Asserting and Checking Determinism for Parallel Programs

Jacob Burnim
Koushik Sen
University of California, Berkeley
Motivation

- **Key:** Easy and worthwhile to specify deterministic behavior of parallel programs
- Parallel programming is difficult
- Culprit: **Non-determinism**
  - Interleaving of parallel threads.
- Often, non-determinism is **internal**
  - Same input => semantically same output
  - Parallel code is outwardly **sequential**
Motivation

- **Goal**: Separately specify/check parallelism and functional correctness.
  - Show parallelism is deterministic.
  - Reason about correctness sequentially.
  - Decomposes correctness proof (or testing)!

- **Example**:
  - Write Cilk program and prove (or test) sequential correctness.
  - Add parallelism, answers should not change
Motivation

- How to specify correctness of parallelism?

**Implicit:**
No *sources* of non-determinism (no data races)

**Explicit:**
Full functional correctness.

Determinism specification: A sweet spot?
- Lightweight, but precise.
Outline

- Motivation
- Deterministic Specification
- Experimental Evaluation
- Related Work
- Future Work + Conclusions
Goal: Specify deterministic behavior.

- Same initial parameters => same image.
- Non-determinism is internal.

```plaintext
// Parallel fractal render
mandelbrot(params, img);
```
Deterministic Specification

- **Specifies**: Two runs from same initial program state have same result state.

\[ \forall s_0 \xrightarrow{m} s_1, \quad s_0 \xrightarrow{m} s'_1 : \quad s_1 = s'_1 \]
Deterministic Specification

- Too restrictive – different schedules may give slightly different floating-point results.

```c
double A[][], b[], x[];
...
deterministic {
  // Solve A*x = b in parallel
  lufact_solve(A, b, x);
}
```
Deterministic Specification

Too restrictive – internal structure of set may differ depending on order of adds.
Deterministic Specification

- Too restrictive – search can correctly return any tree with optimal cost.
Semantic Determinism

- Too strict to require every interleaving to give exact same program state:

\[ \forall s_0 \xrightarrow{P} s_1, \ s_0 \xrightarrow{P} s_1' : \ s_1 = s_1' \]
Semantic Determinism

- Too strict to require every interleaving to give exact same program state:

\[
\forall s_0, s_1, s_0', s_1' : \begin{array}{c}
\text{deterministic} \\
\{ \\
P \\
\} \\
\end{array}
\]

Predicate! Should be user-defined.

\[
\forall S_0 \xrightarrow{P} S_1, S_0' \xrightarrow{P} S_1' : S_1 = S_1'
\]
Semantic Determinism

- Too strict to require every interleaving to give exact same program state:

\[
\text{deterministic} \{
\begin{align*}
P \\
\end{align*}
\}
\text{ assert } \text{Post}(s_1, s_1')
\]

- **Specifies**: Final states are equivalent.

\[
\forall \ s_0 \xrightarrow{P} s_1, \ s_0 \xrightarrow{P} s_1': \ \text{Post}(s_1, s_1')
\]
double A[][[], b[], x[]];
...
deterministic {
// Solve A\*x = b in parallel
lufact_solve(A, b, x);
} assert (|x - x'| < \epsilon)
Semantic Determinism

- Resulting sets are semantically equal.

```java
set t = new RedBlackTreeSet();
deterministic {
    t.add(3) || t.add(5);
} assert (t.equals(t'))
```
Semantic Determinism

deterministic {
// Parallel branch-and-bound
Tree t = min_phylo_tree(data);
} assert (t.cost == t’.cost())
Preconditions for Determinism

- **Too strict** – initial states must be identical
  - Not compositional.

```java
set t = ...

deterministic {
    t.add(3) || t.add(5);
} assert (t.equals(t'))

...

deterministic {
    t.add(4) || t.add(6);
} assert (t.equals(t'))
```
Preconditions for Determinism

- Too strict to require identical initial states:

\[
\text{deterministic} \{ \\
P \\
\} \text{ assert } \text{Post}(s_1, s_1')
\]

\[
\forall s_0 \xrightarrow{P} s_1, \ s_0 \xrightarrow{P} s_1' : \text{Post}(s_1, s_1')
\]
Preconditions for Determinism

- Too strict to require identical initial states:

\[
deterministic \quad \text{assume } (s_0 = s_0') \{ \\
\quad P \\
\} \quad \text{assert } \text{Post}(s_1, s_1')
\]

\[
\forall s_0 \xrightarrow{P} s_1, \ s_0' \xrightarrow{P} s_1' : \\
\quad s_0 = s_0' \Rightarrow \text{Post}(s_1, s_1')
\]
Preconditions for Determinism

- Too strict to require identical initial states:

\[
\text{deterministic assume } (s_0 = s_0') \{ \\
\quad P \\
\} \text{ assert } \text{Post}(s_1, s_1')
\]

\[
\forall s_0 \xrightarrow{P} s_1, \quad s_0' \xrightarrow{P} s_1' : \quad s_0 = s_0' \implies \text{Post}(s_1, s_1')
\]

Predicate! Should be user-defined.
Preconditions for Determinism

- Too strict to require identical initial states:

\[ \text{deterministic assume } \text{Pre}(s_0, s_0') \{ \]
\[ P \]
\[ \} \text{ assert } \text{Post}(s_1, s_1') \]

- Specifies:

\[ \forall s_0 \xrightarrow{P} s_1, s_0' \xrightarrow{P} s_1': \]
\[ \text{Pre}(s_0, s_0') \Rightarrow \text{Post}(s_1, s_1') \]
deterministic assume $Pre(s_0, s_0')$ 

$P$

} assert $Post(s_1, s_1')$

"Bridge" predicate

"Bridge" assertion
Preconditions for Determinism

\[
\text{set } t = \ldots
\]

\textbf{deterministic}

\textbf{assume} (\texttt{t.equals(t'))} {
  \texttt{t.add(4) || t.add(6);}
}\textbf{assert} (\texttt{t.equals(t'))}

- **Specifies**: Semantically equal sets yield semantically equal sets.
Checking Determinism

deterministic assume Pre(s₀,s₀′) {
    P
} assert Post(s₁,s₁′)

- Run P on some number of schedules.
- For every pair $s₀ \rightarrow s₁$ and $s₀′ \rightarrow s₁′$ of executions of P:

\[ Pre(s₀,s₀′) \Rightarrow Post(s₁,s₁′) \]
Outline

- Motivation
- Deterministic Specification
- Experimental Evaluation
  - Ease of Use
  - Effectiveness in Finding Bugs
- Related Work
- Future Work + Conclusions
Ease of Asserting Determinism

- Implemented a deterministic assertion library for Java.
- Manually added deterministic assertions to 13 Java benchmarks with 200 – 4k LoC
- Typically ~10 minutes per benchmark
  - Functional correctness very difficult.
Deterministic Assertion Library

- Implemented assertion library for Java:

  ```java
  Predicate eq = new Equals();
  Deterministic.open();
  Deterministic.assume(set, eq);
  ...
  Deterministic.assert(set, eq);
  Deterministic.close();
  ```

- Records `set` to check:

  ```java
  eq.apply(set_0, set_0') => eq.apply(set, set')
  ```
Ease of Use: Example

Deterministic.open();
Predicate eq = new Equals();
Deterministic.assume(width, eq);
... (9 parameters total) ...  
Deterministic.assume(gamma, eq);

// Compute fractal in threads  
int matrix[][][] = ...;

Deterministic.assert(matrix, eq);
Deterministic.close();
Effectiveness in Finding Bugs

- 13 Java benchmarks of 200 – 4k LoC
- Ran benchmarks on 100-1000 schedules
  - Schedules with data races and other “interesting” interleavings (active testing)
- For every pair of executions of deterministic Pre { P } Post:
  \[ s_0 \xrightarrow{P} s_1, \quad s_0' \xrightarrow{P} s_1' \]
  check that: \[ \text{Pre}(s_0, s_0') \Rightarrow \text{Post}(s_1, s_1') \]
## Experiments: Java Grande Forum

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LoC</th>
<th>Data Races</th>
<th></th>
<th>High-Level Races</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Found</td>
<td>Violations</td>
<td>Found</td>
</tr>
<tr>
<td>sor</td>
<td>300</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>moldyn</td>
<td>1.3k</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lufact</td>
<td>1.5k</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>1.9k</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3.6k</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
## Experiments: Parallel Java Lib

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LoC</th>
<th>Data Races Found</th>
<th>Violations</th>
<th>High-Level Races Found</th>
<th>Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>150</td>
<td>9</td>
<td>0</td>
<td>1+</td>
<td>1</td>
</tr>
<tr>
<td>keysearch3</td>
<td>200</td>
<td>3</td>
<td>0</td>
<td>0+</td>
<td>0</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>250</td>
<td>9</td>
<td>0</td>
<td>0+</td>
<td>0</td>
</tr>
<tr>
<td>phylogeny</td>
<td>4.4k</td>
<td>4</td>
<td>0</td>
<td>0+</td>
<td>0</td>
</tr>
<tr>
<td>tsp*</td>
<td>700</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Experimental Evaluation

- Across 13 benchmarks:
- Found 40 data races.
  - 1 violates deterministic assertions.
Experimental Evaluation

- Across 13 benchmarks:
  - Found 40 data races.
    - 1 violates deterministic assertions.
- Found many “interesting” interleavings (non-atomic methods, lock races, etc.)
  - 1 violates deterministic assertions.
Determinism Violation

- Pair of calls to `nextDouble()` must be atomic.

```java
Deterministic {
    // N trials in parallel.
    foreach (n = 0; n < N; n++) {
        x = Random.nextDouble();
        y = Random.nextDouble();
        ...
    }
} assert (|\pi - \pi'| < 1e-10)
```
Outline

- Motivation
- Deterministic Specification
- Experimental Evaluation
- Related Work
- Future Work + Conclusions
Determinism vs. Atomicity

- **Internal vs. external** parallelism/non-determinism
  - Complementary notions

### Diagram

- **Deterministic**
  - "Closed program"
- **Atomic**
  - "Open program"
Related Work: SingleTrack

- [Fruend, Flanagan, ESOP09]
- Dynamic determinism checker.
  - Treats as atomicity with internal parallelism.
- Communication + results must be identical for every schedule.
Related Work: DPJ

- Deterministic Parallel Java
  [Bocchino, Adve, Adve, Snir, HotPar 09]

- Deterministic by default.
  - Enforced by static effect types.

- Bit-wise identical results for all schedules.

- “Safe” non-determinism quarantined in libraries.
Outline

- Motivation
- Deterministic Specification
- Experimental Evaluation
- Related Work
- Future Work + Conclusions
Verifying Determinism

- Verify determinism of each piece.
- No need to consider cross product of all interleavings.
Verifying Determinism

- Compositional reasoning for determinism?
Conclusions

- “Bridge” predicates and assertions
  - Simple to assert natural determinism
  - Semantic, user-specified determinism

- Can distinguish harmful from benign data races, non-atomic methods, etc.

- Can we prove/verify determinism?
  - Enable us to prove correctness sequentially?
Any Questions?

email jburnim@cs.berkeley.edu