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

Deterministic Specification

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

- **Goal:** Specify deterministic behavior.
  - Same initial parameters => same image.
  - Non-determinism is internal.

Deterministic Specification

```
deterministic {
   // Parallel fractal render
   mandelbrot(params, img);
}
```

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

$$\forall\ s_0 \xrightarrow{m} s_1,\ s_0 \xrightarrow{m} s_1' :\ s_1 = s_1'$$

Deterministic Specification

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

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

Deterministic Specification

```
set t = new RedBlackTreeSet();
deterministic {
   t.add(3)  || t.add(5);
}
```

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

Deterministic Specification

```
deterministic {
  // Parallel branch-and-bound
  Tree t = min_phylo_tree(data);
}
```

- 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:

```
deterministic {
   P
}
```

$$\forall\ s_0 \xrightarrow{P} s_1,\ s_0 \xrightarrow{P} s_1' :\ s_1 = s_1'$$

Semantic Determinism

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

```
deterministic {
    P
}
```

Predicate!
Should be
user-defined.

$$\forall\ s_0 \xrightarrow{P} s_1,\ s_0 \xrightarrow{P} s_1' : \boxed{s_1 = s_1'}$$

Semantic Determinism

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

```
deterministic {
   P
} assert Post(s₁,s₁')
```

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

$$\forall\ s_0 \xrightarrow{P} s_1,\ s_0 \xrightarrow{P} s_1' :\ \mathrm{Post}(s_1, s_1')$$

Semantic Determinism

```
double A[][], b[], x[];
...
deterministic {
   // Solve A*x = b in parallel
   lufact_solve(A, b, x);
} assert (|x – x’| < ε)
```

“Bridge” predicate

Semantic Determinism

```
set t = new RedBlackTreeSet();
deterministic {
   t.add(3)  || t.add(5);
} assert (t.equals(t’))
```

- Resulting sets are *semantically* equal.

Semantic Determinism

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

Preconditions for Determinism

```
set t = …
deterministic {
    t.add(3) || t.add(5);
} assert (t.equals(t’))
…
deterministic {
    t.add(4) || t.add(6);
} assert (t.equals(t’))
```

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

Preconditions for Determinism

- Too strict to require identical initial states:

```
deterministic {
    P
} assert Post(s₁,s₁')
```

$$\forall\ s_0 \xrightarrow{P} s_1,\ s_0 \xrightarrow{P} s_1' : \mathrm{Post}(s_1, s_1')$$

Preconditions for Determinism

- Too strict to require identical initial states:

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

$$\forall\ s_0 \xrightarrow{P} s_1,\ s_0{}' \xrightarrow{P} s_1{}' :$$

$$s_0 = s_0{}' \quad \Rightarrow \quad \mathrm{Post}(s_1,\ s_1{}')$$

Preconditions for Determinism

- Too strict to require identical initial states:

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

Predicate! Should be user-defined.

$$\forall\, s_0 \xrightarrow{P} s_1,\ s_0' \xrightarrow{P} s_1'$$

$$\boxed{s_0 = s_0'} \implies \mathrm{Post}(s_1,\ s_1')$$

Preconditions for Determinism

- Too strict to require identical initial states:

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

- **Specifies**:

$$\forall\; s_0 \xrightarrow{P} s_1,\; s_0' \xrightarrow{P} s_1' :$$

$$\mathrm{Pre}(s_0,\, s_0') \;\Rightarrow\; \mathrm{Post}(s_1,\, s_1')$$

Bridge predicates/assertions

$$\texttt{deterministic assume } \underbrace{\texttt{Pre}(s_0, s_0')}_{\text{“Bridge” predicate}} \ \{$$

$$\quad \texttt{P}$$

$$\} \ \underbrace{\texttt{assert Post}(s_1, s_1')}_{\text{“Bridge” assertion}}$$

Preconditions for Determinism

```
set t = ...
deterministic
assume (t.equals(t’)) {
   t.add(4) || t.add(6);
} assert (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 $s_0 \rightarrow s_1$ and $s_0' \rightarrow s_1'$ of executions of P:

$$Pre(s_0, s_0') \Rightarrow Post(s_1, s_1')$$

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:

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

- Records `set` to check:

  $eq.apply(set_0, set_0') \Rightarrow eq.apply(set, set')$

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: $\mathrm{Pre}(s_0, s_0') \Rightarrow \mathrm{Post}(s_1, s_1')$

Experiments: Java Grande Forum

Benchmark	LoC	Data Races Found	Data Races Violations	High-Level Races Found	High-Level Races Violations
sor	300	2	0	0	0
moldyn	1.3k	2	0	0	0
lufact	1.5k	1	0	0	0
raytracer	1.9k	3	**1**	0	0
montecarlo	3.6k	1	0	2	0

Experiments: Parallel Java Lib

Benchmark	LoC	Data Races Found | Violations		High-Level Races Found | Violations	
pi	150	9	0	1+	1
keysearch3	200	3	0	0+	0
mandelbrot	250	9	0	0+	0
phylogeny	4.4k	4	0	0+	0
tsp*	700	6	0	2	0

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

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

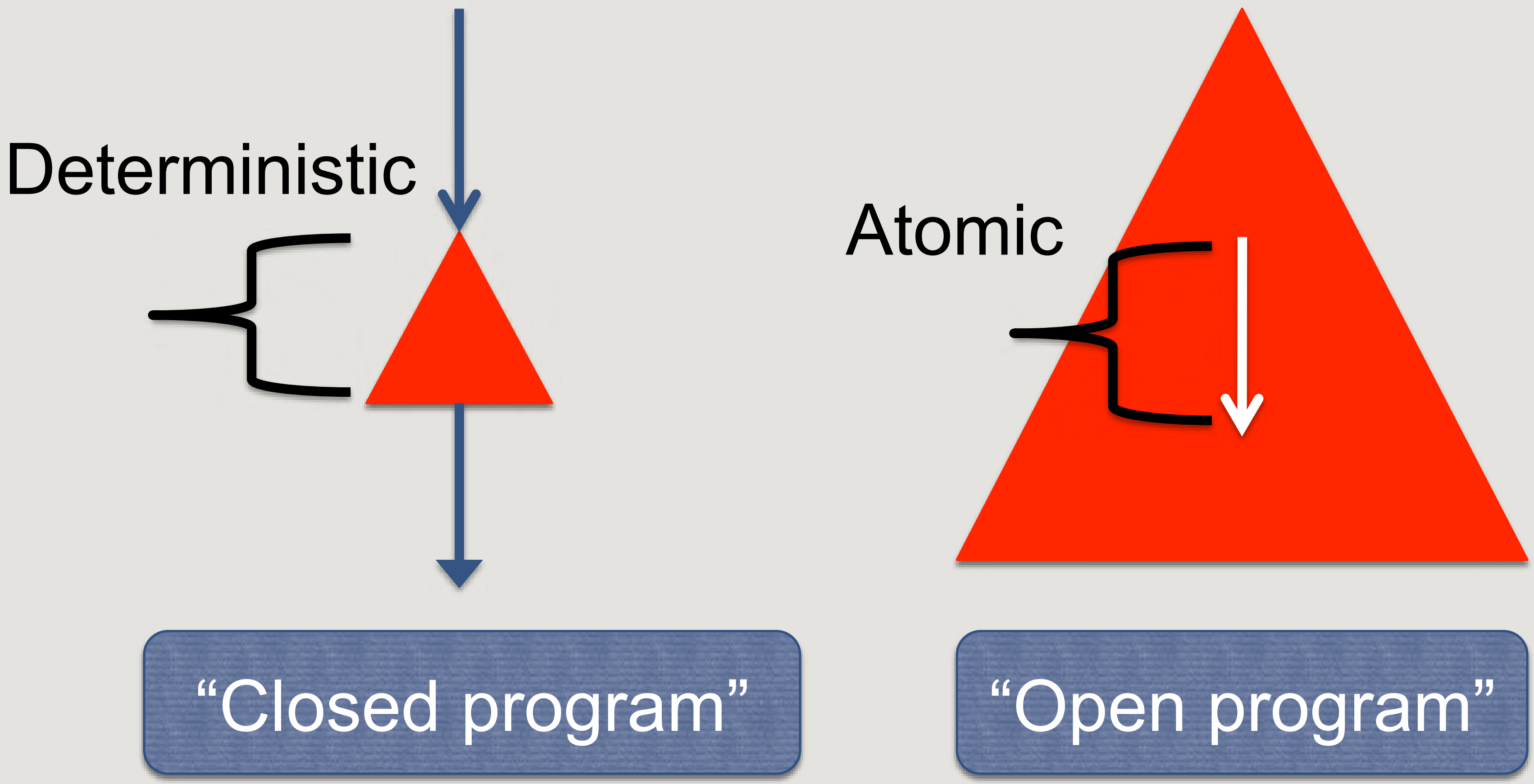
- Pair of calls to `nextDouble()` must be atomic.

Outline

- Motivation
- Deterministic Specification
- Experimental Evaluation
- **Related Work**
- Future Work + Conclusions

Determinism vs. Atomicity

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



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

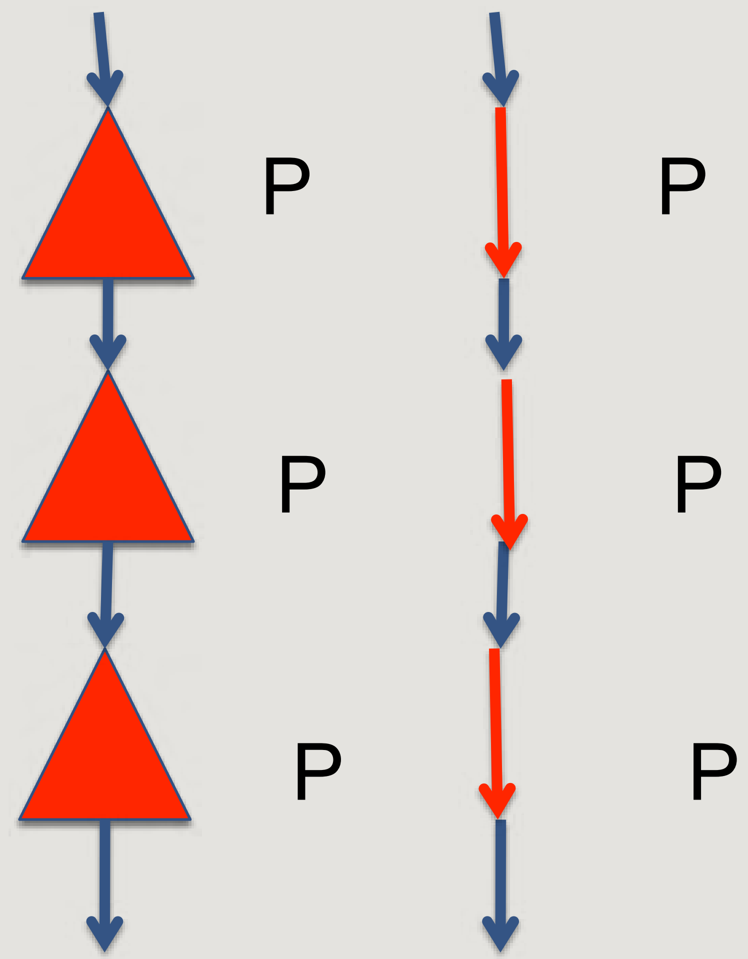
- Deterministic Parallel Java [Bocchino, Adve, Adve, Snir, HotPar 09]
- Deterministic by default.
  - Enforced by static effect types.
- Bit-wise identical results for all schedules.
- “Safe” non-determinism quarantined in libraries.

Outline

- Motivation
- Deterministic Specification
- Experimental Evaluation
- Related Work
- **Future Work + Conclusions**

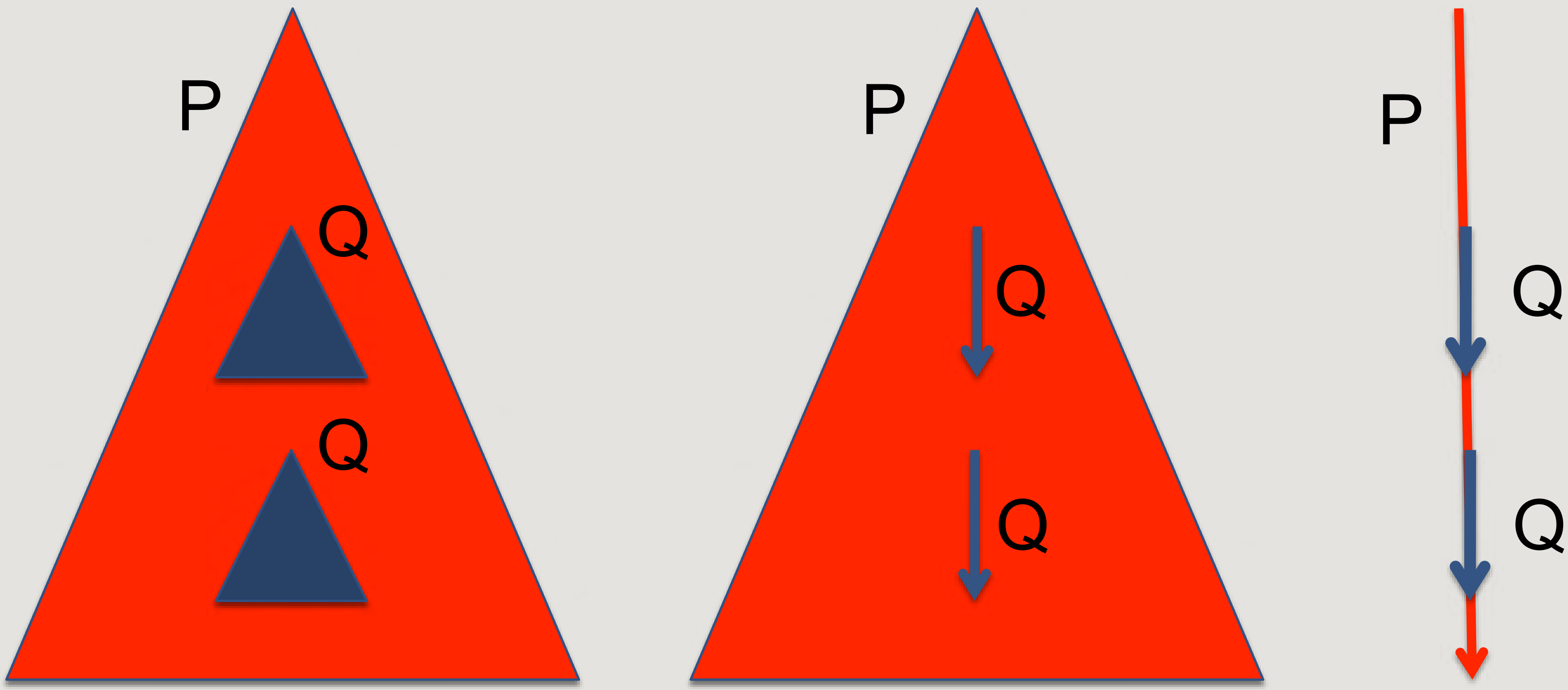
Verifying Determinism

- 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