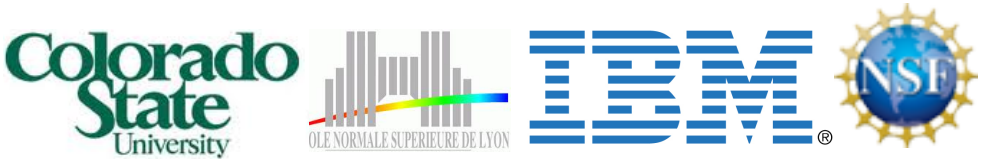
Colorado State University

Array Dataflow Analysis for Polyhedral X10 Programs

Tomofumi Yuki, Paul Feautrier,
Sanjay Rajopadhye, and Vijay Saraswat



Parallel Programming

- Parallel Programing is
  - Difficult
  - Time consuming
- Solutions:
  - Automatic parallelization
    - Only partially achieved
  - Highly Productive Parallel Languages
    - e.g., X10, Chapel, UPC, …
    - Still more difficult than sequential programs

Race Detection

- Parallelism comes with non-determinacy
  - Source of parallel bugs (races)
- Finding parallel bugs is extremely difficult
  - Not (consistently) reproducible
  - Static analysis tend to be too conservative
- Highly-Productive Languages:
  - Still require programmers to think parallel
  - Cannot help with races (at the language level)

Polyhedral X10

- `finish/async`
  - `async S`: Spawn a new *activity* to execute `S`
  - `finish S`: Wait for all activities in `S` to terminate
- Unsupported parallel constructs
  - `atomic/at` (places): minor extensions
  - `clocks`: major extension (on-going)
- Loop bounds and array accesses must be affine

`finish/async` vs `doall`

- Some can be viewed as `doall`

```
finish {
    for (i=0:N) {
        async S0;
    }
}
```

≈

```
forall (i=0:N) {
        S0;
}
```

- However, X10 is more expressive

```
for (i=0:N) {
    S0;
    async S1;
}
```

≈

?

`finish/async` vs `doall`

- Some can be viewed as `doall`

```
finish {
    for (i=0:N) {
        async S0;
    }
    async S1;
}
```

~

```
forall (i=0:N) {
        S0;
}
!
```

Key Challenge:
How to analyze such programs?

Contributions

- Scope: Polyhedral X10 programs
  - Subset of X10; affine loops + `finish/async`
- Succinct characterization of happens-before
  - Algorithm is 5 lines
- Extension of Array Dataflow Analysis
  - To `finish/async` programs
  - Results applied to race detection
- Prototype implementation

Example

- S2<i> use value of
  - S0<i> if 0≤i≤N
  - S1<i> if N≤i≤2N

```
finish {
    for (i=0:N) {
        async X[i] = S0();
    }
    for (i=N:2N) {
        async X[i] = S1();
    }
}
for (i=0:2N) {
    S2(X[i]);
}
```

Example

- S2<i> use value of
  - S0<i> if 0≤i≤N
  - S1<i> if N≤i≤2N
- Race Detection
  - Source of S0<i> overlap at i=N

```
finish {
    for (i=0:N) {
        async X[i] = S0();
    }
    for (i=N:2N) {
        async X[i] = S1();
    }
}
for (i=0:2N) {
   S2(X[i]);
}
```

Example

- S2<i> use value of
  - S0<i> if 0≤i≤N
  - S1<i> if N≤i≤2N
- Race Detection
  - Source of S0<i> overlap at i=N
- Feedback to user
  - Read X[i] of S2<i> has two sources S0<i> and S1<i> when i=N

```
finish {
    for (i=0:N) {
        async X[i] = S0();
    }
    for (i=N:2N) {
        async X[i] = S1();
    }
}
for (i=0:2N) {
    S2(X[i]);
}
```

Outline

- Introduction
- Polyhedral X10
- Array Dataflow Anaysis
- Happens-Before Relation
- Race Detection
- Conclusions

Happens-Before Relation

- `A` happens-before `B`
  - Result of `A` is visible to `B` in all possible orders of execution
- Instance-wise Happens-Before
  - `A<i,j>` happens-before `B<x,y>`
  - Result of `A` at iteration `<i,j>` is visible to `B` at iteration `<x,y>` in all possible execution

Array Data-flow Analysis

- Exact dependence analysis
- Statement instance-wise
  - e.g., Value produced by `A` at iteration `<i,j>` is used by `B` at iteration `<x,y>`
- Array element-wise
  - e.g., Value written to array element `X[i,j]` by `A<i,j>` is used by `B<x,y>`
- Original analysis is for sequential loop nests

ADA Formulation

- Given statement instances
  - `r:` reader
  - `w:` writer
- Candidate producers for `r` are `w` where:
  - `r` and `w` are valid iterations
  - `r` and `w` access the same memory location
  - `w` happens-before `r`
- Then find the most recent `w`
- Can be solved as Parametric ILP

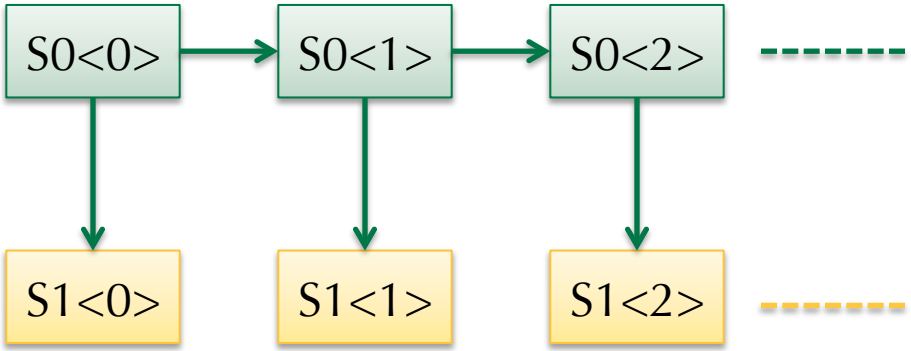
Re-formulating Happens-Before

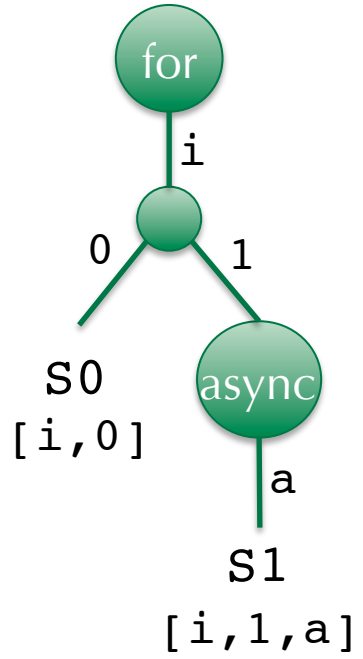
- Happens-Before for sequential program
  - Total order
  - Lexicographic order
- For parallel programs
  - Partial order
- How to re-formulate for `finish/async`?
  - In a way ILP can still be used

Happens-Before with Async

- When are the following true?
  - `S1<i>` happens-before `S1<i'>`
  - `S0<i>` happens-before `S1<i'>`

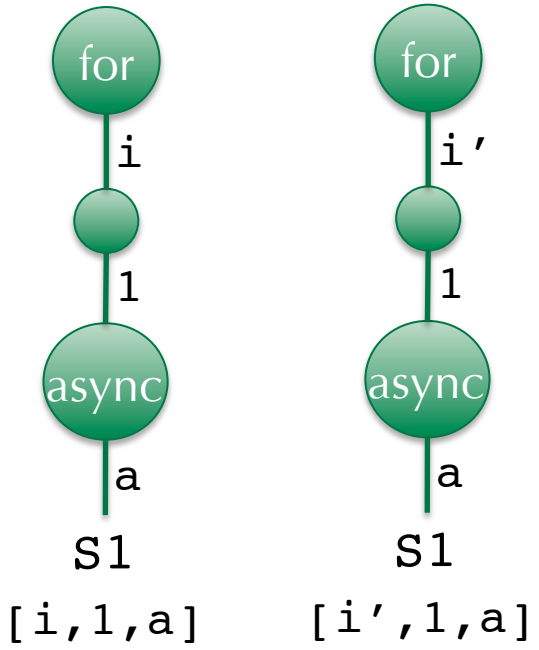
```
for (i=0:N) {
    S0;
    async S1;
}
```





When `async` matters

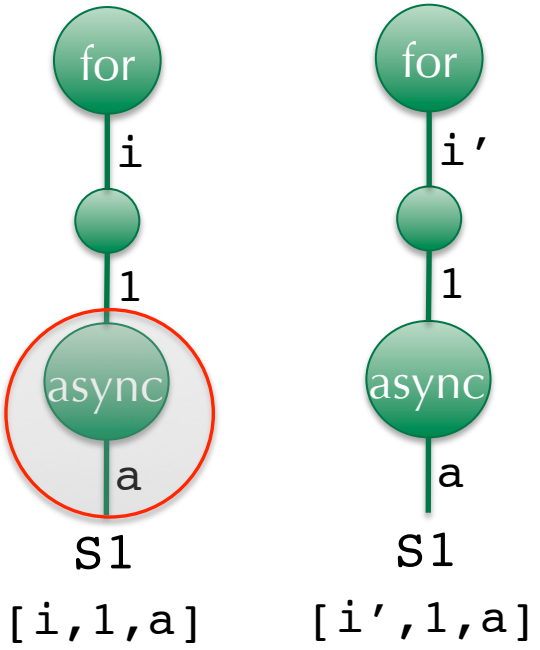
- `S1<i>` happens-before `S1<i'>`?



- `false`
- even if `i<i'` `S1<i>` may be executed after `S1<i'>`

When `async` matters

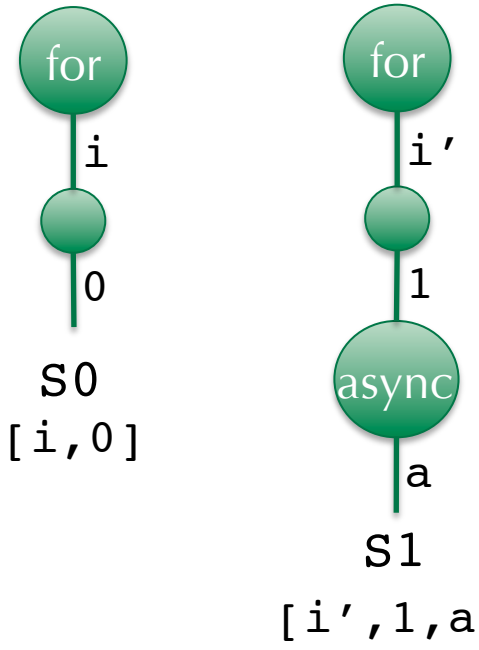
- `S1<i>` happens-before `S1<i'>`?



- `false`
- even if `i<i'` `S1<i>` may be executed after `S1<i'>`
- Intuition:
  `async` may only postpone the execution of its enclosing statements

When `async` does not matter

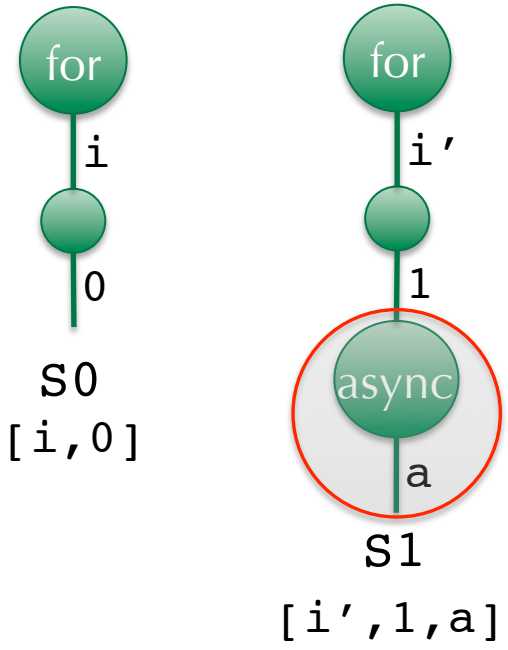
- `S0<i>` happens-before `S1<i'>`?



- `true` if `i≤i'`

When `async` does not matter

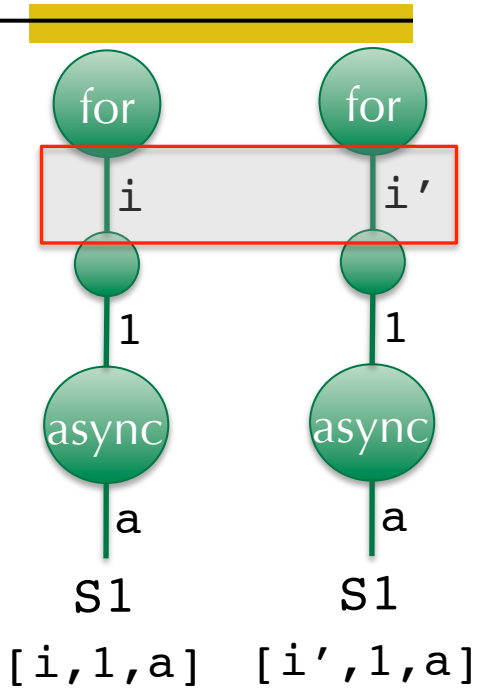
- `S0<i>` happens-before `S1<i'>`?



- `true` if `i≤i'`
- Intuition:
  `S0<i>` is completed before the activity that executes `S1<i'>` is spawned
- if `i=i'`, `S0` is still before `S1` in textual order `(0<1)`

Happens-Before as Incomplete Lexicographic Order

- Lexicographic order
  - Compare each dimension
  - 1st difference defines order
- Incomplete Lexicographic Order
  - Compare a *subset* of dimensions
- Intuition:
  - Some dimensions do not contribute
    - `async` not synchronized with `finish`



Outline

- Introduction
- Polyhedral X10
- Happens-Before Relation
- Array Dataflow Analysis
- Race Detection
- Conclusions

Applying to Race Detection

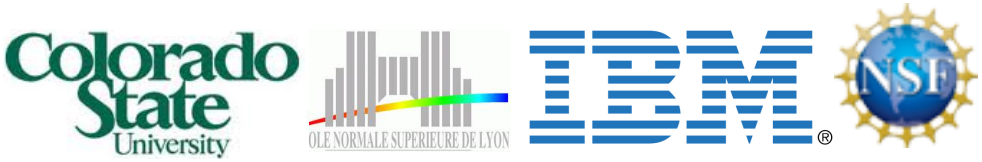
- ADA for sequential programs:
  - Happens-Before is *total*
  - Each read has **exactly one** producer
- ADA for parallel programs:
  - Happens-Before is *partial*
  - The source **may not be unique**
- If the source is ambiguous for a read
  - We have a data race!
  - ADA result can also help fixing the problem

Related Work

- ADA for `doall` parallelism [Collard and Griebl 1996]
  - Cannot handle `finish/async`
- Instance-wise Happens-Before [Agarwal et al. 2007]
  - Not linked to dependence analysis
  - Our formulation is simpler
  - Handles `at` and places
- Instance-wise race detection [Vechev et al. 2010]
  - Array accesses are over-approximated

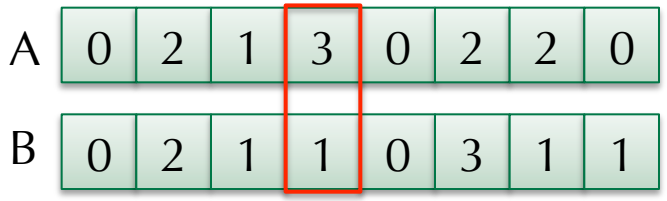
Conclusions

- Extended ADA to subset of X10 programs
- Application to race detection
  - More precise than prior work
  - Guarantees race-freedom at compile-time
- Future work
  - Handling of `clocks` (X10 synchronization)
  - Extending other analyses (e.g., scheduling)
  - Lifting the “polyhedral” restriction

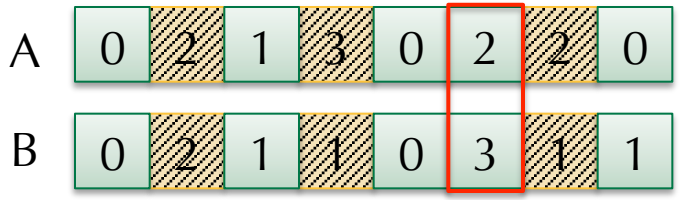

Colorado
State
University
OLE NORMALE SUPERIEURE DE LYON
IBM
NSF

Happens-Before as Incomplete Lexicographic Order

- Lexicographic order
  - Compare each dimension
  - 1st difference defines order

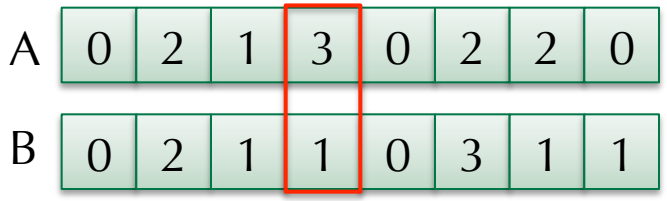


- Incomplete Lexicographic Order
  - Compare a *subset* of dimensions

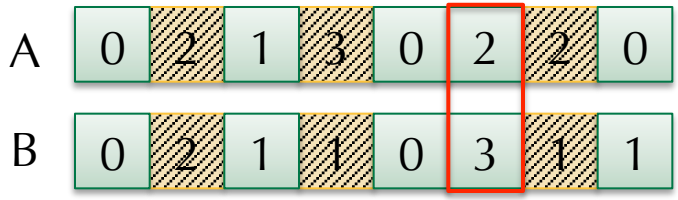


Happens-Before as Incomplete Lexicographic Order

- Lexicographic order
  - Compare each dimension
  - 1st difference defines order



- Incomplete Lexicographic Order
  - Compare a *subset* of dimensions



- Intuition:
  - Parallel dimensions do not contribute
  - Remove parallel iterations from lex. order

Contributions (old)

- Scope: Polyhedral X10 programs
  - Subset of X10; affine loops + `finish/async`
- Extension of Array Dataflow Analysis
  - Instance-wise and Element-wise analysis
  - Analyze `finish/async` parallel programs
  - Apply its results to race detection
- Prototype implementation
  - Can be (eventually) integrated IDEs to flag races while coding