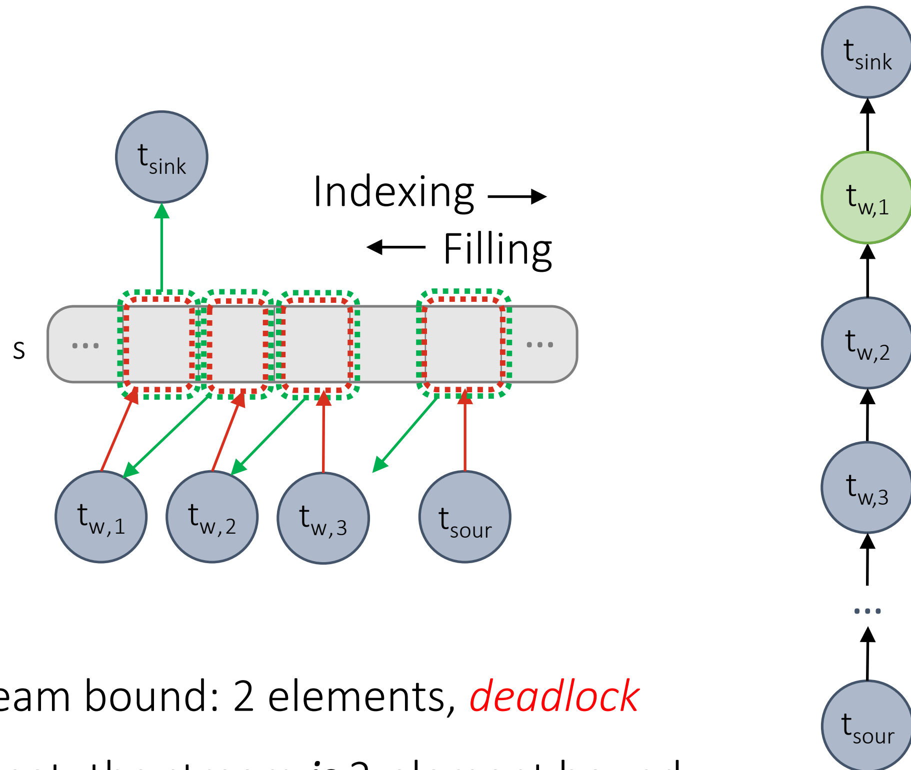
Stream bounding: the implications of (partial) causality

```
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

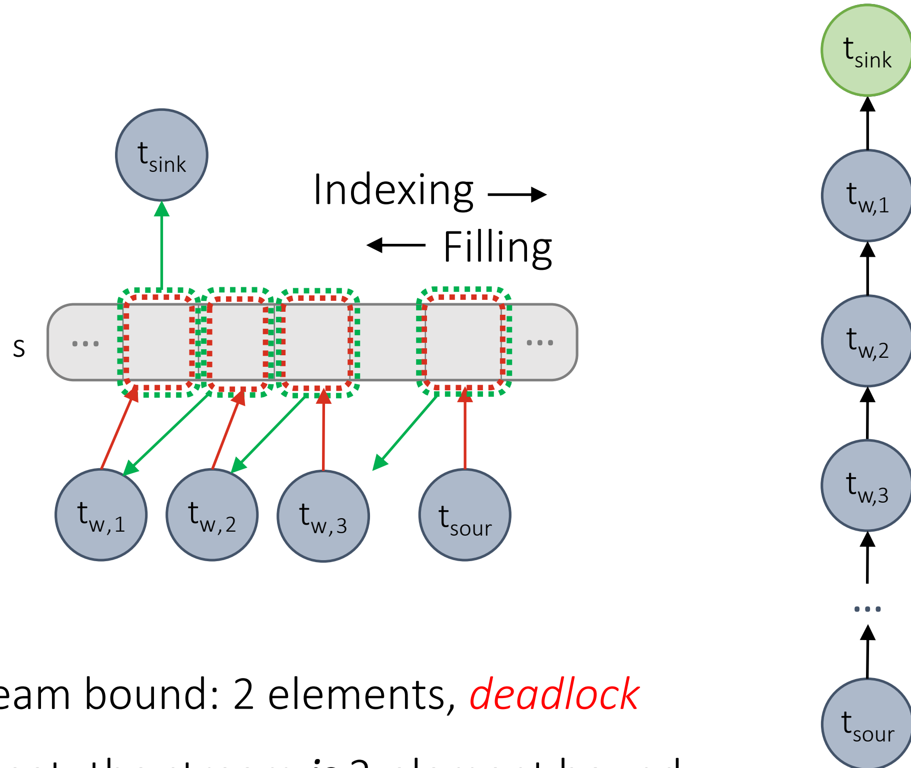
Stream bounding: the implications of (partial) causality

```
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;
}
```



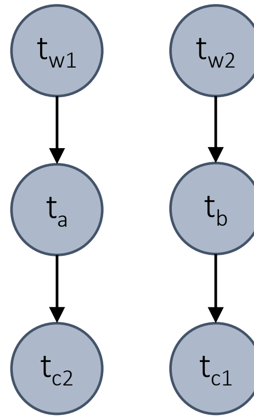
Stream bound: 2 elements, *deadlock*

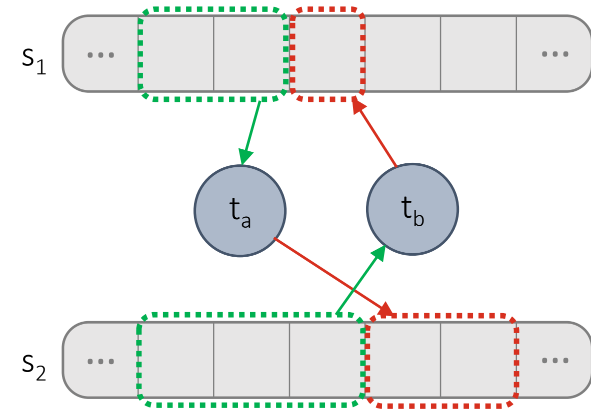
Caveat: the stream ***is*** 2-element bound

Stream bounding: global surface minimization

```
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

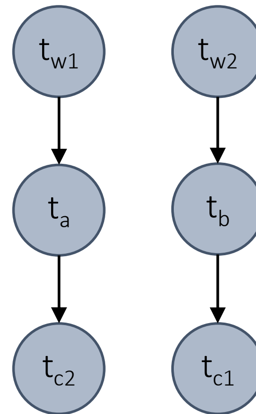
Minimum bounds:

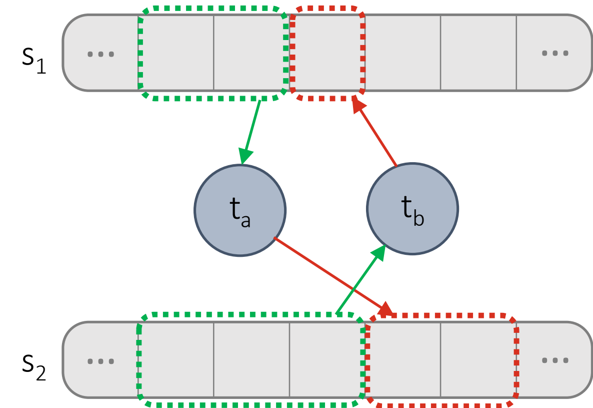
s_1: 2 elements

s_2: 3 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

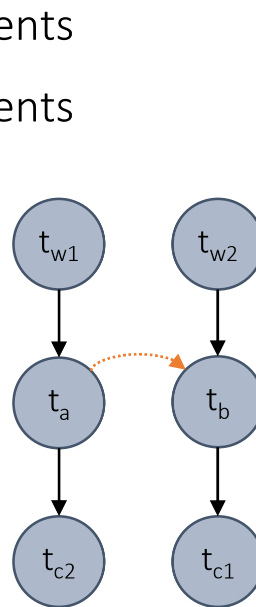
Minimum bounds:

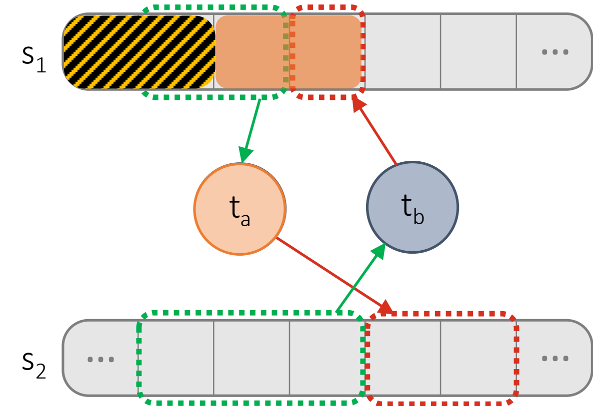
s_1: 2 elements

s_2: 3 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

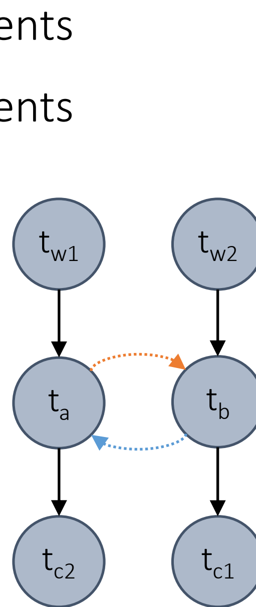
Minimum bounds:

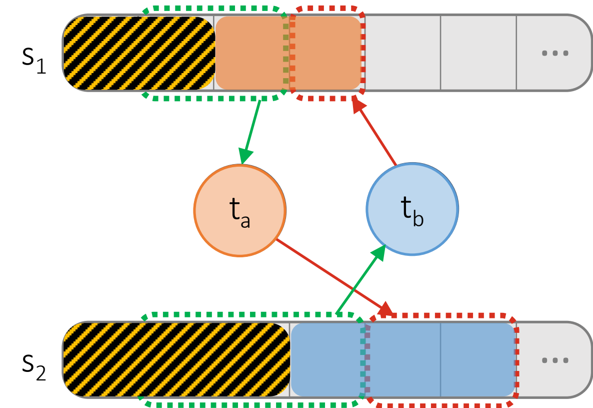
s_1: 2 elements

s_2: 3 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

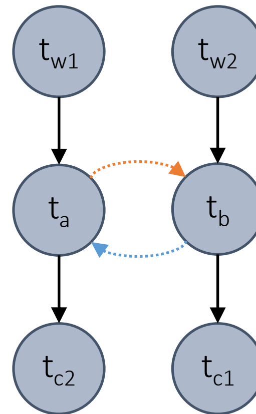
Minimum bounds:

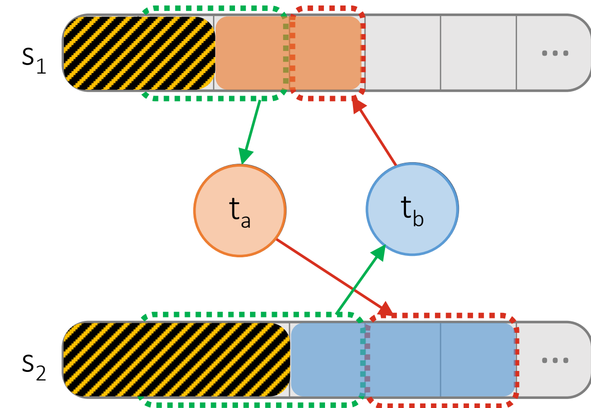
s_1: 2 elements

s_2: 3 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;
}
```





We can have one, but *not both*:

s_1: 2 elements & s_2: ≥5 elements

s_1: ≥3 elements & s_2: 3 elements

Stream bounding: global surface minimization

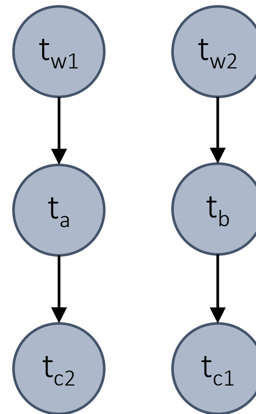
Minimum bounds:

s_1: 2 elements

s_2: 3 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;
}
```



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

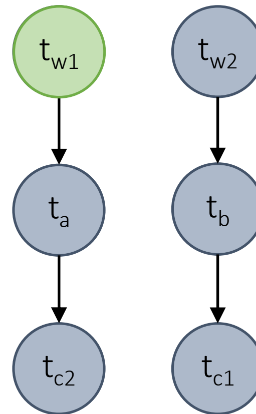
```
stream s1, s2;

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;
}
```

Minimum bounds:

s_1: 2 elements

s_2: 3 elements



s_1: 2 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

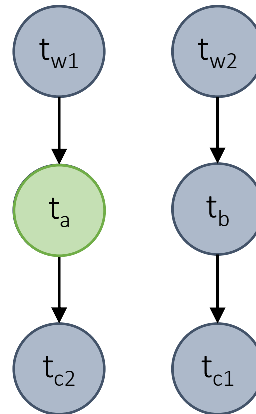
Minimum bounds:

s_1: 2 elements

s_2: 3 elements

```
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

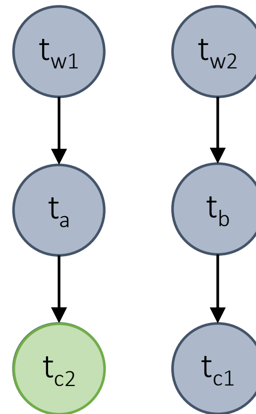
Minimum bounds:

s_1: 2 elements

s_2: 3 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;
}
```



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

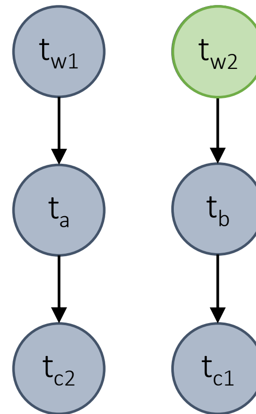
Minimum bounds:

s_1: 2 elements

s_2: 3 elements

```
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: 3 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

```
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: 1 element

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

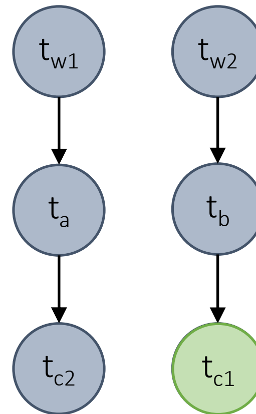
Minimum bounds:

s_1: 2 elements

s_2: 3 elements

```
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: application guidelines

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