Revisiting Loop Transformations with X10 Clocks

Tomofumi Yuki
Inria / LIP / ENS Lyon
X10 Workshop 2015

The Problem

- The Parallelism Challenge
  - cannot escape going parallel
  - parallel programming is hard
  - automatic parallelization is limited
- There won't be any Silver Bullet
- X10 as a partial answer
  - high-level language with parallelism in mind
  - features to control parallelism/locality

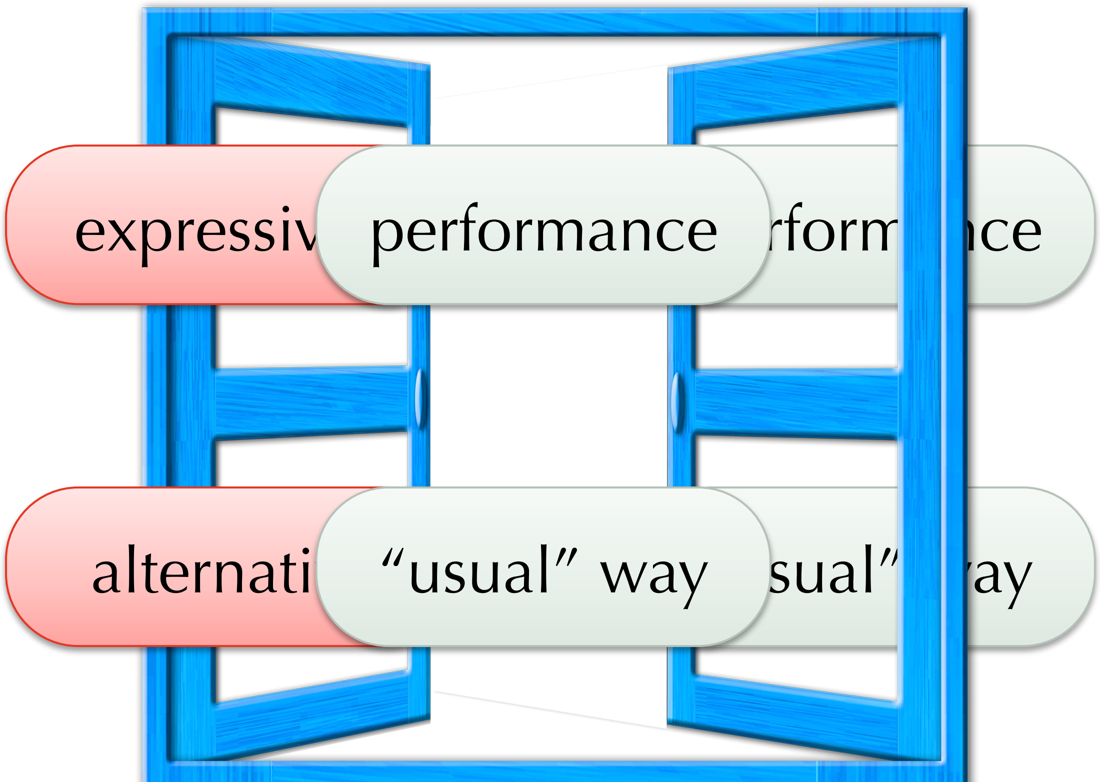
Programming with X10

- Small set of parallel constructs
  - `finish/async`
  - `clocks`
  - `at` (places), `atomic`, `when`
- Can be composed freely
- Interesting for both programmer and compilers
  - also challenging

But, it seems to be under-utilized

This Paper

- Exploring how to use X10 clocks



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
   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)
        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)
                 =ubp;t2++) {
                 );
                 (t2-1));
```

very difficult to understand
➔trust it or not use it

Expressing with Clocks

- Goal: retain the original structure

```
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];
```

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.

Outline

- Introduction
- X10 Clocks
- Examples
- Discussion

`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;
```

INVENTEURS DU MONDE NUMÉRIQUE

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

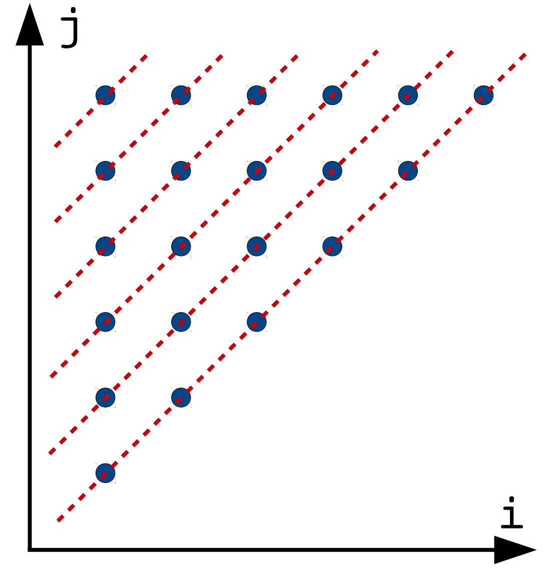
← Each process is un-registered

- 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;
        }
```

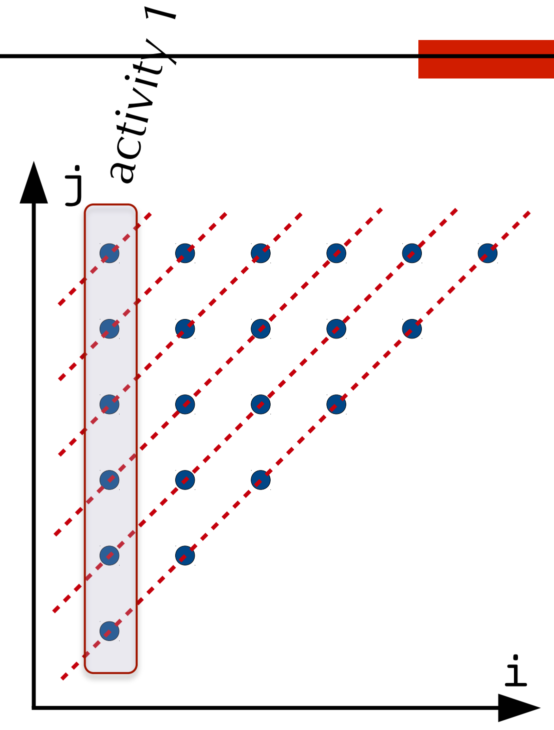


- Each process waits until all processes starts
  - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

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

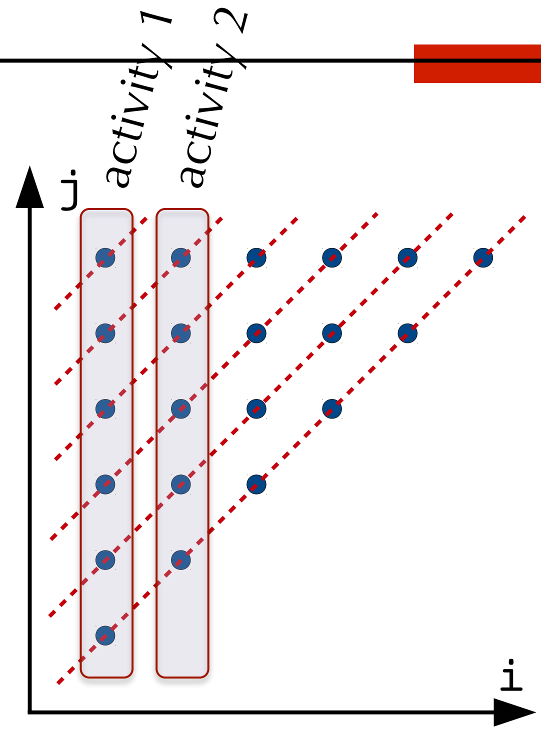


- Each process waits until all processes starts
  - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

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

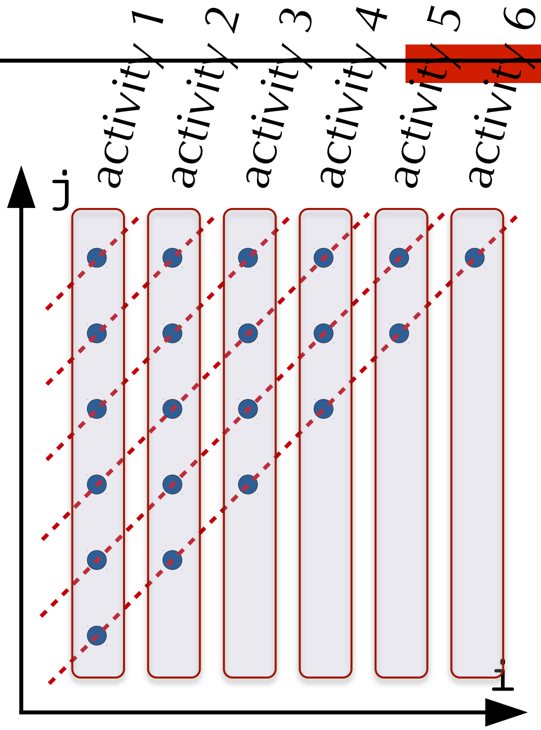


- Each process waits until all processes starts
    - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

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

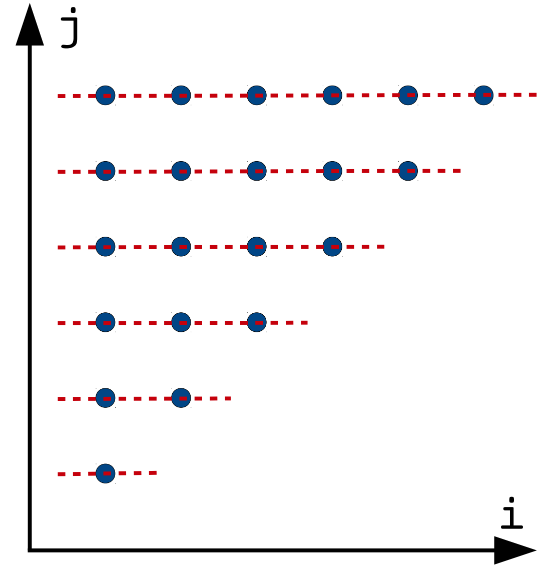


- Each process waits until all processes starts
  - The primary process has to terminate first

Dynamicity of Clocks

- Implicit Syntax

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

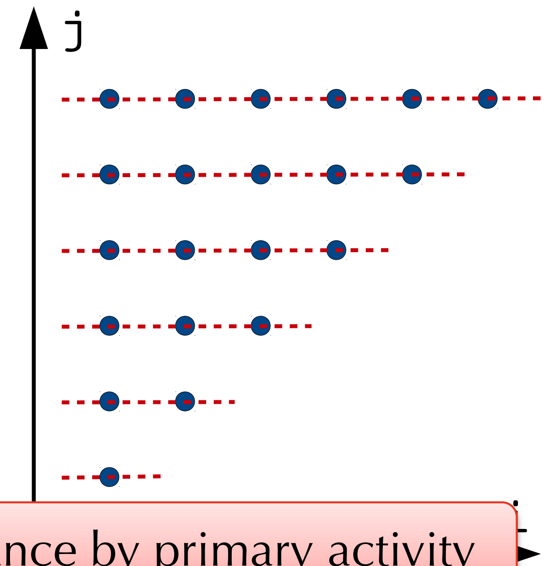


- The primary process calls advance each time
  - Different synchronization pattern

Dynamicity of Clocks

- Implicit Syntax

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



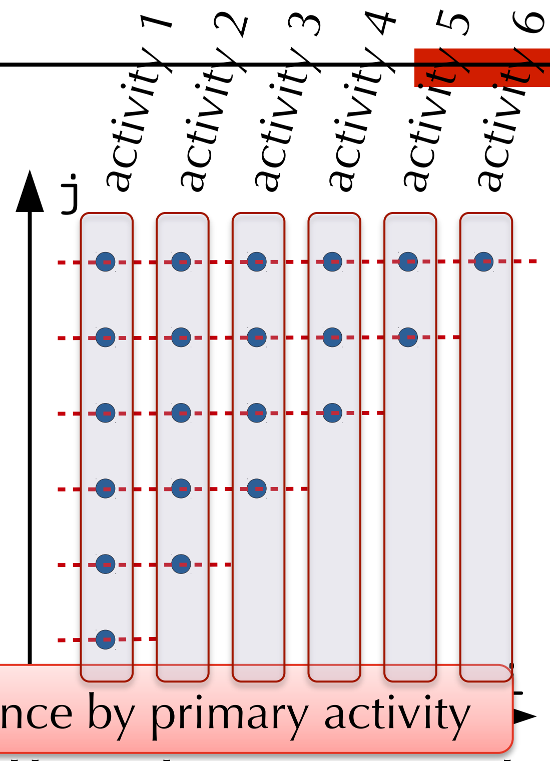
advance by primary activity

- 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; }
```



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

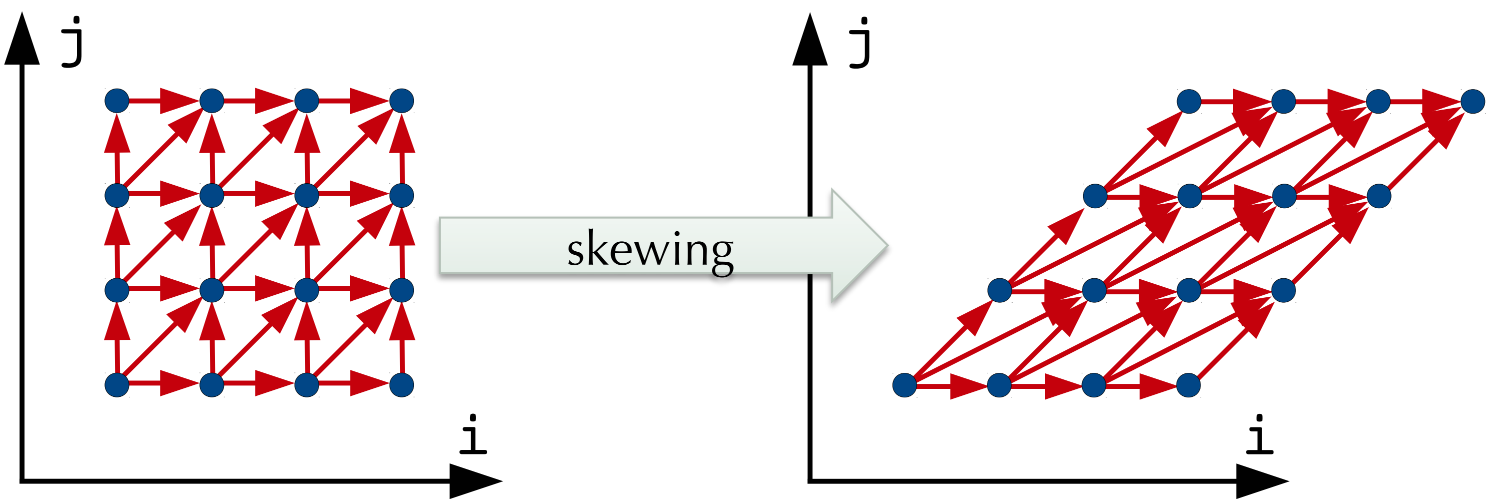
Outline

- Introduction
- X10 Clocks
- Examples
- Discussion

Example: Skewing

- Skewing the loops is not easy

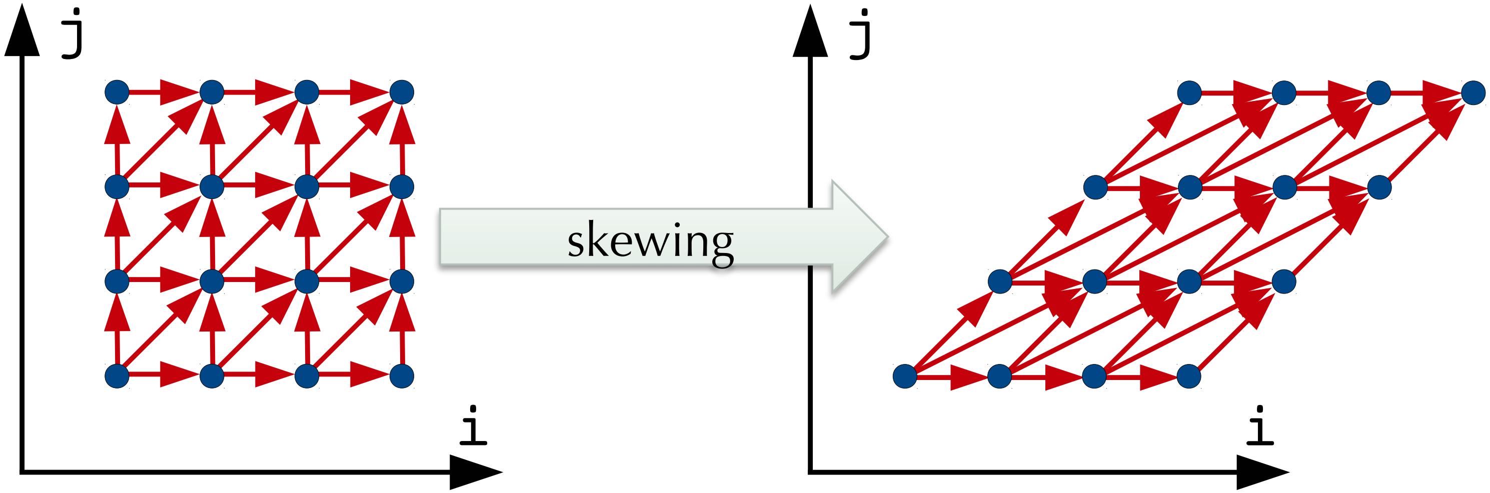
```
for (i=1:N)
   for (j=1:N)
      h[i][j] = foo(h[i-1][j],
                    h[i-1][j-1],
                    h[i][j-1])
```



Example: Skewing

- Skewing the loops is not easy

```
for (i=1:2N-1)
   forall (j=max(1,i-N):min(N,i-1))
      h[i][j] = foo(h[(i-j)-1][j],
                    h[(i-j)-1][j-1],
                    h[(i-j)][j-1])
```

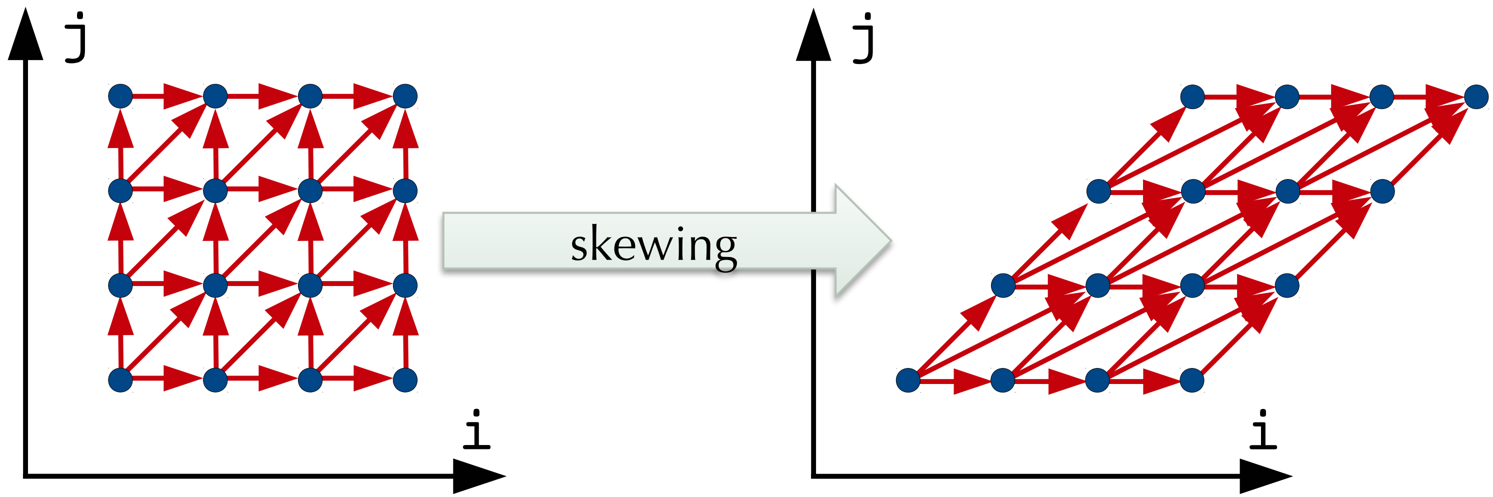


Example: Skewing

- Skewing the loops is not easy

changes to
loop bounds and indexing

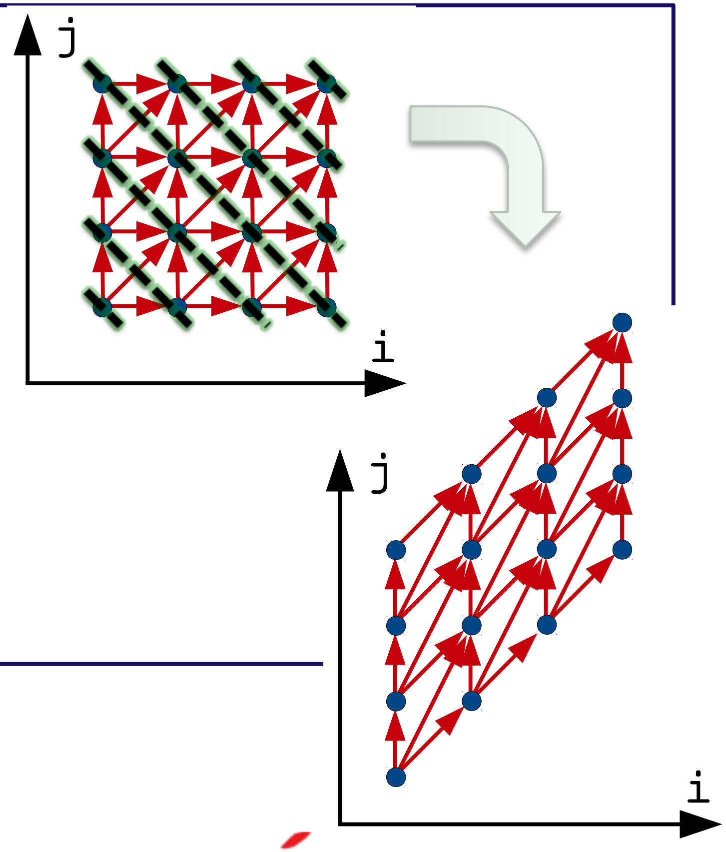
```
for (i=1:2N-1)
    forall (j=max(1,i-N):min(N,i-1))
        h[i][j] = foo(h[(i-j)-1][j],
                      h[(i-j)-1][j-1],
                      h[(i-j)][j-1])
```



Example: Skewing

- Equivalent parallelism without changing loops

```
clocked finish
  for (i=1:N) {
    clocked async
    for (j=1:N) {
      h[i][j] = foo(h[i-1][j],
                    h[i-1][j-1],
                    h[i][j-1]);
      advance;
    }
    advance;
  }
```

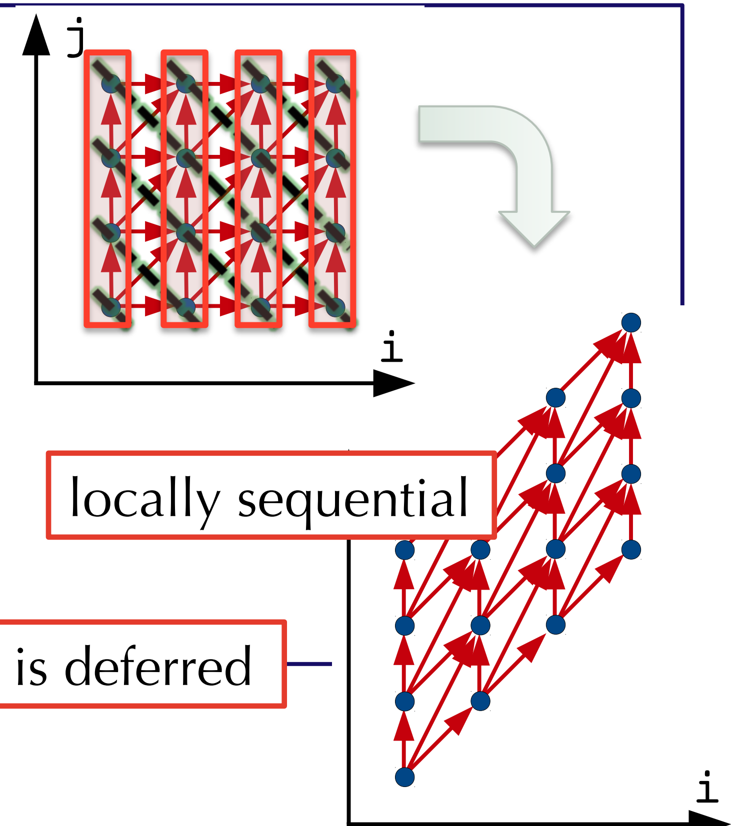


Example: Skewing

- Equivalent parallelism without changing loops

```
clocked finish
  for (i=1:N) {
    clocked async
    for (j=1:N) {
      h[i][j] = foo(h[i-1][j],
                    h[i-1][j-1],
                    h[i][j-1]);
      advance;
    }
    advance;
  }
```

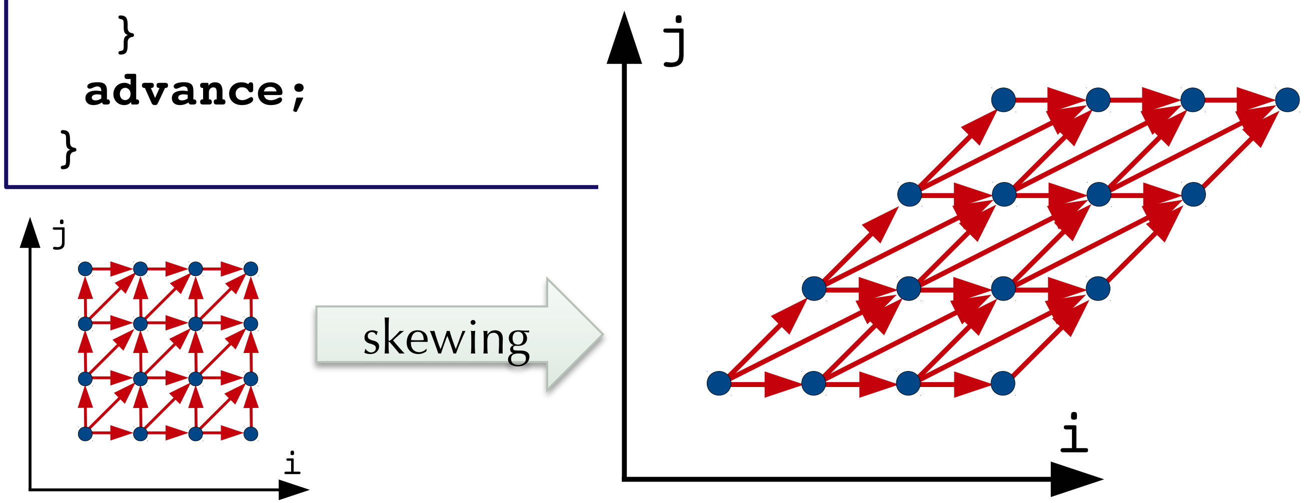
the launch of the entire block is deferred



Example: Skewing

- You can have the same skewing

```
clocked finish
 for (j=1:N) {
  clocked async
   for (i=1:N) {
    h[i][j] = foo(h[i-1][j], h[i-1][j-1], h[i][j-1]);
    advance;
   }
  advance;
 }
```

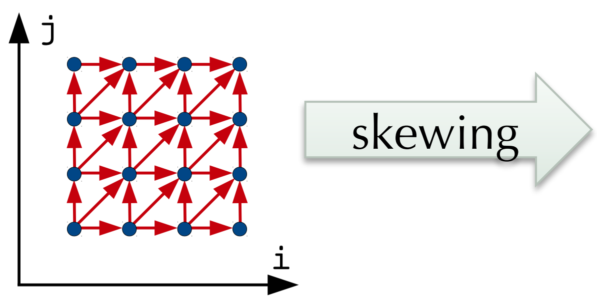


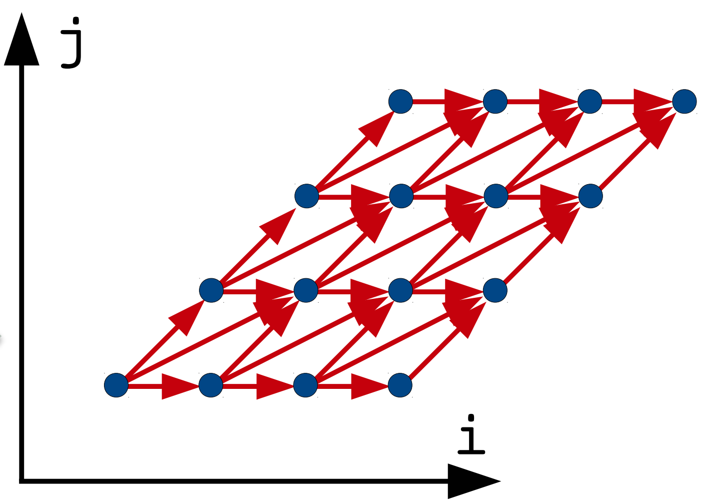
Example: Skewing

- You can have the same skewing

```
clocked finish
 for (j=1:N) {
  clocked async
   for (i=1:N) {
    h[i][j] = foo(h[i-1][j], h[i-1][j-1], h[i][j-1]);
    advance;
   }
  advance;
 }
```

note: interchange
outer parallel loop with clocks

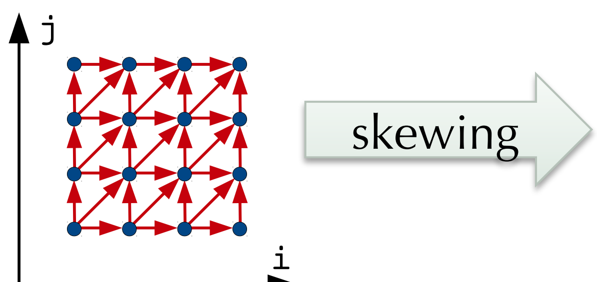


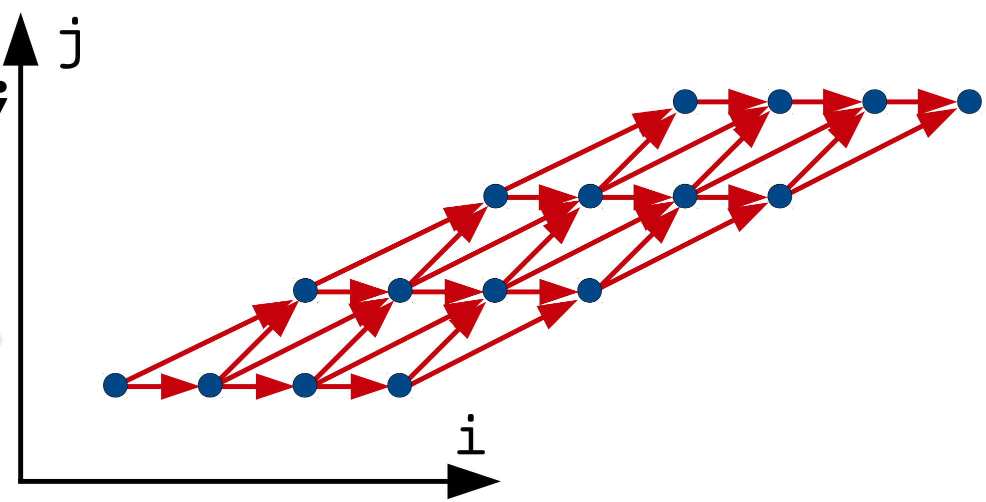


Example: Skewing

- You can have the same skewing

```
clocked finish
 for (j=1:N) {
  clocked async
   for (i=1:N) {
    h[i][j] = foo(h[i-1][j], h[i-1][j-1], h[i][j-1]);
    advance;
   }
  advance;advance;
 }
```





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;
    }
```

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

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

Discussion

- 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