Static Analysis of Parallel Programs with Loops, Tasks, and Synchronizations

Tomofumi Yuki
Inria / LIP / ENS Lyon

Journée LSC

High Performance Computing

- Massively parallel computing
- No such things as fast “enough”
- Speed is translated into
  - Faster simulation
  - Better resolution
- Examples
  - Climate modeling
  - Simulating nuclear physics

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Automatic Parallelization?

- Limited Success
  - polyhedral model
  - instruction-level parallelism
- Compilers are **good** at
  - taking care of the “details”
  - handling small blocks of code
- Compilers are **bad** at
  - raising the level of abstractions
  - using domain specific knowledge

Automatic Parallelization?

- Limited S ccess
  - polyhe model
  - instruction l parall n
- Co
  - t
  - handling sm ocks code
- Compilers not capabl f
  - raisin e level of abstracti
  - using domain specific knowledge

Not “the” Solution

Parallel Programming?

- Extremely difficult
  - parallel bugs
  - “think parallel”
- Parallel Programming Models
  - Productivity vs. Performance
  - Emerging languages
    - X10, Chapel, UPC, ...
  - Domain Specific
    - CUDA, task-graphs, ...

Parallel Programming?

- Extremel difficult
  - paralle s
  - “think para
- Parallel Progr i odels
  - Pr
  - Emerging la ages
    - X10, C , UPC, ...
  - Domai pecific
    - CUDA, task-graphs, ...

Low Productivity

Data Races

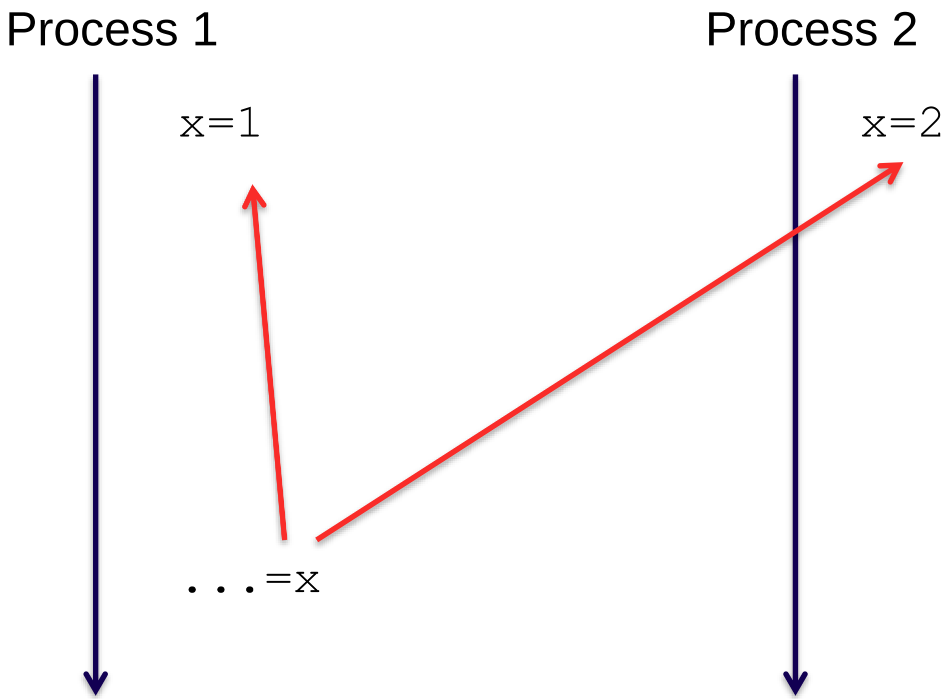
- Parallelism comes with non-determinacy
  - Source of parallel bugs (races)
- Finding parallel bugs is extremely difficult
  - Not (consistently) reproducible
  - Static analysis tend to be too conservative
- Productivity Oriented Languages:
  - Still require programmers to think parallel

Data Races

- Definition:
  - *concurrent* + *conflicting* access
- Concurrent
  - Two operations are concurrent if the two are not ordered
- Conflicting access
  - Accesses the same memory location
  - At least one of the accesses is a write

Data Race Example

- Shared memory



This Talk

- Static Analysis of *Parallel* Programs
- Result: Array Dataflow Analysis for X10
  - extending the core of the polyhedral model
  - applied to data race detection
- Key: modeling the execution order
  - no longer total order
  - not just **`doall`**
  - goal: reuse polyhedral machinery

Context: Loop Transformations

- Key to expose parallelism
  - some times it’s easy

```
for i
  for j
    X[i] += ...
```

```
for i
  forall j
    X[i] += ...
```

  - but not always

```
for i = 0 .. N
  for j = 1 .. M
    X[j] = foo(X[j-1],
               X[j+1]);
```

```
for i = 1 .. 2N+M
  forall j = /*complex bounds*/
    X[j] = foo(X[2*j-i-1],
               X[2*j-i+1]);
```

Automatic Parallelization

- Very sensitive to inputs

```
for (i=1; i<N; i++)
  for (j=1; j<M; j++)
     x[i][j] = x[i-1][j] + x[i][j-1];

for (i=1; i <N-1; i++)
  for (j=1; j<M-1; j++)
     y[i][j] = y[i-1][j] + y[i][j-1] + x[i+1][j+1];
```

```
 for (t1=2;t1<=3;t1++) {
      lbp=1;
      ubp=t1-1;
#pragma omp parallel for private(lbv,ubv,t3)
      for (t2=lbp;t2<=ubp;t2++) {
        S1((t1-t2),t2);
      }
    }

 for (t1=4;t1<=min(M,N);t1++) {
    S1((t1-1),1);
    lbp=2;
    ubp=t1-2;
          #pragma omp parallel for
private(lbv,ubv,t3)
    for (t2=lbp;t2<=ubp;t2++) {
      S1((t1-t2),t2);
      S2((t1-t2-1),(t2-1));
    }
    S1(1,(t1-1));
  }
```

```
for (t1=M+1;t1<=N;t1++) {
      S1((t1-1),1);
      lbp=2;
      ubp=M-1;
#pragma omp parallel for private(lbv,ubv,t3)
```

very difficult to understand

➔trust it or not use it

```
        S1((t1-t2),t2);
        S2((t1-t2-1),(t2-1));
      }
      S1(1,(t1-1));
    }
  }
```

```
for (t1=max(M+1,N+1);t1<=N+M-2;t1++) {
    lbp=t1-N+1;
    ubp=M-1;
          #pragma omp parallel for private(lbv,ubv,t3)
```

Expressing with X10

- Goal: retain the original structure

```
async
  for (i=1; i<N; i++)
    advance;
    async
      for (j=1; j<M; j++)
        x[i][j] = x[i-1][j] + x[i][j-1];
       advance;
advance;
async
  for (i=1; i <N-1; i++)
     advance;
     async
       for (j=1; j<M-1; j++)
         y[i][j] = y[i-1][j] + y[i][j-1] + x[i+1][j+1];
         advance;
```

Outline

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

Polyhedral X10

- Key parallel constructs
  - **`async S`**: Spawn a new *activity* to execute `S`
  - **`finish S`**: Wait for all activities in `S` to terminate
  - **`clocks`**: Synchronization in X10
- Loop bounds and array accesses must be affine
- A few more constructs
  - **`at/places`**: PGAS element
  - **`atomic`**: critical section

`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?

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
  - Producer of `X[i]` at `S2` 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
  - Producer of `X[i]` at `S2` 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 Analysis
- Happens-Before Relation
- Race Detection
- Clocks
- 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 Dataflow 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` *(total order)*
- Then find the most recent `w`
- Can be solved as Parametric ILP

Re-formulating Happens-Before

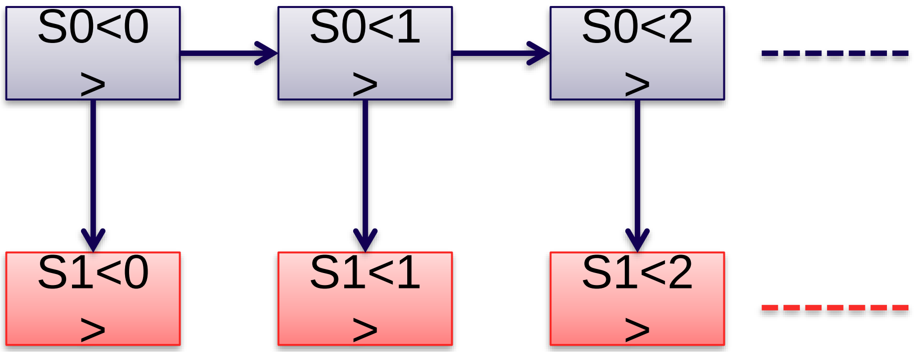
- Happens-Before for sequential program
  - Total order
  - Lexicographic order
- For parallel programs
  - Partial order
- How to re-formulate for `finish/async`?
  - In a way ILP can still be used

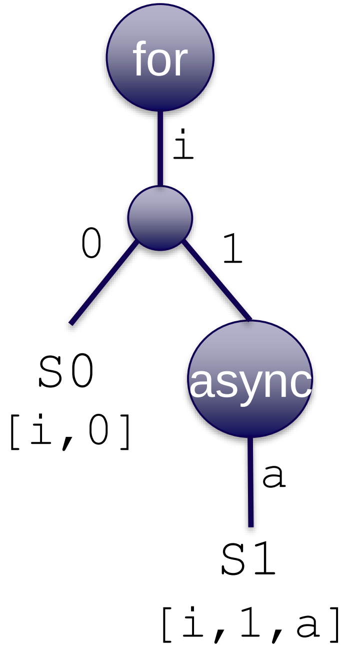
INVENTEURS DU MONDE NUMÉRIQUE

Happens-Before with Async

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

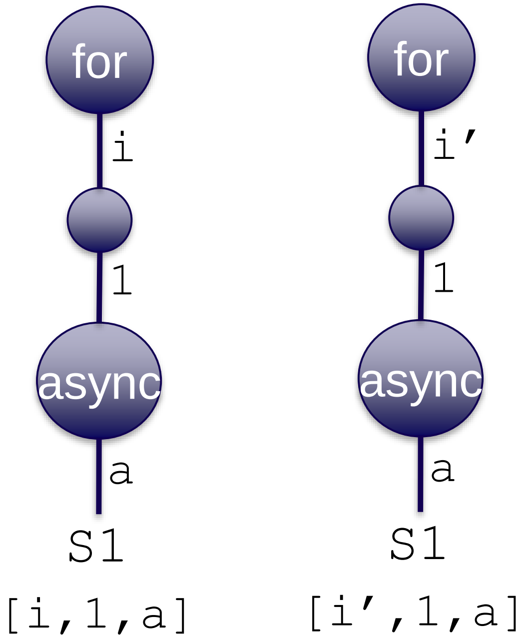
```
for (i=0:N) {
    S0;
   async S1;
}
```





When `async` matters

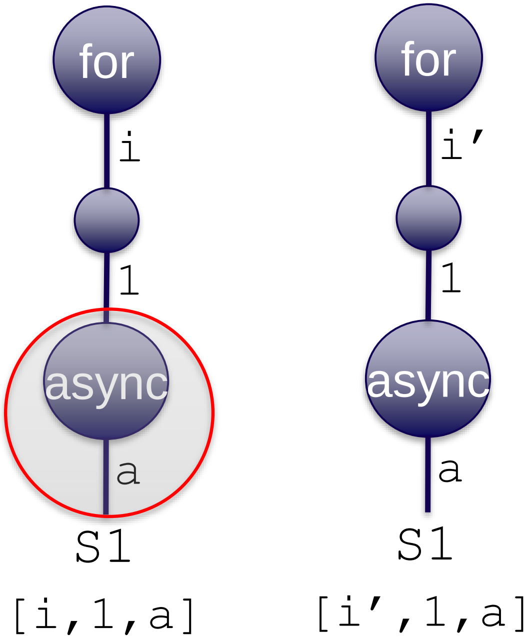
- `S1<i>` happens-before `S1<i'>`?



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

When `async` matters

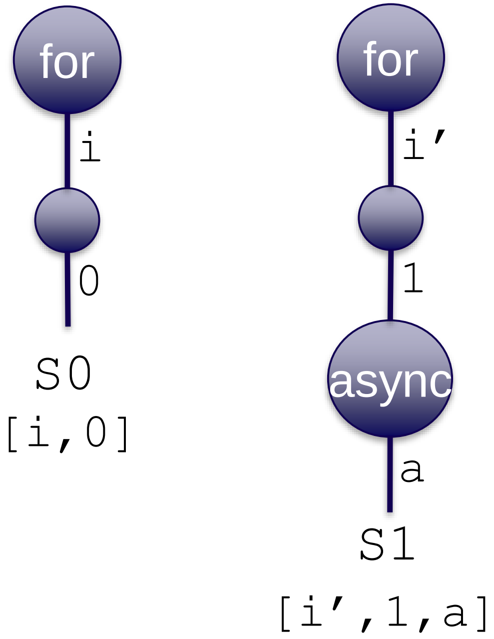
- `S1<i>` happens-before `S1<i'>`?



- `false`
- even if `i<i'` `S1<i>` may be executed after `S1<i'>`
- Intuition: `async` makes two instances unordered w.r.t. `i` iterator

When `async` does not matter

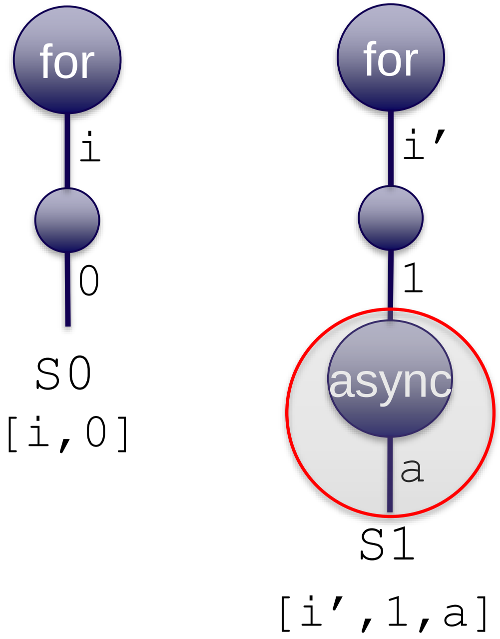
- `S0<i>` happens-before `S1<i'>`?



- `true` if `i≤i'`

When `async` does not matter

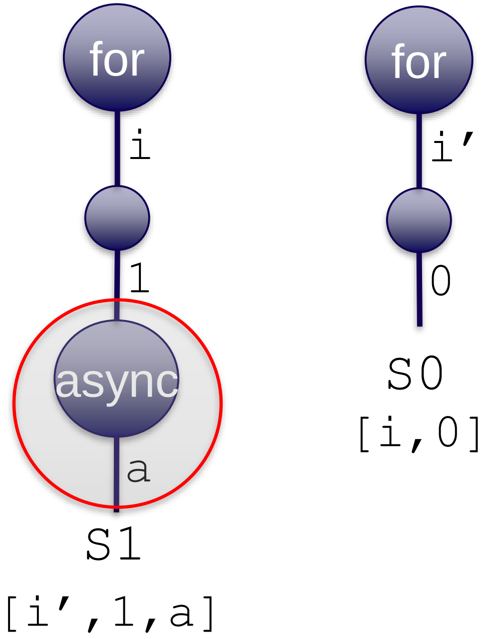
- `S0<i>` happens-before `S1<i’>`?



- `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)`

Asymmetric Relation

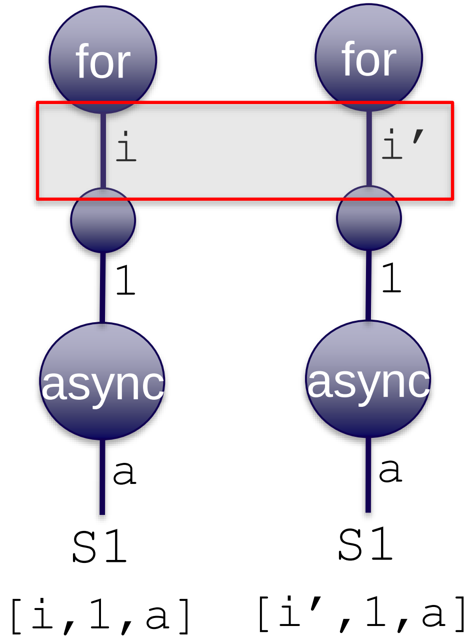
- `S1<i>` happens-before `S0<i'>`?



- `false`
- Intuition: you know
  when `S1<i>` spawns
  when `S0<i'>` happens
- But not
  when `S1<i>` is (surely) done

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 by `finish`



ADA Formulation (partial order)

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

Outline

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

Outline

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

Clocks

- Synchronization in X10
  - more dynamic variant of barriers
- We extended data race detection to clocks
  - requires counting
  - polynomials → undecidable

- I won’t talk about data race with clocks today

`clocks` VS `barriers`

- Barriers can easily deadlock

```
//P1
barrier;
S0;
barrier;
```

```
//P2
barrier;
S1;
```

- Clocks are more dynamic

```
//P1
advance;
S0;
advance;
```

```
//P2
advance;
S1;
```

clocks VS barriers

- Barriers can easily deadlock

```
//P1
barrier;
S0;
barrier;
```

deadlock

```
//P2
barrier;
S1;
```

- Clocks are more dynamic

```
//P1
advance;
S0;
advance;
```

```
//P2
advance;
S1;
```

`clocks` VS `barriers`

- Barriers can easily deadlock

```
//P1
barrier;
S0;
barrier;
```

deadlock

```
//P2
barrier;
S1;
```

- Clocks are more dynamic

```
//P1
advance;
S0;
advance;
```

OK

```
//P2
advance;
S1;
```

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
    for (i=1:N)
        clocked async {
            for (j=i:N)
                advance;
                S0;
        }
```

← Creation of a clock

← Each process is registered

← Sync registered processes

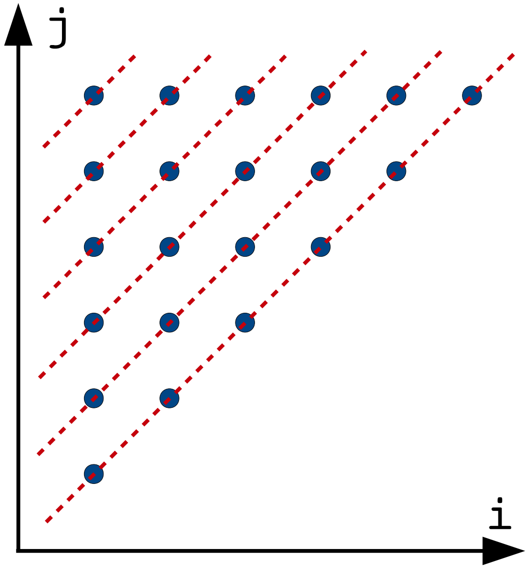
← Each process is un-registered

- The process creating a clock is also registered

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
   for (i=1:N)
      clocked async {
         for (j=i:N)
            advance;
            S0;
      }
```

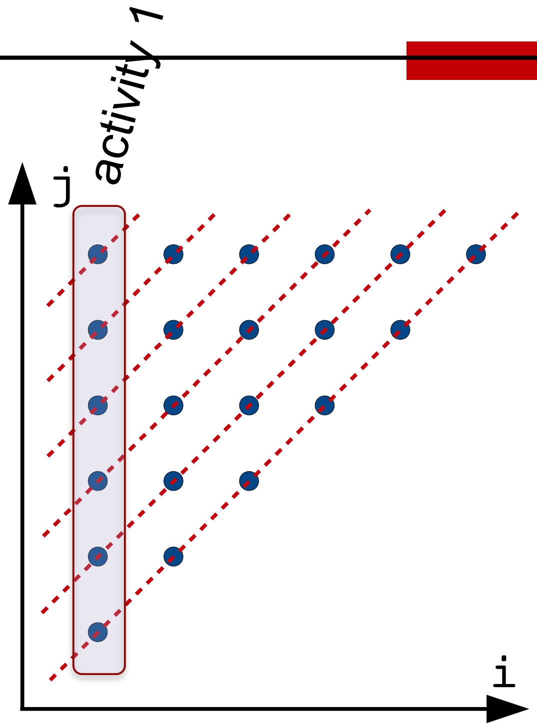


- Each process waits until all processes starts
  - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
    for (i=1:N)
        clocked async {
            for (j=i:N)
                advance;
                S0;
        }
```

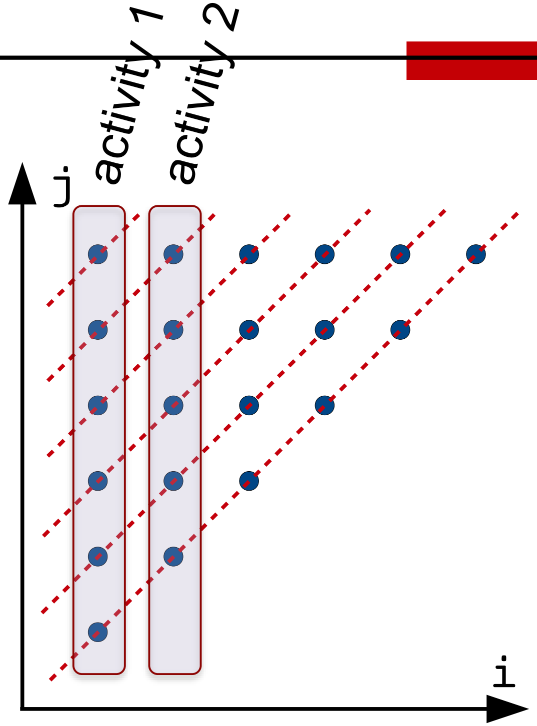


- Each process waits until all processes starts
    - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
    for (i=1:N)
        clocked async {
            for (j=i:N)
                advance;
                S0;
        }
```

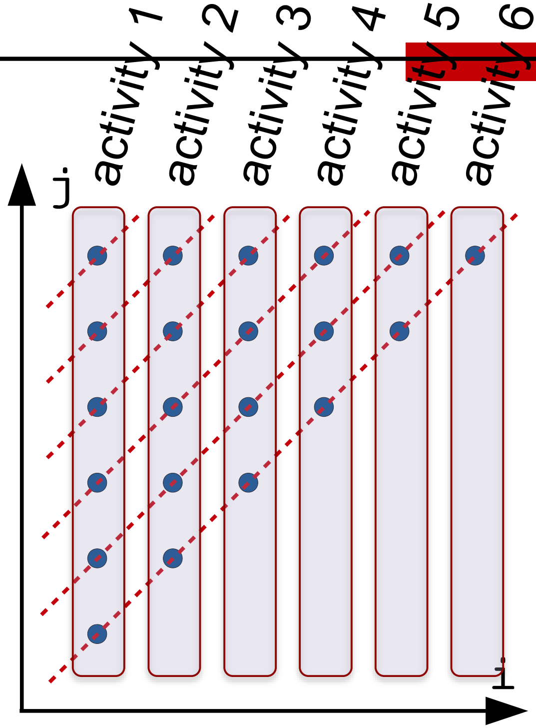


- Each process waits until all processes starts
  - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
    for (i=1:N)
        clocked async {
            for (j=i:N)
                advance;
                S0;
        }
```

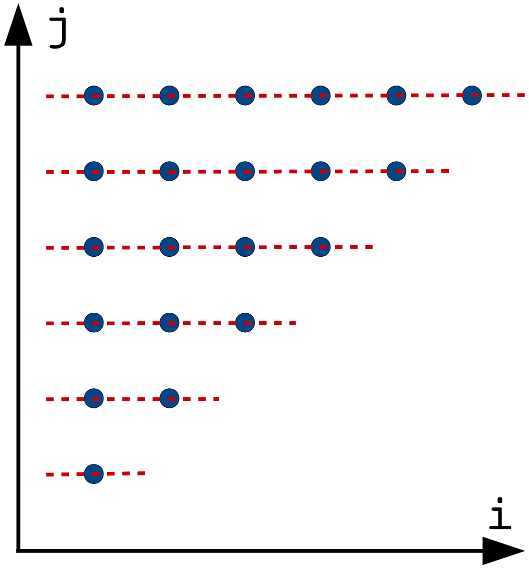


- Each process waits until all processes starts
  - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
    for (i=1:N) {
        clocked async {
            for (j=i:N)
                advance;
                S0;
        }
        advance; }
```

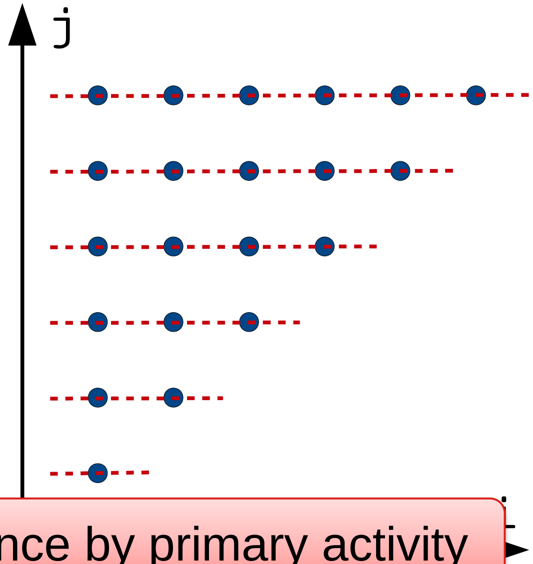


- The primary process calls advance each time
    - Different synchronization pattern

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
    for (i=1:N) {
        clocked async {
            for (j=i:N)
                advance;
                S0;
        }
        advance; }
```



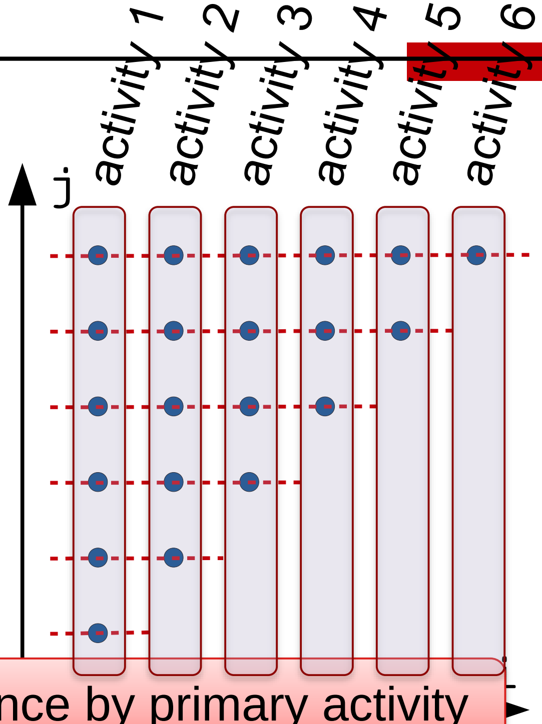
advance by primary activity

- The primary process calls advance each time
  - Different synchronization pattern

Dynamicity of Clocks

- Implicit Syntax

```
clocked finish
    for (i=1:N) {
        clocked async {
            for (j=i:N)
                advance;
                S0;
        }
        advance; }
```

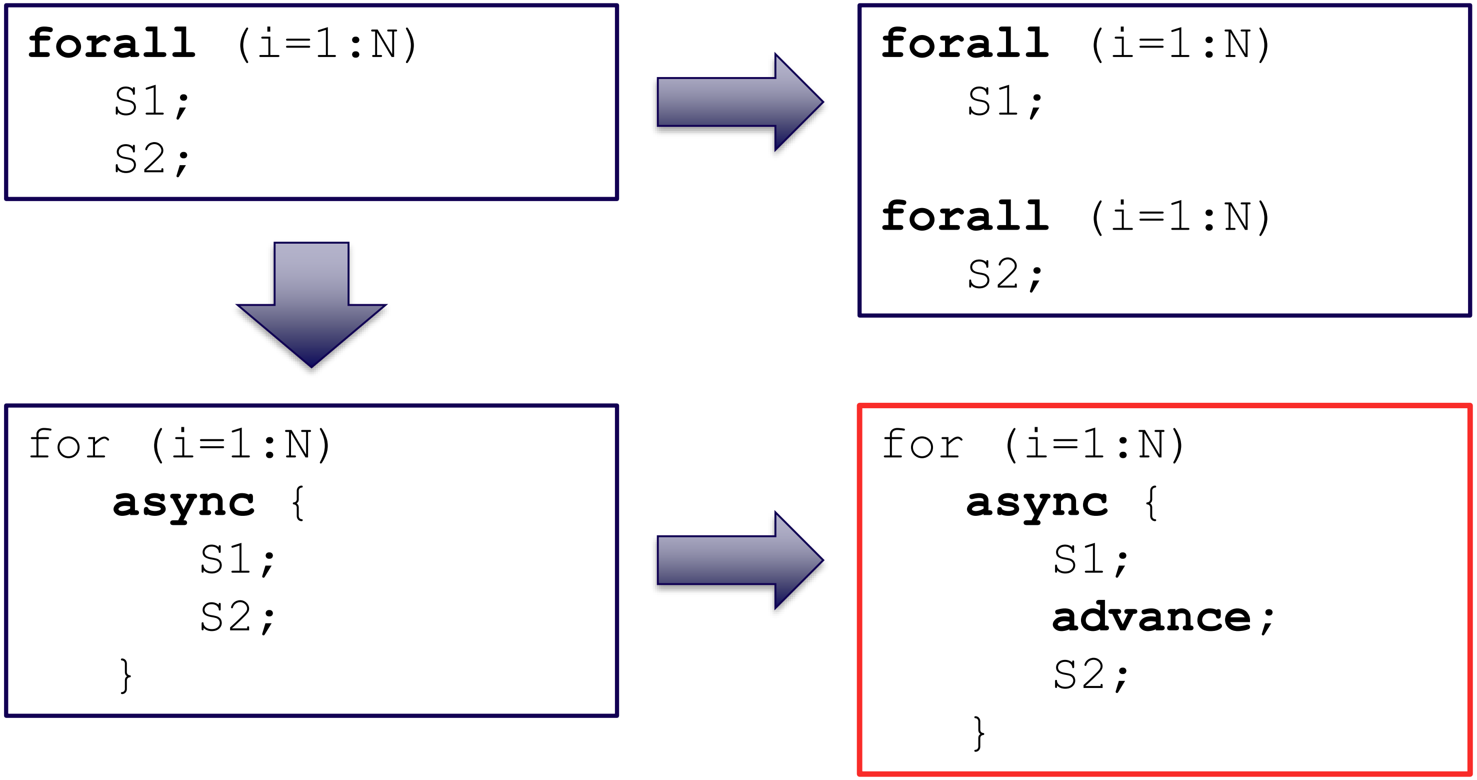


advance by primary activity

- The primary process calls advance each time
  - Different synchronization pattern

Example: Loop Fission

- Common use of barriers



```
forall (i=1:N)
    S1;
    S2;
```

```
forall (i=1:N)
    S1;

forall (i=1:N)
    S2;
```

```
for (i=1:N)
    async {
        S1;
        S2;
    }
```

```
for (i=1:N)
    async {
        S1;
        advance;
        S2;
    }
```

INVENTEURS DU MONDE NUMÉRIQUE

Example: Loop Fusion

- Removes all the parallelism

```
for (i=1:N)
    S1;
for (i=1:N)
    S2;
```

```
for (i=1:N)
    S1;
    S2;
```

```
async
    for (i=1:N)
      S1; advance; advance;
advance;
async
    for (i=1:N)
      S2; advance; advance;
```

Example: Loop Fusion

- Sometimes fusion is not too simple

```
for (i=1:N-1)
   S1(i);
for (i=2:N)
   S2(i);
```

```
S1(1);
   for (i=2:N-1)
      S1(i);
      S2(i);
S2(N);
```

```
async
   for (i=1:N-1)
      S1; advance; advance;
advance;
async
   for (i=2:N)
      S2; advance; advance;
```

code structure stays
# of advance ➔ control

Expressing with Clocks

- Goal: retain the original structure

```
async
  for (i=1; i<N; i++)
    advance;
    async
      for (j=1; j<M; j++)
        x[i][j] = x[i-1][j] + x[i][j-1];
       advance;
advance;
async
  for (i=1; i <N-1; i++)
     advance;
     async
       for (j=1; j<M-1; j++)
         y[i][j] = y[i-1][j] + y[i][j-1] + x[i+1][j+1];
         advance;
```

Expressing with Clocks

- Goal: retain the original structure

```
async
  for (i=1; i<N; i++)
    advance;
    async
      for (j=1; j<M; j++)
        x[i][j] = x[i-1][j] + x[i][j-1];
        advance;
advance;
async
  for (i=1; i <N-1; i++)
     advance;
     async
       for (j=1; j<M-1; j++)
         y[i][j] = y[i-1][j] + y[i][j-1] + x[i+1][j+1];
         advance;
```

1. make many iterations parallel

Expressing with Clocks

- Goal: retain the original structure

```
async
  for (i=1; i<N; i++)
    advance;
    async
      for (j=1; j<M; j++)
        x[i][j] = x[i-1][j] + x[i][j-1];
        advance;
advance;
async
  for (i=1; i <N-1; i++)
     advance;
     async
       for (j=1; j<M-1; j++)
         y[i][j] = y[i-1][j] + y[i][j-1] + x[i+1][j+1];
         advance;
```

1. make many iterations parallel

2. order them by synchronizations

Expressing with Clocks

- Goal: retain the original structure

```
async
  for (i=1; i<N; i++)
    advance;
    async
      for (j=1; j<M; j++)
        x[i][j] = x[i-1][j] + x[i][j-1];
       advance;
advance;
async
  for (i=1; i <N-1; i++)
     advance;
     async
       for (j=1; j<M-1
         y[i][j] = y[i
         advance;
```

1. make many iterations parallel

2. order them by synchronizations

compound effect: parallelism similar to those with loop trans.

What can be expressed?

- Limiting factor: parallelism
  - difficult to use for sequential loop nests
  - works for wave-front parallelism

- Intuition
  - `clocks` *defer* execution
  - deferring parent activity has *cumulative* effect

Is it actually easier?

- Learning curve
  - behavior of `clock`
  - takes time to understand

- How much can you express?
  - 1D affine schedules for sure
  - loop permutation is not possible
  - what if we use multiple clocks?

Potential Applications

- It might be easier for some people
  - have multiple ways to write code
- Detect X10 fragments with such property
  - convert to `forall` for performance

Conclusions

- Static Analysis of Parallel Programs
  - powerful for a restricted scope
  - important for parallel programmer productivity

- Future Work
  - polyhedral is too restrictive
  - less powerful but more general framework