Inferring Nondeterministic Sequential Specifications for Parallelism Correctness

Jacob Burnim, Tayfun Elmas*
George Necula, Koushik Sen
University of California, Berkeley
Correctness of parallel programs

parallel_integer_matrix_multiply(A, B) {

    // compute C = A * B using your // favorite parallel algorithm

    return C
}

Functional specification: \( \forall i, j. \ C_{i,j} = \sum_{k} (A_{i,k} \times B_{k,j}) \)
Checking correctness of parallel programs

Parallel program

Satisfies?

Complicated, end-to-end reasoning

Functional specification
Our proposal: Separate reasoning about functional correctness and thread schedules

Parallel program → Satisfies? → Functional specification

Parallel program → Satisfies? → Nondeterministic sequential specification → Satisfies? → Functional specification

Complicated, end-to-end reasoning

Two independent, simpler verification tasks
Our proposal: Separate reasoning about functional correctness and thread schedules.

Parallelism correctness.

Nondeterministic sequential specification

Parallel fractal generation

Functional specification
Our proposal: Separate reasoning about functional correctness and thread schedules.
Outline

Last retreat: How to write?
Run-time checking

Last retreat: How to write?
Nondeterministic sequential specification

This talk: How to infer?
Functional specification

Parallel program

[Burnim, Elmas, Necula, Sen, PLDI 2011]
Parallel program

```c
......
parallel-for(...) {
    .....  
    S  
    .....  
}
......
```

NDSeq spec

```c
......
nd-for(...) {
    .....  
    if(*) S  
    .....  
}
......
```
Example 1: Parallel reduction

Parallel program

```plaintext
parallel-for(i = 1 to N){
  while(true){
    t1 = x
    t2 = (t1 + i) ** 2
    if(CAS(x, t1, t2))
      end_iteration
    print “retrying”
  }
}
print x
```

NDSeq spec

```plaintext
nd-for(i = 1 to N){
  while(true){
    t1 = x
    t2 = (t1 + i) ** 2
    if(*)
      if(CAS(x, t1, t2))
        end_iteration
    print “retrying”
  }
}
print x
```
Example 2: Branch and bound

Parallel program

```
parallel-for(i = 1 to N){
    b = lower_bound(i)
    if(b >= lowest_cost)
        end_iteration

    c = compute_cost(i)
    synchronized_by(lock){
        if(c < lowest_cost){
            lowest_cost = c
            best_soln = s
        }
    }
}
print best_soln
```

NDSeq spec

```
nd-for(i = 1 to N){
    if(*){
        b = lower_bound(i)
        if(b >= lowest_cost)
            end_iteration
    }

    c = compute_cost(i)
    synchronized_by(lock){
        if(c < lowest_cost){
            lowest_cost = c
            best_soln = s
        }
    }
}
print best_soln
```
Outline

Last retreat: How to write?

This talk: How to infer?

Last retreat: Runtime checking

Parallel program

Nondeterministic sequential specification

Satisfies?

Satisfies?

Functional specification

[Burnim, Elmas, Necula, Sen, PLDI 2011]
Parallelism correctness: Semantic definition

For each parallel execution

\[ \text{Initial State } s_0 \rightarrow \text{Final State } s_1 \]

there exists an equivalent NDSeq execution

\[ \text{Initial State } s_0 \rightarrow \text{Final State } s_1 \]
Q: What does checking mean?

For each parallel execution (generated by testing)

Initial State $s_0$ $\rightarrow$ $\rightarrow$ $\rightarrow$ Final State $s_1$

can we show that there exists an equivalent NDSeq execution?
Q: What does checking mean?
A: Serialize all threads in execution

For each parallel execution (generated by testing)

Initial State $s_0$ → Final State $s_1$

Each thread runs without interleaving

can we show that there exists an equivalent NDSeq execution?
Q: What does checking mean?
A: Serialize all threads in execution
When is it safe to move actions?

Initial State $s_0$ → Can move? → Final State $s_1$

Initial State $s_0$ → Can move? → Final State $s_1$

Initial State $s_0$ → Can move? → Final State $s_1$

Initial State $s_0$ → Can move? → Final State $s_1$
When is it safe to move actions?

Initial State $s_0$ → Final State $s_1$

Initial State $s_0$ → Can move? → Final State $s_1$
Conflicting actions: Cannot move!

Initial State \( s_0 \) → Initial State \( s_0 \)

\[ x = 1 \]

\[ t = x \]

\[ \text{if} \ (t == 1) \]

Final State \( s_1 \) → Final State \( s_1 \)

Cannot move!
Conflict cycles

Cycle of conflicting actions

Initial State $s_0$  \[ \rightarrow \]  Final State $s_1$

Cannot order black and blue threads!

**Conflict serializability:**

**Classic Theorem:** If no conflict cycle, then there exists an equivalent NDSeq execution.
Can we ignore some conflict?

Cycle of conflicting actions

Ignore some actions (Hints from NDSeq spec)

No longer creates a conflict cycle

Theorem: If no cycle remains after ignoring actions, then there exists an equivalent NDSeq execution.
Outline

- Last retreat: How to write?
  - Runtime checking

- Last retreat: How to write?
  - Nondeterministic sequential specification

- This talk: How to infer?
  - Functional specification

[Burnim, Elmas, Necula, Sen, PLDI 2011]
Our approach: Check if there exists an equivalent NDSeq execution

Parallel execution trace

Traditional conflict serializability

conflict cycles?

Ignore some actions based on NDSeq spec

if(*)s in NDSeq spec

No conflict cycle
⇒ exists equivalent NDSeq execution

No conflict cycle
⇒ exists equivalent NDSeq execution

Conflict cycle
⇒ cannot find equivalent NDSeq execution
Encoding parallelism correctness as SAT

Theorem: If SAT(P) then exists an equivalent NDSeq execution.

Idea: If $X_A = 1$ then action $A$ is ignored.

For each conflict cycle $C = \{A_1, \ldots, A_n\}$:
- Add constraint: at least one $X_{A_i} = 1$

if(*)'s are inputs:

if(*) { S } \rightarrow X_S = 1

⇒ exists equivalent NDSeq execution

Conflict cycle
⇒ cannot find equivalent NDSeq execution

Parallel execution trace

Traditional conflict serializability

conflict cycles?

if(*)s in NDSeq spec
Obtaining if(*)s from solution to SAT

Parallel execution trace

Traditional conflict serializability

Theorem 1: If SAT(P) then exists an equivalent NDSeq execution.

Theorem 2: If exist if(*)s that allows to ignore cycles, solution selects them.

if(*)’s are output:

\[ X_S = 1 \Rightarrow \text{if(*)} \{ S \} \]

\( \Rightarrow \) exists equivalent NDSeq execution

Conflict cycle \( \Rightarrow \) cannot find equivalent NDSeq execution

Generate formula P

Check SAT(P)

if(*)s in NDSeq spec

conflict cycles?
BUT, which if(*)s are necessary?

Parallel execution trace

Conflict cycles?

if(*)s in NDSeq spec

Generate formula P

Check SAT(P)

Need:
1) Output only necessary if(*)s.
2) Output minimum number of necessary if(*)s.

WHY? (Functional correctness)

⇒ exists equivalent NDSeq execution

Con?ict cycle
⇒ cannot find equivalent NDSeq execution
Minimal NDSeq spec as MinCostSAT

MinCostSAT(P): Minimize number of $X_s=1$ (statements to add if(*))

Theorem 1: If SAT(P) then exists an equivalent NDSeq execution.

Theorem 2: Solution selects minimum # of if(*)s necessary to ignore actions in conflict cycles.

Parallel execution trace → Conflict cycle? → If(*)s in NDSeq spec

Generate formula P → Check MinCostSAT(P) (P): Minimize number of $X_s=1$ (statements to add if(*))

⇒ exists equivalent NDSeq execution

⇒ cannot find equivalent NDSeq execution
## Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Line of code</th>
<th>Size of trace</th>
<th>Irrelevant events</th>
<th>Number of parallel constructs</th>
<th>Number of if(*)s (manual)</th>
<th>Number of if(*)s (inferred)</th>
<th>Inferred if(*)s right?</th>
</tr>
</thead>
<tbody>
<tr>
<td>phylogeny (fixed)</td>
<td>4.4K</td>
<td>29K</td>
<td>24K</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>yes</td>
</tr>
<tr>
<td>concurrent stack</td>
<td>40</td>
<td>1K</td>
<td>350</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>concurrent queue</td>
<td>60</td>
<td>320</td>
<td>110</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>mesh refinement</td>
<td>1K</td>
<td>930K</td>
<td>845K</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>sor</td>
<td>300</td>
<td>905K</td>
<td>561K</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>matmult</td>
<td>700</td>
<td>962K</td>
<td>8K</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>series</td>
<td>800</td>
<td>2008K</td>
<td>1.2K</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>crypt</td>
<td>1.1K</td>
<td>493K</td>
<td>100K</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>moldyn</td>
<td>1.3K</td>
<td>4517K</td>
<td>4300K</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>lufact</td>
<td>1.5K</td>
<td>1048K</td>
<td>792K</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>raytracer (fixed)</td>
<td>1.9K</td>
<td>9125K</td>
<td>8960K</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3.6K</td>
<td>1723K</td>
<td>731K</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>pi3</td>
<td>150</td>
<td>1062K</td>
<td>141</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>keysearch3</td>
<td>200</td>
<td>1050K</td>
<td>1049K</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>250</td>
<td>576K</td>
<td>330K</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>yes</td>
</tr>
<tr>
<td>raytracer (buggy)</td>
<td>1.9K</td>
<td>9125K</td>
<td>8960K</td>
<td>1</td>
<td>0</td>
<td>UNSAT</td>
<td>-</td>
</tr>
<tr>
<td>phylogeny (buggy)</td>
<td>4.4K</td>
<td>29K</td>
<td>24K</td>
<td>2</td>
<td>3</td>
<td>UNSAT</td>
<td>-</td>
</tr>
</tbody>
</table>
Conclusion

• **Problem:** Adding if(*) most difficult part of writing NDSeq specs

• **Solution:** Sound, automated inference for if(*)s; formulated as MinCostSAT
  — Reasonable starting point for writing NDSeq specs

• Promising evidence for fully-automated testing and verification of parallel programs using NDSeq specs