Correctness Issues in Transforming Task Parallel Programs

V. Krishna Nandivada

IIT Madras

10-Jan-2013

Collaborators: Vivek Sarkar, Jun Shirako and Jisheng Zhao.

“I don’t like the idea of optimizations going wrong!”
Multi-core a new era

“Be the change you want to see in the world.” – Mahatma Gandhi

- New H/W: Opteron, (AMD), Cell (IBM+), Core i7 (Intel), Roadrunner, ...
- New Languages: CAF, Chappel, Fortress, UPC, X10, HJ
Multi-core a new era

“Be the change you want to see in the world.” – Mahatma Gandhi

- New H/W: Opteron, (AMD), Cell (IBM+), Core i7 (Intel), Roadrunner, ...
- New Languages: CAF, Chappel, Fortress, UPC, X10, HJ

New times ⇒ New challenges ⇒ New solutions.
Multi-core a new era

“Be the change you want to see in the world.” – Mahatma Gandhi

- New H/W: Opteron, (AMD), Cell (IBM+), Core i7 (Intel), Roadrunner, ...
- New Languages: CAF, Chappel, Fortress, UPC, X10, HJ

**New challenge:** applications/system software must be redesigned for multi-core parallelism.

- automatic (in the compiler) or semi-automatic (as a source-source refactoring)

**New challenge:** Optimizing task parallel programs.

- Reducing *communication* - activities, synchronization, data.
- Reasoning about correctness of program transformations.
- Reasoning about control and data dependence.

New times ⇒ New challenges ⇒ New solutions.
Relevant HJ syntax

- `async S`: creates an asynchronous activity.
**Relevant HJ syntax**

- `async S`: creates an asynchronous activity.
- `finish S`: ensures activity termination.

```plaintext
// Parent Activity
finish {
    S1; // Parent Activity
    async {
        S2; // Child Activity
    }
    S3; // Parent activity continues
}
S4;
```
Relevant HJ syntax

- `async S`: creates an asynchronous activity.
- `finish S`: ensures activity termination.

```plaintext
// Parent Activity
finish {
    S1; // Parent Activity
    async {
        S2; // Child Activity
    }
    S3; // Parent activity continues
}
S4;

foreach (i: [1..n]) ≡ for (i: [1..n])
    S async S
```
Relevant HJ syntax

- `async S`: creates an asynchronous activity.
- `finish S`: ensures activity termination.

```java
// Parent Activity
finish {
    S1; // Parent Activity
    async {
        S2; // Child Activity
    }
    S3; // Parent activity continues
}
S4;
foreach (i: [1..n]) ≡ for (i: [1..n])
    async S
forall (i: [1..n]) ≡ finish foreach (i: [1..n])
```
IEF and isolated

- Each activity has a unique parent finish – called the Immediately enclosing finish (IEF).
Each activity has a unique parent finish – called the Immediately enclosing finish (IEF).

Statically each async has one or more IEFs.
IEF and isolated

- Each activity has a unique parent finish – called the Immediately enclosing finish (IEF).
- Statically each async has one or more IEFs.
IEF and isolated

- Each activity has a unique parent finish – called the Immediately enclosing finish (IEF).
- Statically each async has one or more IEFs.

```c
void foo()
{
    async {
        S;
    }
}
```
IEF and isolated

- Each activity has a unique parent finish – called the Immediately enclosing finish (IEF).
- Statically each async has one or more IEFs.

```c
void foo()
{
    async {
        S;
    }
}

main()
{
    finish {
        ... foo(); ...
    }
    finish {
        ... foo(); ...
    }
    foo();
}
```
IEF and isolated

- Each activity has a unique parent finish – called the Immediately enclosing finish (IEF).
- Statically each async has one or more IEFs.

```c
void foo(){
    async {
        S;
    }
}

main(){
    finish {
        ... foo(); ... 
    }
    finish {
        ... foo(); ...
    }
    foo();
}
```

- isolated S: global critical section, provides weak isolation.
1. Background

2. Data Dependence in task parallel programs

3. Static Happens Before and Dependence relation

4. Optimization framework

5. Correctness

6. Example optimizations

7. Transformations in the presence of exceptions

8. Conclusion
Correctness of programs

Say a program $P$, is transformed to $P'$. 
Correctness of programs

Say a program $P$, is transformed to $P'$.

**Sequential programs**: If the *behaviour* of $P$ and $P'$ match.
Correctness of programs

Say a program $P$, is transformed to $P'$.

**Sequential programs**: If the *behaviour* of $P$ and $P'$ match.

**Parallel programs**: 
Correctness of programs

Say a program $P$, is transformed to $P'$. 

**Sequential programs**: If the *behaviour* of $P$ and $P'$ match.

**Parallel programs**: If the *behaviours* of $P'$ is a subset of the *behaviours* of $P$. 

V. Krishna Nandivada (IIT Madras)
Correctness of programs

Say a program $P$, is transformed to $P'$.

**Sequential programs**: If the *behaviour* of $P$ and $P'$ match.

**Parallel programs**: If the *behaviours* of $P'$ is a subset of the *behaviours* of $P$.

How to extend it to transformations of parallel programs?
Legality of program transformation requires the preservation of the order of “interfering” memory accesses.

Traditional analysis is not sufficient in the context of task parallel languages.

Constructs like `async` makes it challenging.

```c
for (int i = ...) {
    /*S1*/ X[f(i)] = ...
    async {
        /*S2*/ ... = X[g(i)];
    }
}
```
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs;
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; \( HB(I_A, I_B) = true \), if

\[
\begin{align*}
&(\text{Sequential order}) \quad S_1; \quad I_A \quad S_2; \quad I_B \\
&(\text{Async creation}) \quad \text{async} \quad // \quad I_A \text{S} \quad I_B \\
&(\text{Finish termination}) \quad \text{finish} \{ \quad \text{async} \{ \quad S_1; \quad S_2; \quad I_A \text{S} \quad I_B \} \} \quad \text{finish-end} \quad I_B \\
&(\text{Isolated}) \quad \text{Assume a total order.} \\
&(\text{Async isolation}) \quad S_0; \quad S_1; \quad I_A \quad S_2; \quad I_B \quad S_3; \\
&(\text{Transitivity}) \quad HB(I_A, I_C) = true \quad \text{and} \quad HB(I_C, I_B) = true
\end{align*}
\]
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to happens-before dependence in parallel programs; \( HB(I_A, I_B) = \text{true} \), if

- **(Sequential order)**
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; $HB(I_A, I_B) = true$, if

- **(Sequential order)**
  
  $S1; // I_A$
  $S2; // I_B$
  
  // $I_B$ is control or data dependent on $I_A$. 

---

V. Krishna Nandivada (IIT Madras)
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; \( HB(I_A, I_B) = true \), if

- **(Sequential order)**
  
  \[
  S_1; \quad // \quad I_A
  
  S_2; \quad // \quad I_B
  
  //I_B \text{ is control or data dependent on } I_A.
  
- **(Async creation)**
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; \( HB(I_A, I_B) = true \), if

- **(Sequential order)**
  
  \[
  S1; \quad // \quad I_A \\
  S2; \quad // \quad I_B \\
  \]

  \( I_B \) is control or data dependent on \( I_A \).

- **(Async creation)**

  \[
  \text{async} \quad // \quad I_A \\
  S \quad // \quad I_B \\
  \]
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to \textit{happens-before dependence} in parallel programs; \( HB(I_A, I_B) = true \), if

- (Sequential order)
  \[
  S_1; // I_A \\
  S_2; // I_B \\
  // I_B \text{ is control or} \\
  // \text{data dependent on } I_A.
  \]

- (Async creation)
  \[
  \text{async} // I_A \\
  \text{S} // I_B
  \]

- (Finish termination)
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; \( HB(I_A, I_B) = true \), if

- **(Sequential order)**
  
  \[
  S_1; \quad // \quad I_A \\
  S_2; \quad // \quad I_B \\
  //I_B \text{ is control or} \\
  //\text{data dependent on } I_A.
  \]

- **(Async creation)**
  
  \[
  \text{async} \quad // \quad I_A \\
  S \quad // \quad I_B
  \]

- **(Finish termination)**
  
  \[
  \text{finish} \{ \quad // \quad \text{finish-start} \\
  \quad \text{async} \{ \\
  \quad S_1; \\
  \quad S_2; \quad // \quad I_A \\
  \quad \} \\
  \quad \} \quad // \quad \text{finish-end} \quad I_B
  \]
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to \textit{happens-before dependence} in parallel programs; $HB(I_A, I_B) = \text{true}$, if

- (Sequential order)
  
  \begin{verbatim}
  S1; // I_A
  S2; // I_B
  \end{verbatim}
  
  $I_B$ is control or data dependent on $I_A$.

- (Async creation)
  
  \begin{verbatim}
  async // I_A
  S // I_B
  \end{verbatim}

- (Finish termination)
  
  \begin{verbatim}
  finish { // finish-start
    async {
      S1;
      S2; // I_A
    }
  } // finish-end I_B
  \end{verbatim}
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; \( HB(I_A, I_B) = true \), if

1. **(Sequential order)**
   
   \[
   S_1; // I_A \\
   S_2; // I_B \\
   \]
   
   // \( I_B \) is control or data dependent on \( I_A \).

2. **(Async creation)**

   
   \[
   \text{async} // I_A \\
   S // I_B \\
   \]

3. **(Finish termination)**

   
   \[
   \text{finish} \{ // finish-start \\
   \text{async} \{ \\
   S_1; \\
   S_2; // I_A \\
   \} \\
   \} // finish-end // I_B \\
   \]

   // Assume a total order.
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; $HB(I_A, I_B) = true$, if

- **(Sequential order)**
  
  \[
  \begin{align*}
  &S1; \quad // \quad I_A \\
  &S2; \quad // \quad I_B \\
  // &I_B \text{ is control or} \\
  // &\text{data dependent on} \ I_A.
  \end{align*}
  \]

- **(Async creation)**

  \[
  \begin{align*}
  &\text{async} \quad // \quad I_A \\
  &S \quad // \quad I_B
  \end{align*}
  \]

- **(Finish termination)**

  \[
  \begin{align*}
  &\text{finish} \ {\{ \quad // \quad \text{finish-start} \\
  &\quad \text{async} \ {\{ \quad \}
  \\
  &\quad S1; \\
  &\quad S2; \quad // \quad I_A \\
  &\quad \}
  \\
  &\quad \} \quad // \quad \text{finish-end} \quad I_B
  \end{align*}
  \]

- **(Isolated)** Assume a total order.

  \[
  \begin{align*}
  &\text{isolated} \ {\{ \quad \}
  \\
  &\quad S0; \\
  &\quad S1; \quad // \quad I_A \\
  &\quad \}
  \\
  &\quad \text{isolated} \ {\{ \quad \}
  \\
  &\quad S2; \quad // \quad I_B \\
  &\quad S3;
  \end{align*}
  \]

\[\text{Transitivity:} \]

\[
HB(I_A, I_C) = true \quad \text{and} \quad HB(I_C, I_B) = true
\]
Dynamic Happens-before dependence

Extending the classical definition of data dependence in sequential programs to *happens-before dependence* in parallel programs; $HB(I_A, I_B) = true$, if

- **(Sequential order)**
  
  $S1; \ // I_A$
  $S2; \ // I_B$
  
  // $I_B$ is control or data dependent on $I_A$.

- **(Async creation)**
  
  async  \ // $I_A$
  
  S        \ // $I_B$

- **(Finish termination)**
  
  finish { \ // finish-start
    async {
      S1;     \ // $I_A$
      S2;     \ // $I_A$
    }
  } \ // finish-end $I_B$

- **(Isolated)** Assume a total order.
  
  // A total order
  isolated {
      S0;
      S1;     \ // $I_A$
  }

  isolated {
    S2;     \ // $I_B$
    S3;
  }

- **(Transitivity)** $HB(I_A, I_C) = true$ and $HB(I_C, I_B) = true$
Happens-before dependence using dynamic HB

Given dynamic $HB$, and a two statement $A$ and $B$ in a program, we say that $\text{HBD}(A, B) = true$, if

- $\exists I_A, I_B$, instances of $A$ and $B$, such that

$\exists I_A, I_B$, instances of $A$ and $B$, such that
Given dynamic $HB$, and a two statement $A$ and $B$ in a program, we say that $HBD(A, B) = true$, if

1. $\exists I_A, I_B$, instances of $A$ and $B$, such that
2. $HB(I_A, I_B) = true$, and
3. $\neg \exists I_C$ in the same execution that writes $X$ such that $HB(I_A, I_C) = true$ and $HB(I_C, I_B) = true$. 

V. Krishna Nandivada (IIT Madras)
Happens-before dependence using dynamic HB

Given dynamic $HB$, and a two statement $A$ and $B$ in a program, we say that $\text{HBD}(A, B) = true$, if

- $\exists I_A, I_B$, instances of $A$ and $B$, such that
  1. $HB(I_A, I_B) = true$, and
  2. $I_A$ and $I_B$ access the same location $X$ and at least one of the accesses is a write, and
Happens-before dependence using dynamic HB

Given dynamic $HB$, and a two statement $A$ and $B$ in a program, we say that $HBD(A, B) = true$, if

- $\exists I_A, I_B$, instances of $A$ and $B$, such that
  1. $HB(I_A, I_B) = true$, and
  2. $I_A$ and $I_B$ access the same location $X$ and at least one of the accesses is a write, and
  3. $¬\exists I_C$ in the same execution that writes $X$ such that $HB(I_A, I_C) = true$ and $HB(I_C, I_B) = true$. 
HBD details

- If no parallelism $\rightarrow \text{HBD} =$ traditional data dependence.
- \text{HBD} is conservative.
- We classify dependence as \textit{flow}, \textit{anti}, and \textit{output} dependence.
HBD analysis example

```java
for (int i = ...) {
    /*S1*/ X[f(i)] = ...
    async {
        /*S2*/ ... = X[g(i)];
    }
}
```

Sequential compiler, sequential program – exists a loop carried dependence cycle.

In the parallel version – no dependence from \( S_2 \) to \( S_1 \); hence no cycle – loop can be distributed.

```java
for (int i = ...) {
    /*S1*/ X[f(i)] = ...
    for (int i = ...) {
        /*S2*/ ... = X[g(i)];
    }
}
```

\[ \Rightarrow \] // After loop dist
HBD analysis example

```c
for (int i = ...) {
    /*S1*/ X[f(i)] = ...
    async {
        /*S2*/ ... = X[g(i)]; 
    }
}
```

- Sequential compiler, sequential program – exists a loop carried dependence cycle.
- In the parallel version – no dependence from $S_2$ to $S_1$; hence no cycle – loop can be distributed.
HBD analysis example

for (int i = ...) {
    /*S1*/ X[f(i)] = ...
    async {
        /*S2*/ ... = X[g(i)];
    }
}

- Sequential compiler, sequential program – exists a loop carried dependence cycle.
- In the parallel version – no dependence from $S_2$ to $S_1$; hence no cycle – loop can be distributed.

for (int i = ...) {
    /*S1*/ X[f(i)] = ...
    async {
        /*S2*/ ... = X[g(i)];
    }
}

// After loop dist
for (int i = ...) {
    /*S1*/ X[f(i)] = ...
    async {
        /*S2*/ ... = X[g(i)];
    }
}
Outline

1. Background
2. Data Dependence in task parallel programs
3. Static Happens Before and Dependence relation
4. Optimization framework
5. Correctness
6. Example optimizations
7. Transformations in the presence of exceptions
8. Conclusion
Compute Static Happens-before relation.

- Use Program Structure Graph (PSG) as the program representation.
Compute Static Happens-before relation.

- Use Program Structure Graph (PSG) as the program representation.
  - nodes = root, statement, loop, async, finish, isolated and call.
Compute Static Happens-before relation.

- Use Program Structure Graph (PSG) as the program representation.
  - nodes = root, statement, loop, async, finish, isolated and call.
  - edges = subset of abstract syntax tree.
Compute Static Happens-before relation.

- Use Program Structure Graph (PSG) as the program representation.
  - nodes = root, statement, loop, async, finish, isolated and call.
  - edges = subset of abstract syntax tree.
- In two phases.
Compute Static Happens-before relation.

- Use Program Structure Graph (PSG) as the program representation.
  - nodes = root, statement, loop, async, finish, isolated and call.
  - edges = subset of abstract syntax tree.

- In two phases.
  - Generate and solve a set of constraints to compute static happens-before information, without considering isolated statements.
Compute Static Happens-before relation.

- Use Program Structure Graph (PSG) as the program representation.
  - nodes = root, statement, loop, async, finish, isolated and call.
  - edges = subset of abstract syntax tree.
- In two phases.
  - Generate and solve a set of constraints to compute static happens-before information, without considering isolated statements.
  - Improve the partial may-happen-before information by considering isolated statements.
Phase 1
For each $N_1, N_2 \in Nodes$
Phase 1
For each $N_1, N_2 \in \text{Nodes}$

- Same activity:
Phase 1
For each $N_1, N_2 \in Nodes$

1. Same activity: $(N_1, N_2) \in MHB$
Phase 1
For each $N_1, N_2 \in Nodes$

1. Same activity: $(N_1, N_2) \in MHB$

2. Loop ancestor: $N_1$ and $N_2$
Phase 1
For each $N_1, N_2 \in Nodes$

1. Same activity: $(N_1, N_2) \in MHB$

2. Loop ancestor:
Phase 1
For each \( N_1, N_2 \in Nodes \)

1. Same activity: \((N_1, N_2) \in MHB\)

2. Loop ancestor: \(\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB\)
Phase 1
For each $N_1, N_2 \in Nodes$

1. Same activity: $\{(N_1, N_2) \in MHB\}$

2. Loop ancestor: $\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB$

3. Async and stmt: $\{(N_1, N_2) \in MHB\}$
Phase 1
For each $N_1, N_2 \in Nodes$

1. Same activity: $(N_1, N_2) \in MHB$

2. Loop ancestor: $\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB$

3. Async and stmt:
Static MHB

Phase 1
For each \(N_1, N_2 \in \text{Nodes}\)

1. Same activity: \((N_1, N_2) \in \text{MHB}\)

   \[
   \begin{align*}
   \text{async} & \quad N_1 \quad \cdots \quad N_2 \\
   \text{async} & \quad \text{loop} \\
   \text{async} & \quad (N_1, N_2) \\
   \end{align*}
   \]

2. Loop ancestor: \(\{(N_1, N_2), (N_2, N_1)\} \subseteq \text{MHB}\)

   \[
   \begin{align*}
   \text{async} & \quad N_1 \quad N_2 \\
   \text{loop} & \\
   \text{async} & \quad (N_1, N_2) \\
   \end{align*}
   \]

3. Async and stmt: \((N_1, N_2) \in \text{MHB}\)

   \[
   \begin{align*}
   \text{async} & \quad N_2 \quad \cdots \\
   \text{async} & \quad (N_1, N_2) \\
   \end{align*}
   \]
Async and IEF

\[ \text{finish} \]

\[ \text{finStart} \]

\[ \text{finEnd} (N_2) \]

\[ \ldots \]

\[ \text{async} \]

\[ \text{finish} \]

\[ \text{finStart} \]

\[ \text{finEnd} (N_2) \]

\[ \ldots \]

\[ N_1 \]

\[ \ldots \]

\[ (N_1, N_2) \in MHB; \]

Transitivity: if \( \exists N_3 \in \text{Nodes}, (N_1, N_3) \in MHB \) and \( (N_3, N_2) \in MHB \) then \( (N_1, N_2) \in MHB \).
Async and IEF

Static MHB (contd)

V. Krishna Nandivada (IIT Madras)

10-Jan-2013 15 / 32
Async and IEF

\[(N_1, N_2) \in MHB;\]

Tansitivity: if \( \exists N_3 \in Nodes, (N_1, N_3) \in MHB \) and \( (N_3, N_2) \in MHB \) then \( (N_1, N_2) \in MHB. \)
For any two nodes $N_1$ and $N_2$, we say that $N_2$ has a may-happen-before-dependence on $N_1$, denoted by $\text{MHBD}(N_1, N_2) = true$, if

1. $(i, (N_1, N_2)) \in \text{MHB}$,
2. $N_1$ and $N_2$ access the same variable or storage location and one of the access is a write,
3. $\neg \exists N_3 \in \text{Nodes}: \text{MHBD}(N_3, N_1) = true$ and $\text{MHBD}(N_2, N_3) = true$. 
Static Happens-before dependence

For any two nodes $N_1$ and $N_2$, we say that $N_2$ has a may-happen-before-dependence on $N_1$, denoted by $\text{MHBD}(N_1, N_2) = true$, if

i. $(N_1, N_2) \in \text{MHB}$,
Static Happens-before dependence

For any two nodes $N_1$ and $N_2$, we say that $N_2$ has a may-happen-before-dependence on $N_1$, denoted by $\text{MHBD}(N_1, N_2) = true$, if

i. $(N_1, N_2) \in MHB$,

ii. $N_1$ and $N_2$ access the same variable or storage location and one of the access is a write,
For any two nodes $N_1$ and $N_2$, we say that $N_2$ has a may-happen-before-dependence on $N_1$, denoted by $\text{MHBD}(N_1, N_2) = true$, if

i. $(N_1, N_2) \in MHB$,

ii. $N_1$ and $N_2$ access the same variable or storage location and one of the access is a write,

iii. $\neg \exists N_3 \in \text{Nodes}: \text{MHBD}(N_3, N_1) = true$ and $\text{MHBD}(N_2, N_3) = true$. 
Correctness of a transformation

A transformation of a parallel program is semantics-preserving if the set of happens-before dependencies of all the variables at all program points in the source program are conservatively preserved in the translated program.
Outline

1. Background
2. Data Dependence in task parallel programs
3. Static Happens Before and Dependence relation
4. Optimization framework
   - Extending traditional loop transformations
   - New transformations
5. Correctness
6. Example optimizations
7. Transformations in the presence of exceptions
8. Conclusion
Extending traditional loop transformations

1. Serial loop distribution:
   
   ```
   for (...) { S1; S2; } 
   // no dependence cycle between S1 & S2
   =>
   { for (...) { S1; } 
   for (...) { S2; } }
   ```

2. Parallel loop distribution:
   
   ```
   forall (point p : R1) 
   { S1; S2; } 
   // S1 has no dependence on S2
   =>
   { forall (point p : R1) S1; 
   forall (point p : R1) S2; }
   ```
1. Serial loop distribution:
   for (...) { S1; S2; }
   // no dependence cycle between S1 & S2
   \[\Rightarrow\] { for (...) {S1;} }

2. Parallel loop distribution:
   forall (point p : R1)
   \{ S1; S2; \}
   // S1 has no dependence on S2
   \[\Rightarrow\] { forall (point p : R1) S1; }
   { forall (point p : R1) S2; }

3. Loop/Finish interchange:
   for (S1;cond;S2)
   finish S3;
   // Say \(E_s\) = set of e-asyns in S3
   // \(\neg\exists e \in E_s: cond\) has dependence on e
   // \(\neg\exists e \in E_s:body\ of\ e\ has\ loop\)
   // carried dependence on S2, cond or S3
   \[\Rightarrow\] { S1;
   finish
   for (;cond;S2)
   S3; }

4. Serial-parallel loop interchange:
   for (i: [1..n])forall (point p : R1) S;
   // iterations of the for loop are independent.
   \[\Rightarrow\] { forall (point p : R1)for (i: [1..n])S; }

5. Parallel-serial loop interchange:
   forall (point p : R1)for (point q : R2) S
   // R2 is independent of p
   // S contains no break/continue
   \[\Rightarrow\] { for (point q : R2)forall (point p : R1)S }

V. Krishna Nandivada (IIT Madras)
10-Jan-2013 19 / 32
1. Serial loop distribution:
   
   ```
   for (...) { S1; S2; }
   // no dependence cycle between S1 & S2
   =>
   for (...) { S1; }
   for (...) { S2; }
   ```

2. Parallel loop distribution:
   
   ```
   forall (point p : R1)
   { S1; S2; }
   // S1 has no dependence on S2
   =>
   forall (point p : R1) S1;
   forall (point p : R1) S2;
   ```

3. Loop/Finish interchange:
   
   ```
   for (S1; cond; S2)
   finish S3;
   // Say Es = set of e-asyns in S3
   // \( \neg \exists e \in E_s: \text{cond has dependence on e} \)
   // \( \neg \exists e \in E_s: \text{body of e has loop} \)
   // carried dependence on S2, cond or S3
   =>
   S1;
   finish
   for (;cond;S2)
   S3;
   ```

4. Serial-parallel loop interchange:
   
   ```
   for (i: [1..n])
   forall (point p : R1) S;
   // iterations of the for loop are independent.
   // R1 does not depend on i
   =>
   forall (point p : R1)
   for (i: [1..n])
   S;
   ```
### Extending traditional loop transformations I

**1. Serial loop distribution:**
```
for (...) { S1;S2; }
// no dependence cycle between S1 & S2
```

- $\Rightarrow$
  - `{ for (...) {S1;} }
  - `{ for (...) {S2;} }

**2. Parallel loop distribution:**
```
forall (point p : R1) 
{ S1; S2; }
// S1 has no dependence on S2
```

- $\Rightarrow$
  - `{ forall (point p : R1) S1; }
  - `{ forall (point p : R1) S2; }

**3. Loop/Finish interchange:**
```
for (S1;cond;S2) finish S3;
// Say $E_s$ = set of e-asynchs in S3
// $\neg\exists e \in E_s$: cond has dependence on e
// $\neg\exists e \in E_s$: body of e has loop
// carried dependence on S2, cond or S3
```

- $\Rightarrow$
  - `{ S1; 
    finish
    `{ for (;cond;S2)
  }
  }
  `{ S3; }

**4. Serial-parallel loop interchange:**
```
for (i: [1..n])forall (point p : R1) S;
// iterations of the for loop are independent.
// R1 does not depend on i
```

- $\Rightarrow$
  - `{ forall (point p : R1) 
    `{ for (i: [1..n]) S;
  }
  }

**5. Parallel-serial loop interchange:**
```
forall (point p : R1)forall (point q : R2) S
// R2 is independent of p
// S contains no break/continue
```

- $\Rightarrow$
  - `{ forall (point q : R2) 
    `{ for (point q : R2) S;
  }
  }
  `{ forall (point p : R1) S }

### 6. Loop unpeeling:

$$\forall (\text{point } p: \mathbb{R}) S_1; \\
S_2; \quad \Rightarrow \{ \forall (\text{point } p: \mathbb{R}) S_1; S_2; \}$$

- no break/continue in \(S_2\).
- \(E_s = \text{set of } e\text{-asyncs in } S_1\)
- \(\neg \exists e \in E_s: S_2 \text{ has dependence on } e\)

### 7. Loop fusion:

$$\forall (\text{point } p: \mathbb{R}_1) S_1; \\
\forall (\text{point } p: \mathbb{R}_2) S_2; \quad \Rightarrow \{ \forall (\text{point } p: \mathbb{R}_1 \mid \mathbb{R}_2) \}
\begin{cases} 
\text{if (R1.contains (p)) } S_1; \\
\text{if (R2.contains (p)) } S_2; \\
\end{cases}$$

- \(E_s = \text{set of } e\text{-asyncs in } S_1\)
- \(\neg \exists e \in E_s: S_2 \text{ has dependence on } e\)

### 8. Loop switching:

$$\text{if (c)}$$

$$\Rightarrow \{ \text{final boolean } v = c; \forall (\text{point } p: \mathbb{R}) \}
\begin{cases} 
\text{if (v) } S; \\
\end{cases}$$

### 9. Parallel loop unswitching:

$$\forall (\text{point } p : \mathbb{R}_1)$$

$$\text{if (e) } S \quad \Rightarrow \{ \text{if (e)} 
\begin{cases} 
\forall (\text{point } p : \mathbb{R}_1) S \\
\end{cases} \}$$

- \(e\) is a pure function and is independent of \(p\)

### 10. Serial loop unswitching:

$$\text{for}(S_2; \text{cond1}; S_3)\{ \\
\text{if (cond2) } S_4; \text{ else } S_5; \\
\} \quad \Rightarrow \{ \text{if (cond2) } \}
\begin{cases} 
\text{for}(S_2; \text{cond1}; S_3) S_4; \\
\} \text{ else } \{ \\
\text{for}(S_2; \text{cond1}; S_3) S_5; \\
\}$$

- \(\text{cond2 has no dependence}\)
- \(\text{on } S_2, S_3, S_4 \text{ and } S_5,\)
- \(\text{cond2 has no side effects}\)
Variations of traditional transformations

1. **Finish distribution:**

   \[
   \text{finish}\{ \text{S1; S2;} \} \quad \Rightarrow \quad \begin{cases} 
   \text{S1;} \\
   \text{finish}\{ \text{S2;} \}
   \end{cases}
   \]

   \text{S1 has no e-asyncs.}
**Variations of traditional transformations**

<table>
<thead>
<tr>
<th>1. Finish distribution:</th>
<th>[ \text{finish}{ \text{S1; S2; } } ]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ \text{// S1 has no e-asyncs.} ] =&gt; [ { \text{S1; finish}{ \text{S2; } } } ]</td>
</tr>
</tbody>
</table>
| 2. Finish unswitching: | \[ \text{finish} \]
|                        | \[ \text{if}\text{(cond)}\text{S1; else S2;} \] |
|                        | \[ \text{// cond has no e-async} \] => \[ \{ \text{if (cond) finish S1; else finish S2; } \} \] |
### Variations of traditional transformations

1. **Finish distribution:**
   
   ```
   finish { S1; S2; }  
   // S1 **has no** e-asyncs.  
   $$\Rightarrow$$  
   \begin{cases}  
   S1;  
   \end{cases}  
   \begin{cases}  
   \text{finish} \{ S2; \}  
   \end{cases}
   ```

2. **Finish unswitching:**
   
   ```
   finish  
   if (cond) S1; else S2;  
   // cond **has no** e-async  
   $$\Rightarrow$$  
   \begin{cases}  
   \text{if (cond) finish S1;}  
   \text{else finish S2;}  
   \end{cases}
   ```

3. **If expansion:**
   
   ```
   finish {  
   S1;  
   if (cond) S2; else S3;  
   S4; }  
   // **no dependence between** cond **and** S1  
   $$\Rightarrow$$  
   \begin{cases}  
   \text{finish} \{  
   \text{if (cond)}  
   \{ S1; S2; S4; \}  
   \text{else}  
   \{ S1; S3; S4 \}  
   \}  
   \end{cases}
   ```
### Variations of traditional transformations

<table>
<thead>
<tr>
<th>Variation</th>
<th>Code</th>
<th>Transform to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Finish distribution:</td>
<td><code>finish { S1; S2; }</code> // <code>S1</code> has no e-asyncs.</td>
<td><code>{ S1; finish { S2; }</code></td>
</tr>
</tbody>
</table>
| 2. Finish unswitching: | `finish`  
if(cond) `S1;` else `S2;` // `cond` has no e-async | `{ if (cond) finish `S1;` else finish `S2;` }                               |
| 3. If expansion: | `finish {`  
`S1;`  
if(cond) `S2;` else `S3;`  
`S4; }` // no dependence between `cond` and `S1` | `{ finish {
if (cond)
`S1; S2; S4;`
else
`S1; S3; S4`} }                                                      |
| 4. Redundant finish elimination: | `finish `S;` // `S` has no e-async. | `{ `S;` }                                                               |

---

V. Krishna Nandivada (IIT Madras)  
10-Jan-2013 21 / 32
## Variations of traditional transformations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| **1. Finish distribution:** | finish \{ S1; S2; \}  
// S1 *has no* e-asyncs.  
\[\Rightarrow\] \{ S1; finish \{ S2; \} \} |
| **2. Finish unswitching:** | finish  
if (cond) S1; else S2;  
// cond *has no* e-async  
\[\Rightarrow\] \{ if (cond) finish S1; else finish S2; \} |
| **3. If expansion:** | finish \{  
S1;  
if (cond) S2; else S3;  
S4; \}  
// *no dependence between* cond *and* S1  
\[\Rightarrow\] \{ finish \{  
if (cond) \{ S1; S2; S4; \}  
else \{ S1; S3; S4 \} \} \} |
| **4. Redundant finish elimination:** | finish S;  
// S *has no* e-async.  
\[\Rightarrow\] \{ S; \} |
| **5. Tail finish elimination:** | finish \{ S1; finish S2; \}  
\[\Rightarrow\] \{ finish \{ S1; S2; \} \} |
## Variations of traditional transformations

1. **Finish distribution:**
   
   ```
   finish \{ \text{S1; S2;} \} \\
   // \text{S1 has no e-asynccs.} 
   \Rightarrow \{ \text{S1;} \} 
   
   finish \{ \text{S2;} \} 
   ```

2. **Finish unswitching:**
   
   ```
   finish 
   \text{if (cond) S1; else S2;} \\
   // \text{cond has no e-async} 
   \Rightarrow \{ \text{if (cond) finish S1;} \} 
   
   \text{else finish S2;} 
   ```

3. **If expansion:**
   
   ```
   finish \{ \text{S1;} \\
   \text{if (cond) S2; else S3;} \\
   \text{S4;} \} \\
   // \text{no dependence between cond and S1} \n   \Rightarrow \{ \text{if (cond)} \} 
   
   \{ \text{S1; S2; S4;} \} \\
   \text{else} 
   \{ \text{S1; S3; S4} \} 
   ```

4. **Redundant finish elimination:**
   
   ```
   finish \text{S;} \\
   // \text{S has no e-async.} 
   \Rightarrow \{ \text{S;} \} 
   ```

5. **Tail finish elimination:**
   
   ```
   finish \{ \text{S1; finish S2;} \} 
   \Rightarrow \{ \text{finish \{S1; S2;} \} 
   ```

6. **Finish fusion**
   
   ```
   finish \text{S1;} \\
   finish \text{S2;} \\
   // \text{Say } E_s = \text{set of e-asynccs in S1} \\
   // \text{\neg } \exists e \in E_s: \text{S2 has dependence on e} 
   \Rightarrow \{ \text{finish\{S1;} \} 
   
   \text{S2;} 
   ```
Outline

1. Background
2. Data Dependence in task parallel programs
3. Static Happens Before and Dependence relation
4. Optimization framework
5. Correctness
6. Example optimizations
7. Transformations in the presence of exceptions
8. Conclusion
Correctness

Definition

A transformation of a parallel program is semantics-preserving if the set of happens-before dependencies of all the variables at all program points in the source program are conservatively preserved in the translated program.
Correctness

Definition
A transformation of a parallel program is semantics-preserving if the set of happens-before dependencies of all the variables at all program points in the source program are conservatively preserved in the translated program.

Lemma
The preconditions for each rule ensure that the individual transformation resulting from each of the rules is semantics-preserving.
Correctness

Definition
A transformation of a parallel program is semantics-preserving if the set of happens-before dependencies of all the variables at all program points in the source program are conservatively preserved in the translated program.

Lemma
The preconditions for each rule ensure that the individual transformation resulting from each of the rules is semantics-preserving.

Theorem
Any optimization pass consisting of applying one or more instances of the rules shown is semantics-preserving.
Outline

1. Background
2. Data Dependence in task parallel programs
3. Static Happens Before and Dependence relation
4. Optimization framework
5. Correctness
6. Example optimizations
7. Transformations in the presence of exceptions
8. Conclusion
void foo(int n) {
    ...
    finish {
        for (...) {
            if (c) {
                async foo(n-1);
            } else {
                foo(n-1);
            }
        } // for
    } // finish
}

void foo(int n) {
    ... 
    finish {
        for (...) {
            if (c) {
                async foo(n-1);
            } else {
                foo(n-1);
            }
        } // for
    } // finish
} // finish

void foo(int n) {
    ... 
    if (c) {
        finish {
            for (...) {
                async foo(n-1);
            } // for
        } // finish
    } else {
        for (...) {
            foo(n-1);
        } // for
    }
}
Motivating example - finish elimination

```java
void sim_village_par_par(Village vil){
  // Traverse village hierarchy
  finish {
    final Iterator it = vil.forward.iterator();
    while (it.hasNext()){
      final Village v=(Village)it.next();
      if ((sim_level-vil.level) < cutoff){
        async sim_village_par_par(v);
      } else {
        sim_village_par_par(v);
      }
    } // while
  } // finish
} // end function

BOTS Health benchmark
```

V. Krishna Nandivada (IIT Madras)
Optimizing the “running” example

// Input program.
void sim_village_par(final Village vil) {
  finish {
    final Iterator it = vil.iterator();
    while (it.hasNext()) {
      final Village v = (Village) it.next();
      async seq ((sim_level - vil.level) >= bots_cutoff_value)
      sim_village_par(v);
    } // while
  } // finish:
  ....... }

(a)

// After if expansion
void sim_village_par(final Village vil) {
  finish {
    final Iterator it = vil.iterator();
    while (it.hasNext()) {
      if ((sim_level - vil.level) < bots_cutoff_value)
        final Village v = (Village) it.next();
      async sim_village_par(v);
      else {
        final Village v = (Village) it.next();
        sim_village_par(v);
      } } // while
  } // finish/* finish*/......

(b)
Optimizing the “running” example

Next: Loop unswitching
Optimizing the “running” example

```java
// After Loop Unswitching
void sim_village_par(final Village vil) {
    finish {
    1:finish {
    2: final Iterator it=vil.iterator();
    3: if (((sim_level - vil.level)
          < bots_cutoff_value){
    4:    while (it.hasNext()) {
    5:        final Village v=(Village)it.next();
    6:        async sim_village_par(v);} //while
    7:    } else {
    8:    while (it.hasNext()) {
    9:        final Village v=(Village)it.next();
   10:       sim_village_par(v);} } } /*finish*/ ... ...;
}
```

```java
// After if expansion.
void sim_village_par(final Village vil) {
    finish {
    1:finish {
    2: if((sim_level-vil.level)
        <bots_cutoff_value){
    3:    final Iterator it=vil.iterator();
    4:    while (it.hasNext()) {
    5:        final Village v=(Village)it.next();
    6:        async sim_village_par(v);} // while
    7:    ... ...;}
    8: }else {
    9:    final Iterator it=vil.iterator();
   10:    while (it.hasNext()) {
   11:       final Village v=(Village)it.next();
   12:       sim_village_par(v);}
   13:    ... ...; } /*finish*/ ... ...;
```
```
Optimizing the “running” example

Next: finish unswitching

```java
// After Loop Unswitching
void sim_village_par(final Village vil) {
    finish {
        final Iterator it=vil.iterator();
        if ((sim_level - vil.level) < bots_cutoff_value) {
            while (it.hasNext()) {
                final Village v=(Village)it.next();
                async sim_village_par(v); //while
            } else {
                while (it.hasNext()) {
                    final Village v=(Village)it.next();
                    sim_village_par(v); }
                ...
                } /*finish*/ ...
            }
}
```

```java
// After if expansion.
void sim_village_par(final Village vil) {
    finish {
        if ((sim_level - vil.level) < bots_cutoff_value) {
            final Iterator it=vil.iterator();
            while (it.hasNext()) {
                final Village v=(Village)it.next();
                async sim_village_par(v); //while
            } ...
        }
```

(c)
Optimizing the “running” example

```java
// After finish unswitching
void sim_village_par(final Village vil) {
  1: if ((sim_level - vil.level) < bots_cutoff_value) {
  2:     finish {
  3:         final Iterator it=vil.iterator();
  4:         while (it.hasNext()) {
  5:             final Village v=(Village)it.next();
  6:             async sim_village_par(v); // while
  7:             ... ...; } // finish
  8: } else {
  9:     finish {
 10:         final Iterator it=vil.iterator();
 11:         while (it.hasNext()) {
 12:             final Village v=(Village)it.next();
 13:             sim_village_par(v); // while
 14:             ... ...; } // finish
 15: } ... ...; }
}

// After redundant finish elimination
void sim_village_par(final Village vil) {
  1: if ((sim_level - vil.level) < bots_cutoff_value) {
  2:     finish {
  3:         final Iterator it=vil.iterator();
  4:         while (it.hasNext()) {
  5:             final Village v=(Village)it.next();
  6:             async sim_village_par(v); // while
  7:             ... ...; } // finish
  8: } else {
  9:     finish {
 10:         final Iterator it=vil.iterator();
 11:         while (it.hasNext()) {
 12:             final Village v=(Village)it.next();
 13:             sim_village_par(v); // while
 14:             ... ...;
 15:         } ... ...; }
```

(e) (f)
Transformations in the presence of exceptions

Finish distribution:

\[
\begin{align*}
\text{finish}\{ \ S1; \ S2; \ \} \quad &\rightarrow\quad \left\{ \ S1; \right. \\
// \ S1 \ has \ no \ e\text{-asyncs.} \\
\left. \ \text{finish}\{ \ S2; \ \} \right. \\
\end{align*}
\]
### Transformations in the presence of exceptions

<table>
<thead>
<tr>
<th>Finish distribution:</th>
<th>finish { S1; S2; } ⇒ (no exceptions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{S1 has no e-asynCs.})</td>
<td>({ \text{S1;} ) (\text{finish}{ \text{S2;} } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Finish distribution:</th>
<th>(with exceptions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish { S1; S2; }</td>
<td>try { S1; }</td>
</tr>
<tr>
<td>(\text{(1) S1 has no e-asynCs.})</td>
<td>catch (Exception e)</td>
</tr>
<tr>
<td>(\text{(a) S2 has e-asynCs.})</td>
<td>{ MultiException me tt=\text{new} \ldots; me.pushEx(e1); throw me; }</td>
</tr>
<tr>
<td></td>
<td>(\text{finish}{ \text{S2;} } )</td>
</tr>
</tbody>
</table>
Control and Data dependence in the context of task parallel programs.

Correctness argument in the presence of multiple tasks, procedures and Exceptions.

Extend traditional optimizations in the context of task parallel programs.

Results in significant performance improvement:

- geometric average performance improvement of $6.56 \times$, $6.28 \times$, and $9.77 \times$ on three platforms (Sparc 128 cores, Intel 16 cores, and IBM 32 cores) respectively