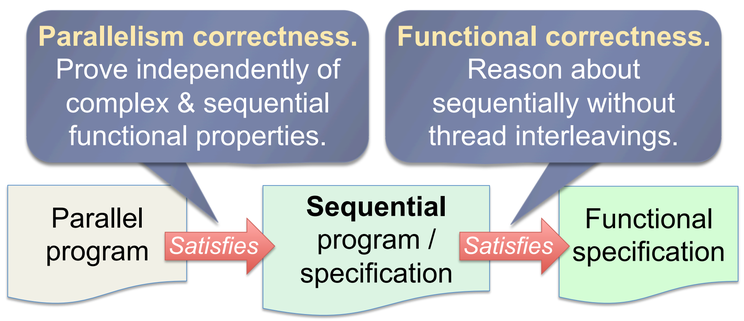
NDSeq: Specifying and Checking Parallelism Correctness Using Nondeterministic Sequential Programs

Jacob Burnim, Tayfun Elmas, George Necula, Koushik Sen

Berkeley Parlab

OVERVIEW

- Verifying parallel programs is very challenging.
  - Painful to reason simultaneously about correctness of parallelism and about functional correctness.
  - Functional correctness often largely sequential.
- **Goal**: Decompose effort of verifying parallelism and verifying functional correctness.



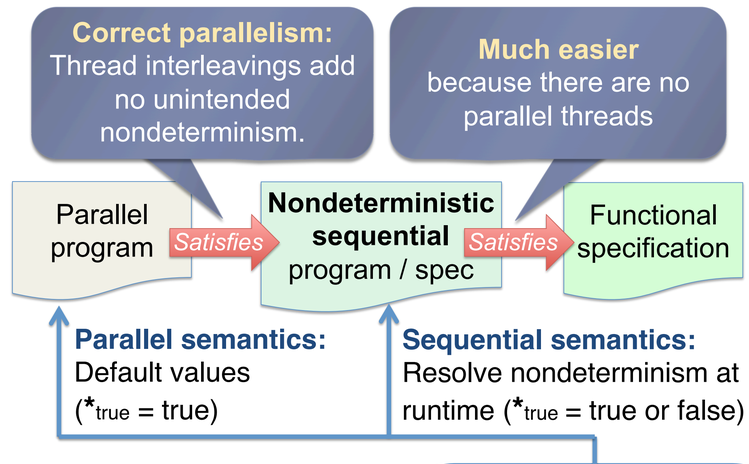
- **Question:** What is **parallelism correctness**?
- **Previous work:** Deterministic specifications. [Burnim and Sen, FSE 2009]
  - **Idea:** **Parallel correctness** means every thread schedule gives semantically equivalent results.

```
deterministic assume (data == data') {
  // Parallel branch-and-bound search
  Solution best = parallel-search(data);
} assert (cost(best) == cost(best'));
```

Figure 1. Deterministic specification for a generic parallel branch-and-bound search to find minimum-cost solution.
Asserts that different runs may return different optimal solutions that that are semantically equivalent for equivalent inputs.

  - **Not a complete spec of parallel correctness!** For complex programs, determinism proof attempts get entangled in details of sequential correctness.
- **Our proposal:** For a parallel program, use a **sequential** but **nondeterministic (NDSeq)** version as the intermediate specification.
  - **Lightweight annotations:** Specify intended nondeterminism without referring to an explicit functional specification.

OUR APPROACH




```
coforeach i in [1..N]
  b = lower_bound(i)
  if b >= lowest_cost
    continue // skip search
  s = search(i)
  synchronized_by(lock)
    if cost(s) < lowest_cost
      lowest_cost = cost(s)
      best_soln = s
```

Figure 2. Generic parallel branch-and-bound search.

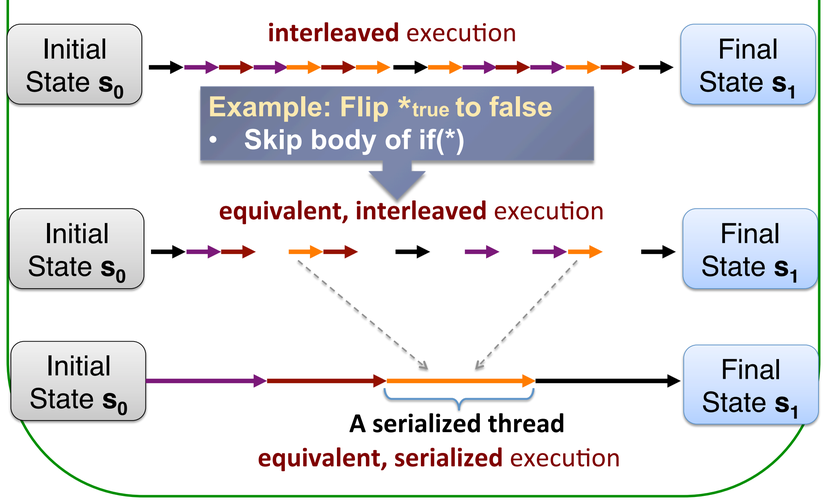
Programmer annotates

```
nd-coforeach in [1..N]
  b = lower_bound(i)
  if (*true)
    if b >= lowest_cost
      continue // skip search
  s = search(i)
  synchronized_by(lock)
    if cost(s) < lowest_cost
      lowest_cost = cost(s)
      best_soln = s
```

Figure 3. Parallel program with its NDSEq specification.

Runtime Checking Parallelism Correctness

- **Our technique:** Improves traditional conflict serializability by flipping default value of *



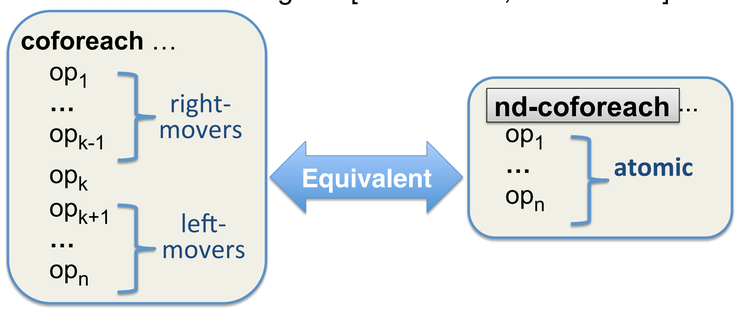
EXPERIMENTAL EVALUATION

- **Implementation for Java:** In Calfuzzer active testing framework
- **Results (Ease of use & precision):**
  1. **Ease of use:** Feasible to write NDSeq specifications for real Java benchmarks.
  2. **Precision:** Our technique produces significantly fewer false positives than a traditional conflict-serializability.

Benchmark	LoC	Num. of parallel constructs	Num. of if(*) and focus annotations		Size of trace All	Size of trace Relevant	Distinct warning Naive	Distinct warning With slicing
series	800	1	0	5	11k	140	0	0
crypt	1.1K	2	0	3	504K	236	0	0
raytracer	1.9K	1	0	1	6170K	44K	1	1 (real bug)
montecarlo	3.6K	1	0	1	1897K	534K	2	0
pi3	150	1	0	1	1062K	141	0	0
keysearch3	200	2	0	4	2059K	91K	0	0
mandelbrot	250	1	0	6	1707K	954	0	0
phylogeny	4.4K	2	3	8	470K	5K	6	6 (real bug)
stack	40	1	2	8	1744	536	5	0
queue	60	1	2	8	868	229	9	0
meshrefine	1K	1	2	50	747K	302K	30	0

FUTURE WORK

- **What is the process (manual and tool support) for writing NDSeq specifications?**
  - Propose patterns for guiding introduction of if(*)'s.
  - How can tools help the user when writing the NDSeq spec.? Can we automatically infer NDSeq spec.?
- **Proofs by reduction**: Statically prove that operations are atomic by reduction [Lipton, CACM 1974].
  - Use SMT solving like [Elmas et al., POPL 2009].



- **Applications to debugging?**
  - Allow programmer to sequentially debug a parallel execution by mapping a parallel trace to a nondeterministic sequential one.