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.

```
// 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.

```
    // 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.

```
    // 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
forall (i: [1..n])  ≡ finish foreach (i: [1..n])
    S                         S
```

IEF and isolated

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

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.

IEF and isolated

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

```
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.

```
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.

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

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

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

Outline

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.

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?

Data Dependence in Task parallel programs - challenges

- 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.

```
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

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)**

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.
```

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**)

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
```

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**)

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;
      S2; // I_A
    }
  }        // finish-end  I_B
  ```

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;
      S2; // I_A
    }
  }        // finish-end  I_B
  ```

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;
    S2; // $I_A$
  }
}          // finish-end  $I_B$
```

- (**Isolated**) 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; $HB(I_A, I_B) = \mathit{true}$, 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;
    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;
}
```

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;
    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 $\texttt{HBD}(A, B) = \mathit{true}$, if

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

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

Happens-before dependence using dynamic HB

Given dynamic HB, and a two statement A and B in a program, we say that $\texttt{HBD}(A, B) = true$, 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. $\neg\exists\ I_C$ in the same execution that writes X such that $HB(I_A, I_C) = true$ and $HB(I_C, I_B) = true$.

HBD details

- If no parallelism → `HBD` = traditional data dependence.
- `HBD` is conservative.
- We classify dependence as *flow*, *anti*, and *output* dependence.

HBD analysis example

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

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.

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)]; }
}
```

$\Longrightarrow$

```
// After loop dist
for (int i = ...)
  /*S1*/  X[f(i)] = ...
for (int 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

Static HBD

Compute Static Happens-before relation.

- Use Program Structure Graph (PSG) as the program representation.

Static HBD

Compute Static Happens-before relation.

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

Static HBD

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.

Static HBD

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.

Static HBD

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.

Static HBD

Compute Static Happens-before relation.

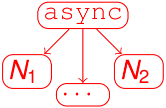
- Use Program Structure Graph (PSG) as the program representation.
    - nodes = root, statement, loop, async, finish, isolated and *call*.
    - edges = subset of abstract syntax tree.
- In two phases.
    - Generate and solve a set of constraints to compute static happens-before information, without considering `isolated` statements.
    - Improve the partial may-happen-before information by considering `isolated` statements.

Static MHB

Phase 1

For each $N_1, N_2 \in Nodes$

1. Same activity:

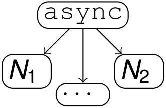


Static MHB

Phase 1

For each $N_1, N_2 \in Nodes$

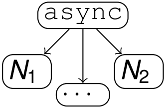
1. Same activity:



Static MHB

Phase 1

For each $N_1, N_2 \in Nodes$

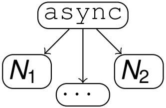
1. Same activity: 

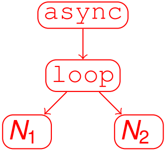
$(N_1, N_2) \in MHB$

Static MHB

Phase 1

For each $N_1, N_2 \in Nodes$

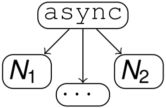
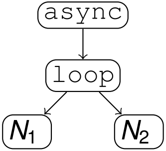
1. Same activity:  $(N_1, N_2) \in MHB$

2. loop ancestor: 

Static MHB

Phase 1

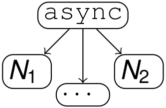
For each $N_1, N_2 \in Nodes$

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

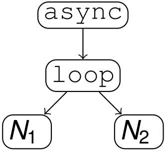
Static MHB

Phase 1

For each $N_1, N_2 \in Nodes$



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

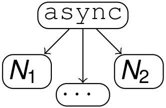
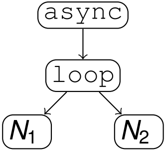
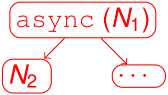


2. loop ancestor: $\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB$;

Static MHB

Phase 1

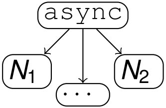
For each $N_1, N_2 \in Nodes$

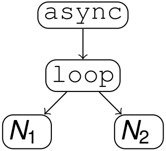
1. Same activity: 
   `async` N_1 $\cdots$ N_2
   $(N_1, N_2) \in MHB$
2. loop ancestor: 
   `async` `loop` N_1 N_2
   $\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB$;
3. Async and stmt: 
   `async` (N_1) N_2 $\cdots$

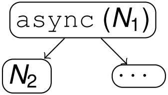
Static MHB

Phase 1

For each $N_1, N_2 \in Nodes$

1. Same activity: 

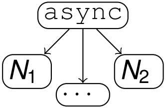
$(N_1, N_2) \in MHB$
2. loop ancestor: 

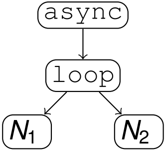
$\{(N_1, N_2), (N_2, N_1)\} \subseteq MHB$;
3. Async and stmt: 

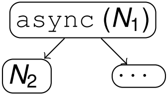
Static MHB

Phase 1

For each $N_1, N_2 \in \mathit{Nodes}$

1. Same activity: 

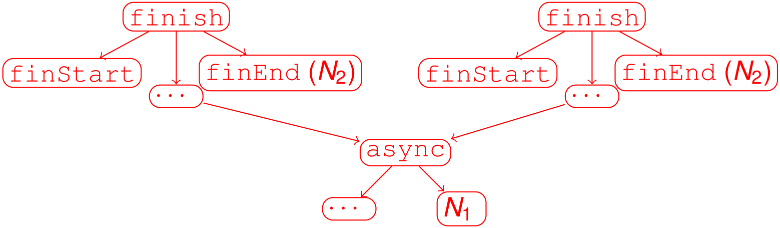
$(N_1, N_2) \in \mathit{MHB}$
2. loop ancestor: 

$\{(N_1, N_2), (N_2, N_1)\} \subseteq \mathit{MHB}$;
3. Async and stmt: 

$(N_1, N_2) \in \mathit{MHB}$;

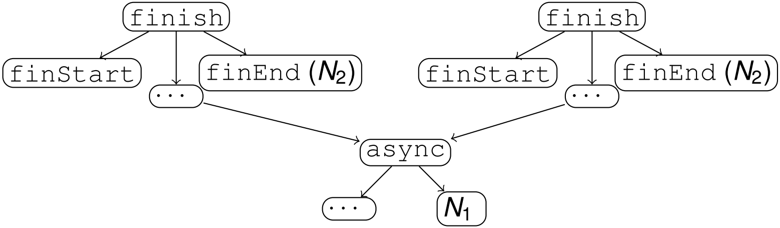
Static MHB (contd)

4. Async and IEF



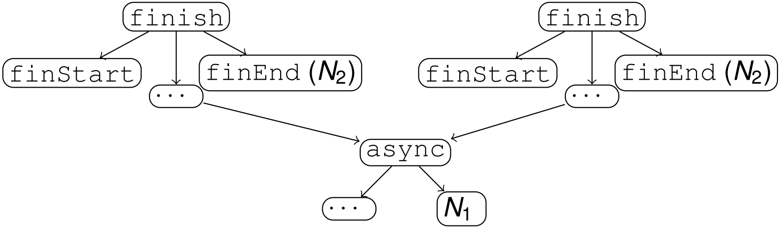
Static MHB (contd)

4. Async and IEF



Static MHB (contd)

4. Async and IEF



$(N_1, N_2) \in MHB$;

5. Tansitivity: if $\exists N_3 \in Nodes$, $(N_1, N_3) \in MHB$ and $(N_3, N_2) \in MHB$ then $(N_1, N_2) \in MHB$.

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 $\texttt{MHBD}(N_1, N_2) = \textit{true}$, if

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 $\texttt{MHBD}(N_1, N_2) = \mathit{true}$, if

i $(N_1, N_2) \in \mathit{MHB}$,

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 $\texttt{MHBD}(N_1, N_2) = \mathit{true}$, 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,

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 $\texttt{MHBD}(N_1, N_2) = \mathit{true}$, if

i $(N_1, N_2) \in \mathit{MHB}$,

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

iii $\neg\exists N_3 \in \mathit{Nodes}$: $\texttt{MHBD}(N_3, N_1) = \mathit{true}$ and $\texttt{MHBD}(N_2, N_3) = \mathit{true}$.

Correctness of a transformation

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.

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 I

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;
```

Extending traditional loop transformations I

1. Serial loop distribution:

```
for (...)  { S1;S2; }
// no dependence cycle between  S1 & S2
```

$\Longrightarrow \left\{ \right.$

```
for (...)  {S1;}
for (...)  {S2;}
```

2. Parallel loop distribution:

```
forall (point p :  R1)
  { S1;  S2; }
// S1 has no dependence on S2
```

$\Longrightarrow \left\{ \right.$

```
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-asyncs in S3
// ¬∃e ∈ E_s: cond has dependence on e
// ¬∃e ∈ E_s:body of e has loop
//      carried dependence on S2, cond or S3
```

$\Longrightarrow \left\{ \right.$

```
S1;
finish
  for (;cond;S2)
    S3;
```

Extending traditional loop transformations I

1. Serial loop distribution:

```
for (...)  { S1;S2; }
// no dependence cycle between S1 & S2
```

$\Longrightarrow$

```
for (...)  {S1;}
for (...)  {S2;}
```

2. Parallel loop distribution:

```
forall (point p :  R1)
  { S1;  S2; }
// S1 has no dependence on S2
```

$\Longrightarrow$

```
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-asyncs in S3
// ¬∃e ∈ Es: cond has dependence on e
// ¬∃e ∈ Es:body of e has loop
//      carried dependence on S2, cond or S3
```

$\Longrightarrow$

```
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
```

$\Longrightarrow$

```
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
```

⟹

```
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 $E_s$ = set of e-asyncs 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
```

⟹

```
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;
```

5. Parallel-serial loop interchange:

```
forall (point p :  R1)
  for  (point q :  R2)  S
// R2 is independent of p
// S contains no break/continue
```

⟹

```
for  (point q :  R2)
  forall (point p :  R1)
   S
```

Extending traditional loop transformations II

6. Loop unpeeling:

```
forall (point p:  R) S1;
S2;
```

// *no break/continue in* `S2`.
// *Say* E_s *= set of e-asyncs in* `S1`
// $\neg\exists e \in E_s$: `S2` *has dependence on* e

$\Longrightarrow$

```
forall (point p:  R)
  {S1;  S2;}
```

7. Loop fusion:

```
forall (point p:  R1) S1;
forall (point p:  R2) S2;
```

// *Say* E_s *= set of e-asyncs in* `S1`
// $\neg\exists e \in E_s$: `S2` *has dependence on* e

$\Longrightarrow$

```
forall (point p:  R1||R2)
 {if (R1.contains (p)) S1;
  ;
  if (R2.contains (p)) S2;}
```

8. Loop switching:

```
if (c)
  forall (point p:  R)
    S;
```

$\Longrightarrow$

```
final boolean v = c;
forall (point p:  R)
  if (v)  S;
```

9. Parallel loop unswitching:

```
forall (point p :  R1)
  if (e) S
```

//`e` *is a pure function and is independent of* `p`

$\Longrightarrow$

```
if (e)
  forall (point p :  R1)  S
```

10. Serial loop unswitching:

```
  for(S2;cond1;S3){
    if (cond2) S4;  else S5;
  }
```

// `cond2` *has no dependence*
//   *on* `S2,S3,S4 and S5,`
// `cond2` *has no side effects*

$\Longrightarrow$

```
  if (cond2) {
    for(S2;cond1;S3) S4;
  } else {
    for(S2;cond1;S3) S5;
  }
```

Variations of traditional transformations

1. Finish distribution:

`finish { S1; S2; }`
// `S1` *has no* `e-async`*s.*

$\Longrightarrow$

```
{ S1;
  finish { S2; }
```

Variations of traditional transformations

1. Finish distribution:

```
finish { S1; S2; }
// S1 has no e-asyncs.
```

$\Longrightarrow$

```
S1;
finish { S2; }
```

2. Finish unswitching:

```
finish
  if(cond) S1; else S2;
// cond has no e-async
```

$\Longrightarrow$

```
if (cond) finish S1;
else finish S2;
```

Variations of traditional transformations

1. Finish distribution:

```
finish { S1; S2; }
// S1 has no e-asyncs.
```

$\Longrightarrow$

```
S1;
finish { S2; }
```

2. Finish unswitching:

```
finish
   if(cond) S1; else S2;
// cond has no e-async
```

$\Longrightarrow$

```
if (cond) finish S1;
else finish S2;
```

3. If expansion:

```
finish {
   S1;
   if(cond) S2; else S3;
   S4; }
// no dependence between cond and S1
```

$\Longrightarrow$

```
finish {
   if (cond)
     {S1; S2; S4;}
   else
     {S1; S3; S4}
}
```

Variations of traditional transformations

1. Finish distribution:

```
finish { S1; S2; }              ⟹  { S1;
// S1 has no e-asyncs.              { finish { S2; }
```

2. Finish unswitching:

```
finish                          ⟹  { if (cond) finish S1;
  if(cond)S1; else S2;              { else finish S2;
// cond has no e-async
```

3. If expansion:

```
                                    { finish {
finish {                            {   if (cond)
  S1;                               {    {S1; S2; S4;}
  if(cond) S2; else S3;         ⟹  {   else
  S4; }                             {    {S1; S3; S4}
// no dependence between cond and S1 { }
```

4. Redundant finish elimination:

```
finish S;                       ⟹  { S;
// S has no e-async.
```

Variations of traditional transformations

1. Finish distribution:

```
finish { S1; S2; }
// S1 has no e-asyncs.
```

$\Longrightarrow$

```
{ S1;
  finish { S2; }
```

2. Finish unswitching:

```
finish
   if(cond)S1; else S2;
// cond has no e-async
```

$\Longrightarrow$

```
{ if (cond) finish S1;
  else finish S2;
```

3. If expansion:

```
finish {
   S1;
   if(cond) S2; else S3;
   S4; }
// no dependence between cond and S1
```

$\Longrightarrow$

```
{ finish {
     if (cond)
       {S1; S2; S4;}
     else
       {S1; S3; S4}
  }
```

4. Redundant finish elimination:

```
finish S;
// S has no e-async.
```

$\Longrightarrow$

```
{ S;
```

5. Tail finish elimination:

```
finish { S1;finish S2;}
```

$\Longrightarrow$

```
{ finish {S1; S2; }
```

Variations of traditional transformations

1. Finish distribution:

```
finish { S1; S2; }
```

// `S1` *has no* `e-async`*s.*

$\Longrightarrow$

```
S1;
finish { S2; }
```

2. Finish unswitching:

```
finish
   if(cond)S1; else S2;
```

// `cond` *has no* `e-async`

$\Longrightarrow$

```
if (cond) finish S1;
else finish S2;
```

3. If expansion:

```
finish {
   S1;
   if(cond) S2; else S3;
   S4; }
```

// *no dependence between* `cond` *and* `S1`

$\Longrightarrow$

```
finish {
   if (cond)
     {S1; S2; S4;}
   else
     {S1; S3; S4}
}
```

4. Redundant finish elimination:

```
finish S;
```

// `S` *has no* `e-async`*.*

$\Longrightarrow$

```
S;
```

5. Tail finish elimination:

```
finish { S1;finish S2;}
```

$\Longrightarrow$

```
finish {S1; S2; }
```

6. Finish fusion

```
finish S1;
finish S2;
```

// *Say* E_s *= set of e-asyncs in* `S1`

// $\neg\exists e \in E_s$: `S2` *has dependence on e*

$\Longrightarrow$

```
finish{
    S1;
    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

Motivating example - finish elimination

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

Motivating example - finish elimination

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

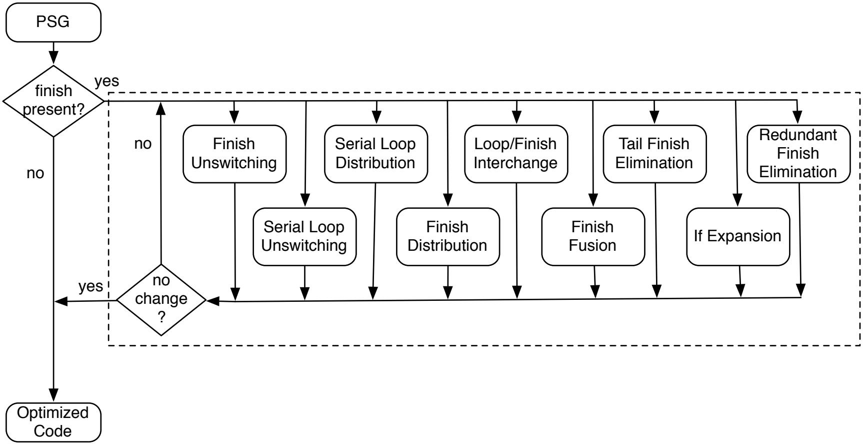
```
void foo(int n) {
  ...
  if (c) {
  finish {
      for (...) {
      async foo(n-1);
      } // for
    } // finish
  } else {
   for (...) {
     foo(n-1);
   } // for
  }
}
```

Motivating example - finish elimination

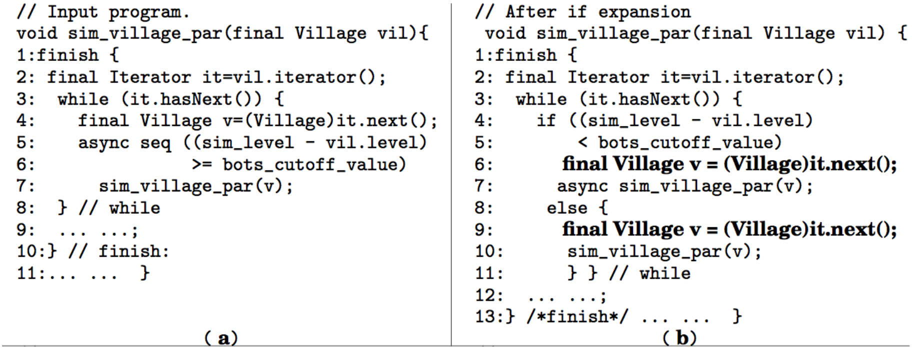
```
void sim_village_par(Village vil){
 // Traverse village hierarchy
1:  finish {
2:   final Iterator it = vil.forward.iterator();
3:   while (it.hasNext()){
4:      final Village v=(Village)it.next();
5:      if ((sim_level-vil.level) < cutoff){
6:          async sim_village_par(v);
       } else {
7:         sim_village_par(v);
       }
       ... ...;}  // while
    } // finish
 }  // end function
```

BOTS Health benchmark

Finish elimination - block diagram



Optimizing the “running” example



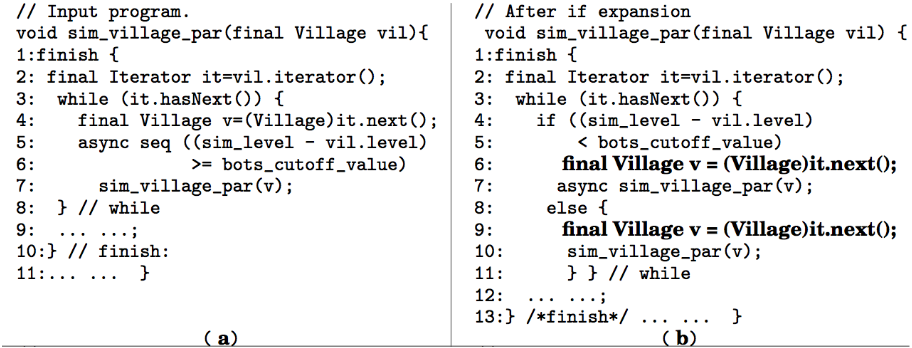
```
// Input program.
void sim_village_par(final Village vil){
finish {
 final Iterator it=vil.iterator();
  while (it.hasNext()) {
    final Village v=(Village)it.next();
    async seq ((sim_level - vil.level)
                >= bots_cutoff_value)
      sim_village_par(v);
  } // while
  ... ...;
}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

(b)

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:
... ...   }
```

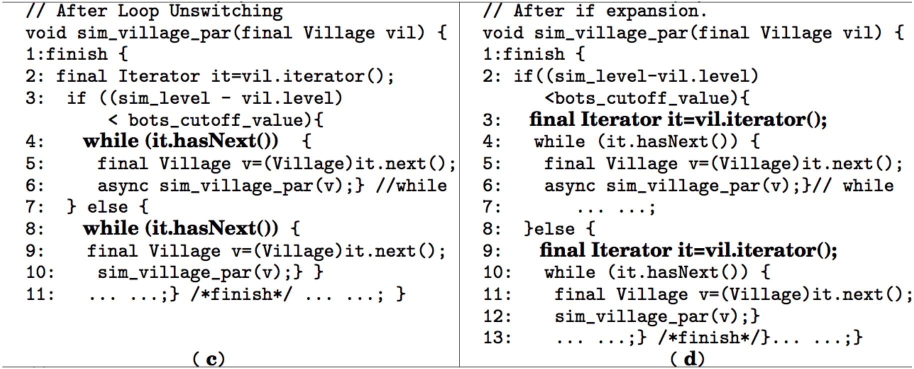
(a)

```
// 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*/ ... ...   }
```

(b)

Next: Loop unswitching

Optimizing the “running” example



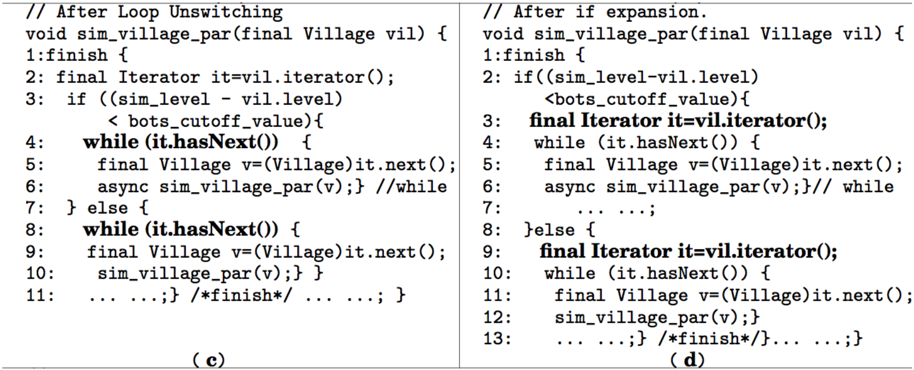
```
// After Loop Unswitching
void sim_village_par(final Village vil) {
finish {
final Iterator it=vil.iterator();
 if ((sim_level - vil.level)
      < bots_cutoff_value){
   while (it.hasNext())  {
    final Village v=(Village)it.next();
    async sim_village_par(v);} //while
 } else {
   while (it.hasNext())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

(c)

```
// 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
      ... ...;
  }else {
    final Iterator it=vil.iterator();
    while (it.hasNext()) {
     final Village v=(Village)it.next();
     sim_village_par(v);}
     ... ...;} /*finish*/}... ...;}
```

(d)

Optimizing the “running” example



```
// 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*/ ... ...; }
```

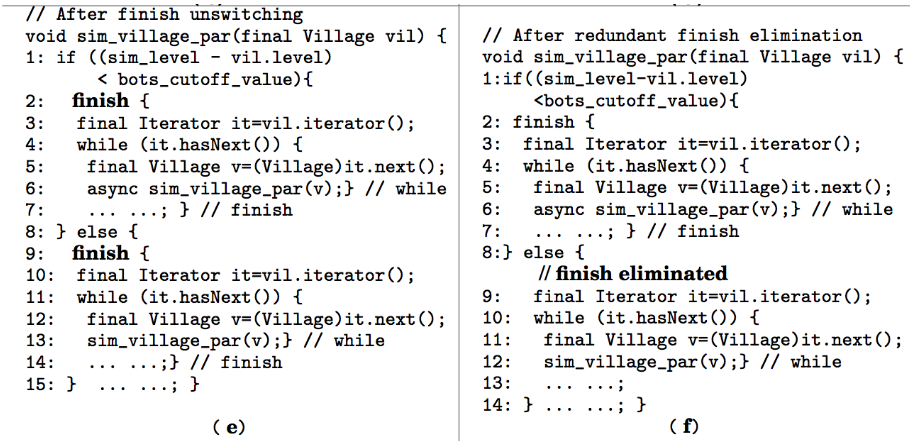
(c)

```
// 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
      ... ...;
  }else {
    final Iterator it=vil.iterator();
    while (it.hasNext()) {
     final Village v=(Village)it.next();
     sim_village_par(v);}
     ... ...;} /*finish*/}... ...;}
```

(d)

Next: finish unswitching

Optimizing the “running” example



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

(e)

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

(f)

Transformations in the presence of exceptions

Finish distribution:		(no exceptions)
`finish { S1; S2; }` // `S1` *has no* `e-async`*s.*	$\Longrightarrow$	$\left\{ \begin{array}{l} \texttt{S1;} \\ \texttt{finish \{ S2; \}} \end{array} \right.$

Transformations in the presence of exceptions

Finish distribution: (no exceptions)

```
finish { S1; S2; }
// S1 has no e-asyncs.
```

$\Longrightarrow$

```
{ S1;
  finish { S2; }
```

Finish distribution: (with exceptions)

```
finish { S1; S2; }
// (1) S1 has no e-asyncs.
// (a) S2 has e-asyncs.
```

$\Longrightarrow$

```
{ try {S1;}
  catch(Exception e){
   MultiException me tt=new ···;
    me.pushEx(e1); throw me; }
  finish { S2; }
```

Conclusion

- 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×, 6.28×, and 9.77× on three platforms (Sparc 128 cores, Intel 16 cores, and IBM 32 cores) respectively