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

expressive performance

alternative “usual” way

performance

“usual” way

X10 Workshop 2015
Context: Loop Transformations

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

\[
\text{for } i \\
\quad \text{for } j \\
X[i] += ... \\
\]

\[
\text{for } i \\
\quad \text{forall } j \\
X[i] += ... \\
\]

- But not always

\[
\text{for } i = 0 .. N \\
\quad \text{for } j = 1 .. M \\
X[j] = \text{foo}(X[j-1], X[j+1]); \\
\]

\[
\text{for } i = 1 .. 2N+M \\
\quad \text{forall } j = /* \text{complex bounds} */ \\
X[j] = \text{foo}(X[2*j-i-1], X[2*j-i+1]); \\
\]
Automatic Parallelization

- Very sensitive to inputs

```plaintext
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(1,(t1-1));
    lbp = 2;
    ubp = M-1;
    #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));
}

very difficult to understand ➔ trust it or not use it
```
Expressing with Clocks

- Goal: retain the original structure

```plaintext
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

```c
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

```java
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

```
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;
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

```x10
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][j-1] + y[i][j-1];
y[i+1][j+1];
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;
```

- Clocks are more dynamic

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

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

```
//P2
advance;
S1;
```
clocks vs barriers

- Barriers can easily deadlock

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

- Clocks are more dynamic

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

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

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

```plaintext
deadlock
```

```plaintext
OK
```

```plaintext
OK
```
Dynamicity of Clocks

- Implicit Syntax

```plaintext
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**

  ```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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

\[
\text{for } (i=1:2N-1) \\
\text{forall } (j=\max(1,i-N):\min(N,i-1)) \\
h[i][j] = \text{foo}(h[(i-j)-1][j], \\
h[(i-j)-1][j-1], \\
h[(i-j)][j-1])
\]
Example: Skewing

Skewing the loops is not easy

\[
\text{for } (i=1:2N-1) \\
\text{forall } (j=\max(1,i-N):\min(N,i-1)) \\
h[i][j] = \text{foo}(h[(i-j)-1][j], \\
h[(i-j)-1][j-1], \\
h[(i-j)][j-1])
\]

changes to loop bounds and indexing

skewing changes to loop bounds and indexing
Example: Skewing

- Equivalent parallelism without changing loops

```plaintext
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

```plaintext
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

```c
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

```plaintext
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

\[
\text{forall } (i=1:N) \\
S1; \\
S2;
\]

\[
\text{forall } (i=1:N) \\
S1; \\
\text{forall } (i=1:N) \\
S2;
\]

\[
\text{for } (i=1:N) \\
\text{async } \{ \\
S1; \\
S2; \\
\}
\]

\[
\text{for } (i=1:N) \\
\text{async } \{ \\
S1; \\
\text{advance}; \\
S2; \\
\}
\]
Example: Loop Fusion

- Removes all the parallelism

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

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

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

```c
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