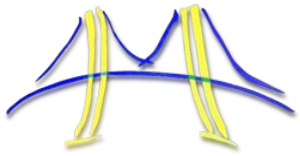
EECS
Electrical Engineering and Computer Sciences



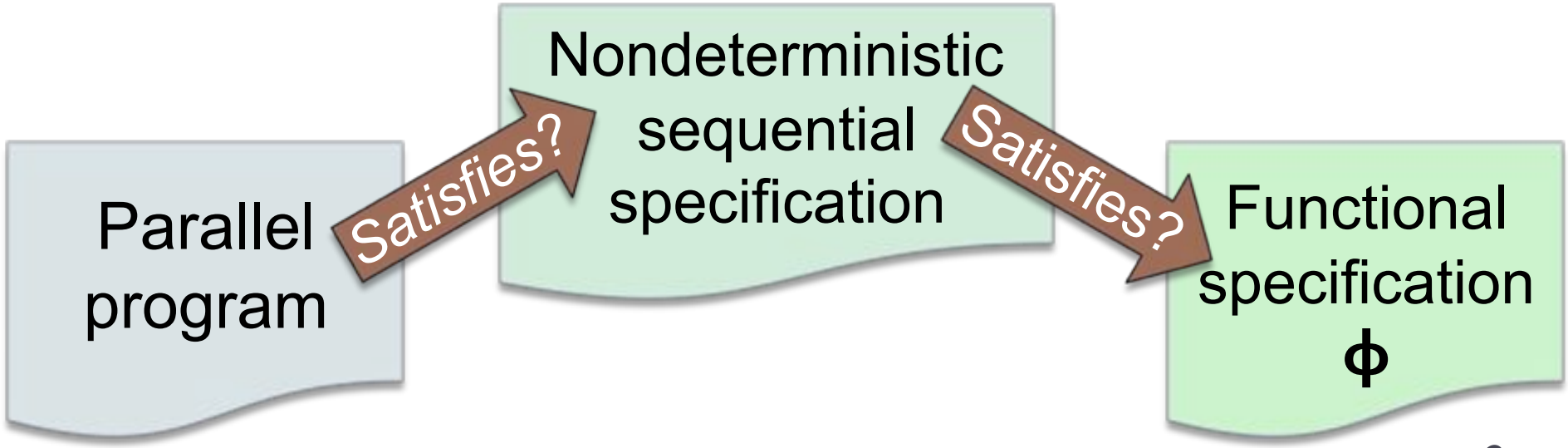
PARALLEL COMPUTING LABORATORY

NDSeq: Runtime Checking for Nondeterministic Sequential Specs of Parallel Correctness

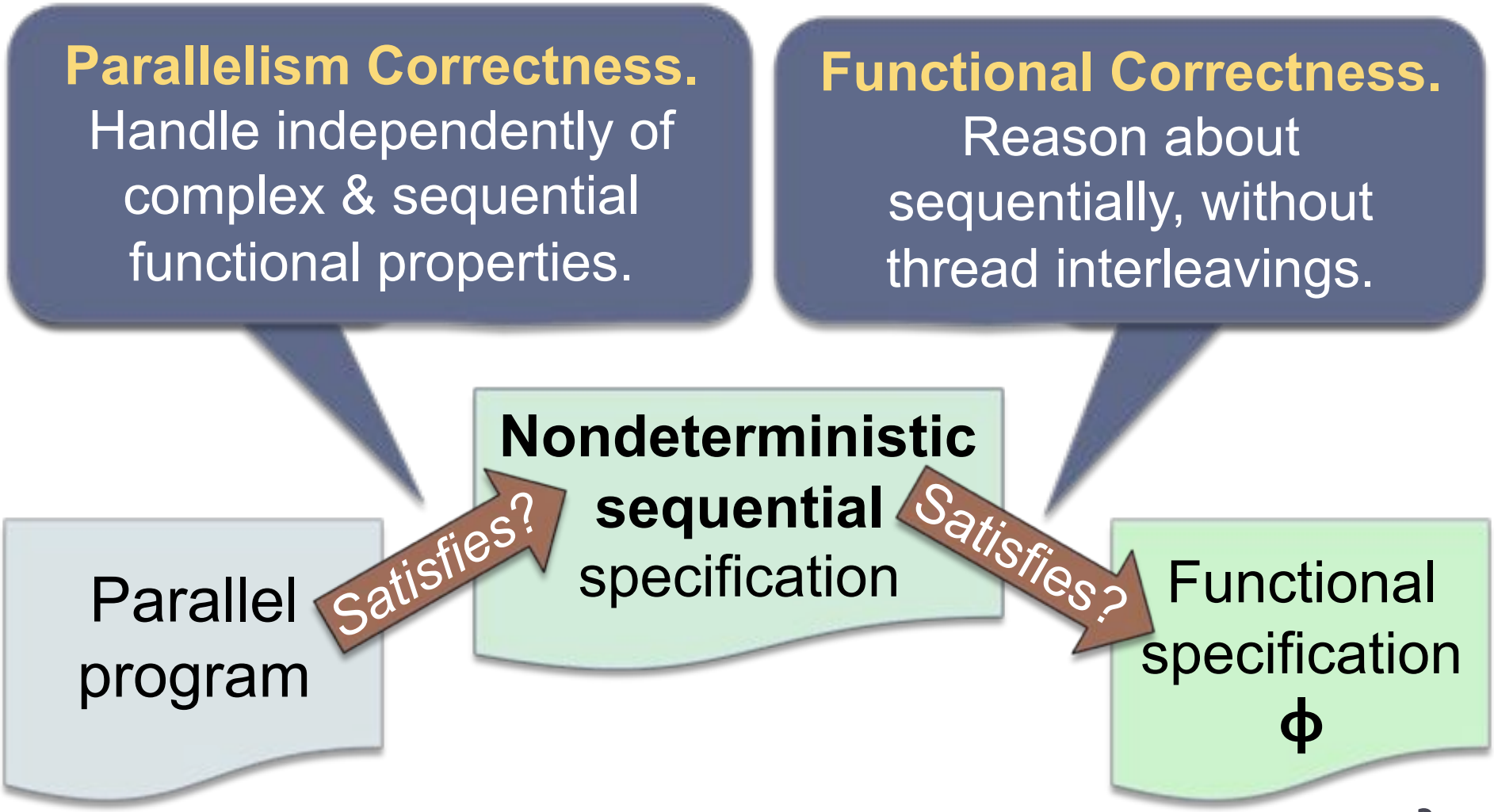
Jacob Burnim, Tayfun Elmas,
George Necula, Koushik Sen

University of California, Berkeley

Goal: Decompose effort in addressing parallelism and functional correctness



Goal: Decompose effort in addressing parallelism and functional correctness



Goal: Decompose effort in addressing parallelism and functional correctness

1. **NDSeq: easy-to-write spec for parallelism.**
2. **Runtime checking of NDSeq specifications.**

Outline

- Overview
- **Motivating Example**
- Nondeterministic Sequential (NDSeq) Specifications for Parallel Correctness
- Runtime Checking of NDSeq Specifications
- Experimental Results
- Conclusion

Motivating Example

- **Goal:** Find minimum-cost item in list.

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Input: N items.

Output: min_cost and min_item.

Motivating Example

- **Goal:** Find minimum-cost item in list.

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Computes **cheap** lower bound on cost of **i**.

Prune when **i** cannot have minimum-cost.

Computes cost of item **i**. **Expensive.**

Motivating Example

- **Goal:** Find minimum-cost item in list.

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  if cost < min_cost:
    min_cost = cost
    min_item = i
```

How do we parallelize this code?

Parallel Motivating Example

- **Goal:** Find min-cost item in list, **in parallel**.

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Loop iterations can be run in **parallel.**

Updates to best are **protected by lock.**

Parallel Motivating Example

- **Goal:** Find min-cost item in list, **in parallel**.

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Claim: Parallelization is clearly correct.

How can we specify this parallel correctness?

Specifying Parallel Correctness

- **Idea**: Use sequential program as spec.

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Satisfies? **No.**

```
for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)

  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Parallel-Sequential Equivalence?

items: (1) bound: 5 cost: 5 (2) bound: 5 cost: 5

min_item: –
min_cost: ∞

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

prune?(1)

Parallel-Sequential Equivalence?

items: (1) bound: 5 cost: 5 (2) bound: 5 cost: 5

min_item: –
min_cost: ∞

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

prune?(1)

prune?(2)

Parallel-Sequential Equivalence?

items: (1) bound: 5 cost: 5 | (2) bound: 5 cost: 5

min_item: **(2)**
min_cost: **5**

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

prune?(**1**)

prune?(**2**)

update(**2**)

Parallel-Sequential Equivalence?

items: (1) bound: 5 cost: 5 | (2) bound: 5 cost: 5

min_item: **(2)**
min_cost: **5**

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

prune?(**1**)

prune?(**2**)

update(**2**)

update(**1**)

Parallel-Sequential Equivalence?

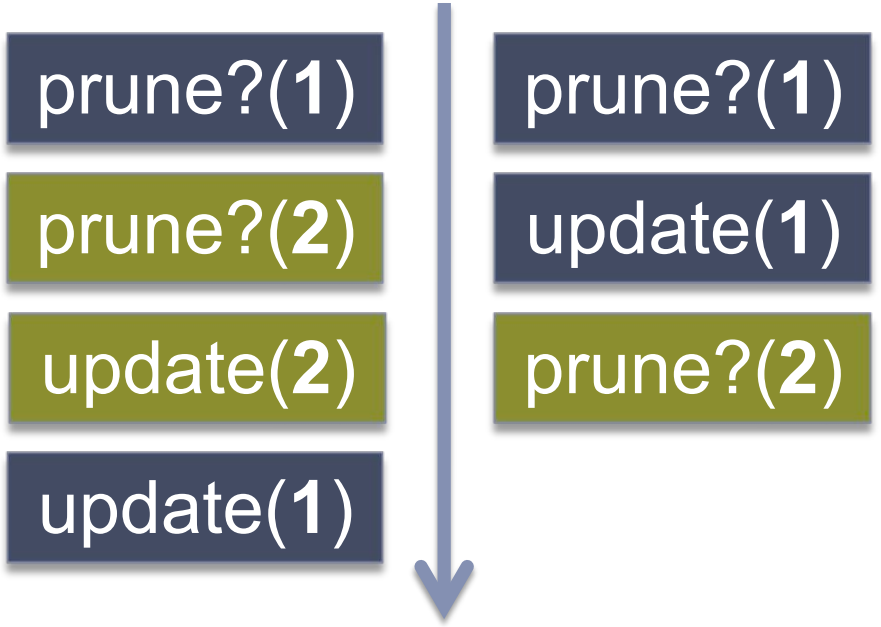
items: (1) bound: 5 cost: 5 | (2) bound: 5 cost: 5

min_item: **(2)**
min_cost: **5**

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

But sequential program:

- Returns min_item = **(1)**.
- Prunes (2).



Specifying Parallel Correctness

- Parallel program has freedom to:

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Process items in a **nondeterministic** order.

Avoid pruning by scheduling check before updates.

Specifying Parallel Correctness

Must give sequential spec this freedom.

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Process items in a **nondeterministic** order.

Avoid pruning by scheduling check before updates.

Nondeterministic Sequential Spec

Runs iterations **in any order.**

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
    min_cost = cost
    min_item = i
```

Can **choose not** to prune item.

```
nd-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if * && b >= c:
    continue
  cost = compute_cost(i)

  if cost < min_cost:
    min_cost = cost
    min_item = i
```

NDSeq Specification Patterns

- Found three recipes for adding *'s:
  1. Optimistic Concurrent Computation (optimistic work with conflict detection)
  2. Redundant Computation Optimization (e.g., pruning in branch-and-bound)
  3. Irrelevant Computation (e.g., updating a performance counter)
- With these recipes, fairly simple to write NDSeq specifications for our benchmarks.

Nondeterministic Sequential Spec

- Parallelism correct if no more nondeterminism:

```
parallel-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if b >= c:
    continue
  cost = compute_cost(i)
  synchronized (lock):
    if cost < min_cost:
      min_cost = cost
      min_item = i
```

Yes.

Satisfies?

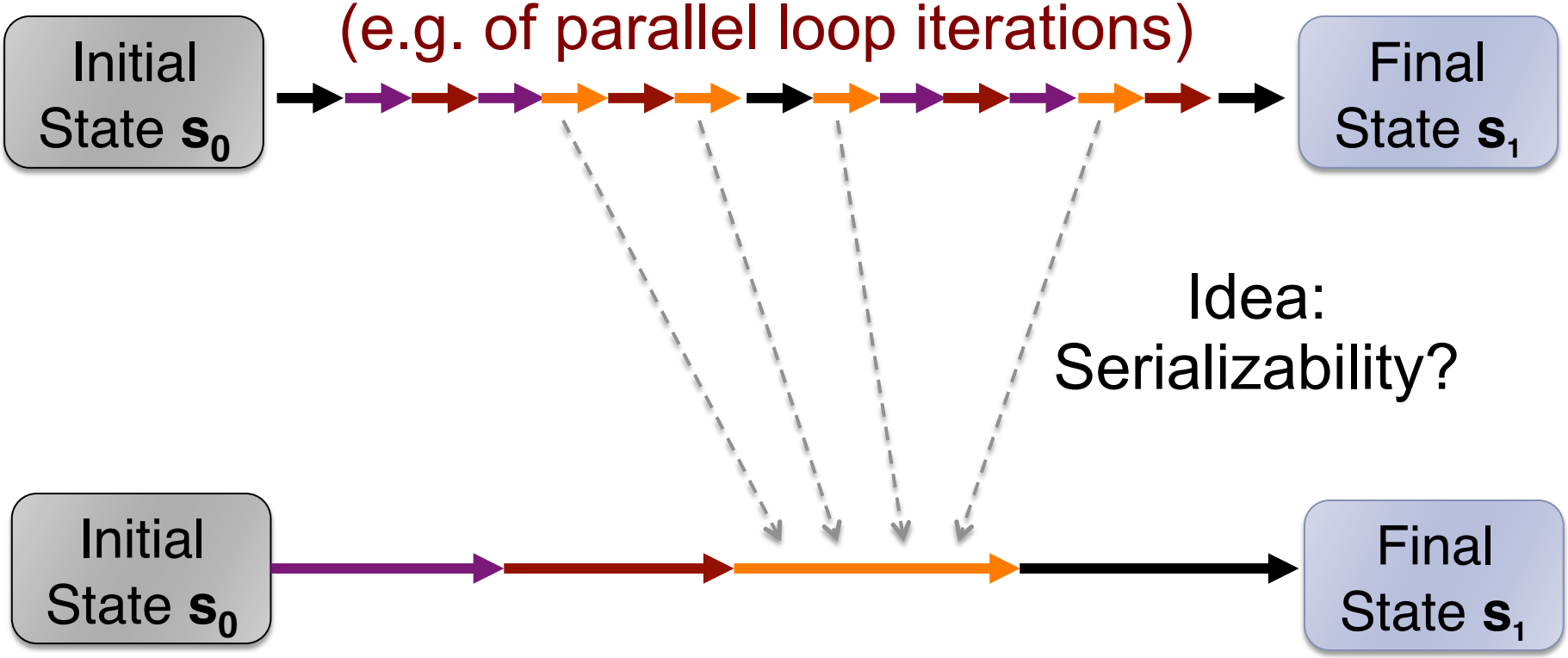
```
nd-for (i in [1..N]):
  c = min_cost
  b = lower_bound(i)
  if * && b >= c:
    continue
  cost = compute_cost(i)

  if cost < min_cost:
    min_cost = cost
    min_item = i
```

Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSeq) Specifications for Parallel Correctness
- **Runtime Checking of NDSeq Specs**
- Experimental Results
- Conclusion

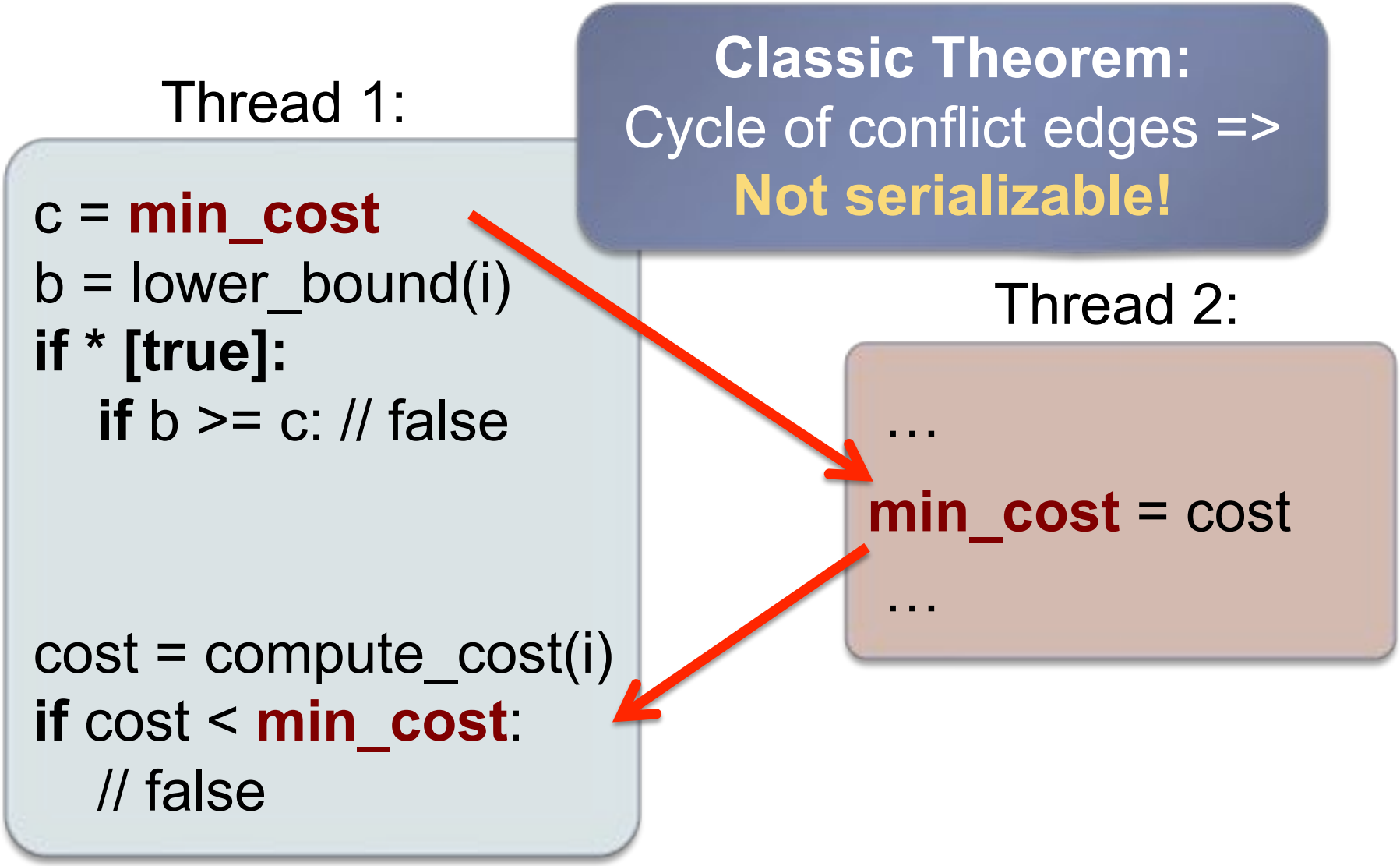
Testing Parallelism Correctness

Given: an execution of parallel program (e.g. of parallel loop iterations)

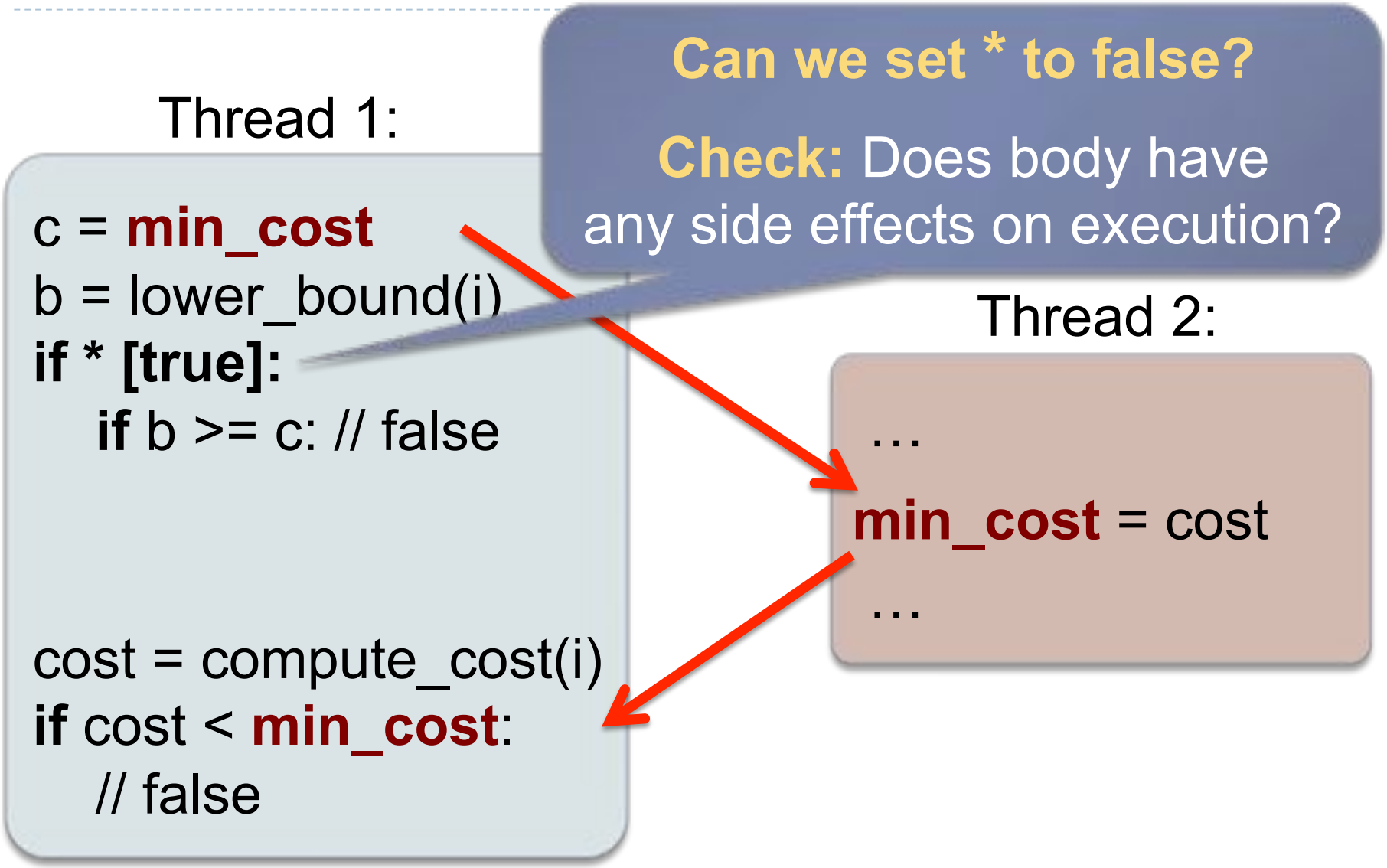


Is there an **equivalent** execution of NDSeq spec?

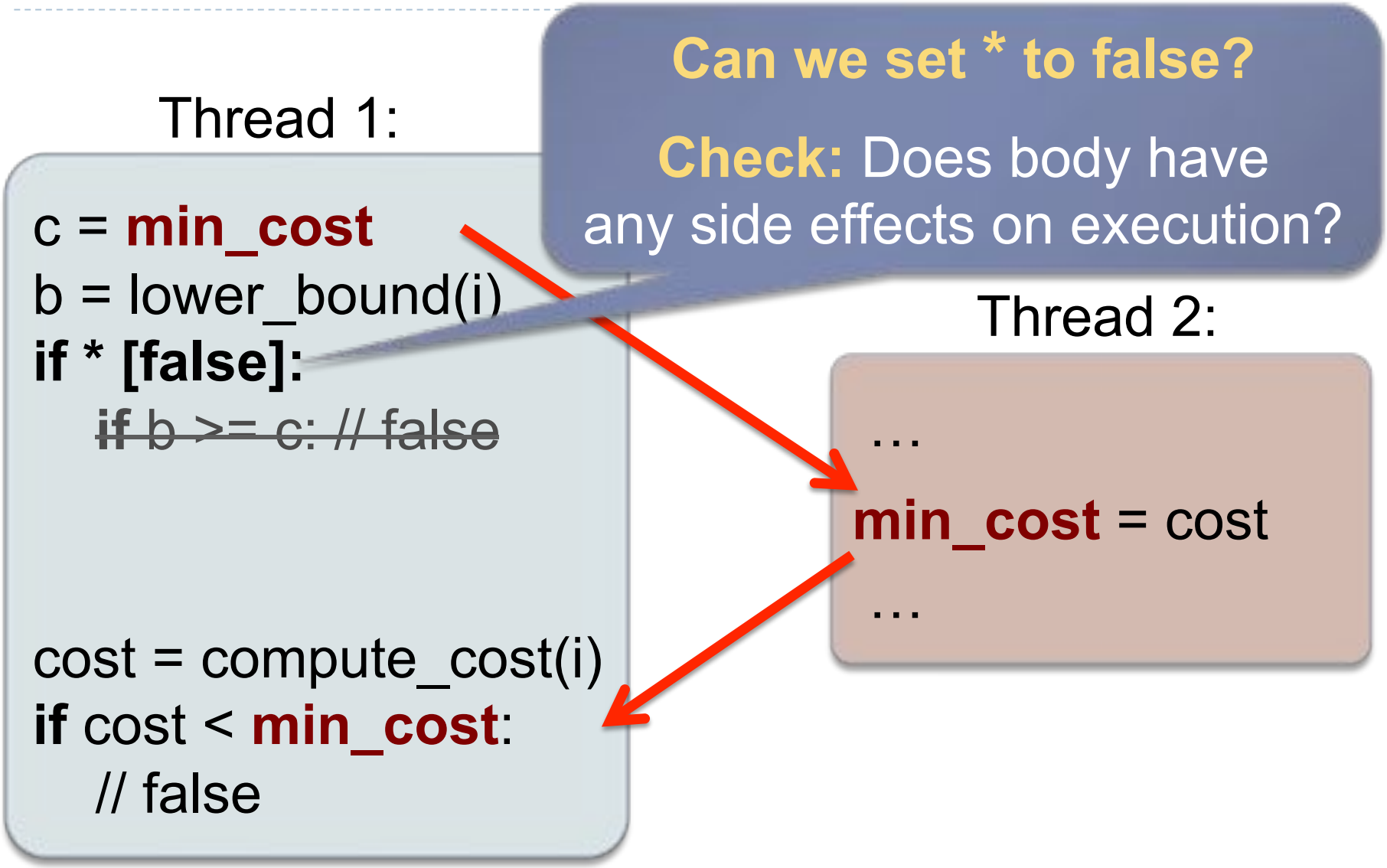
Conflict-Serializability is Too Strict



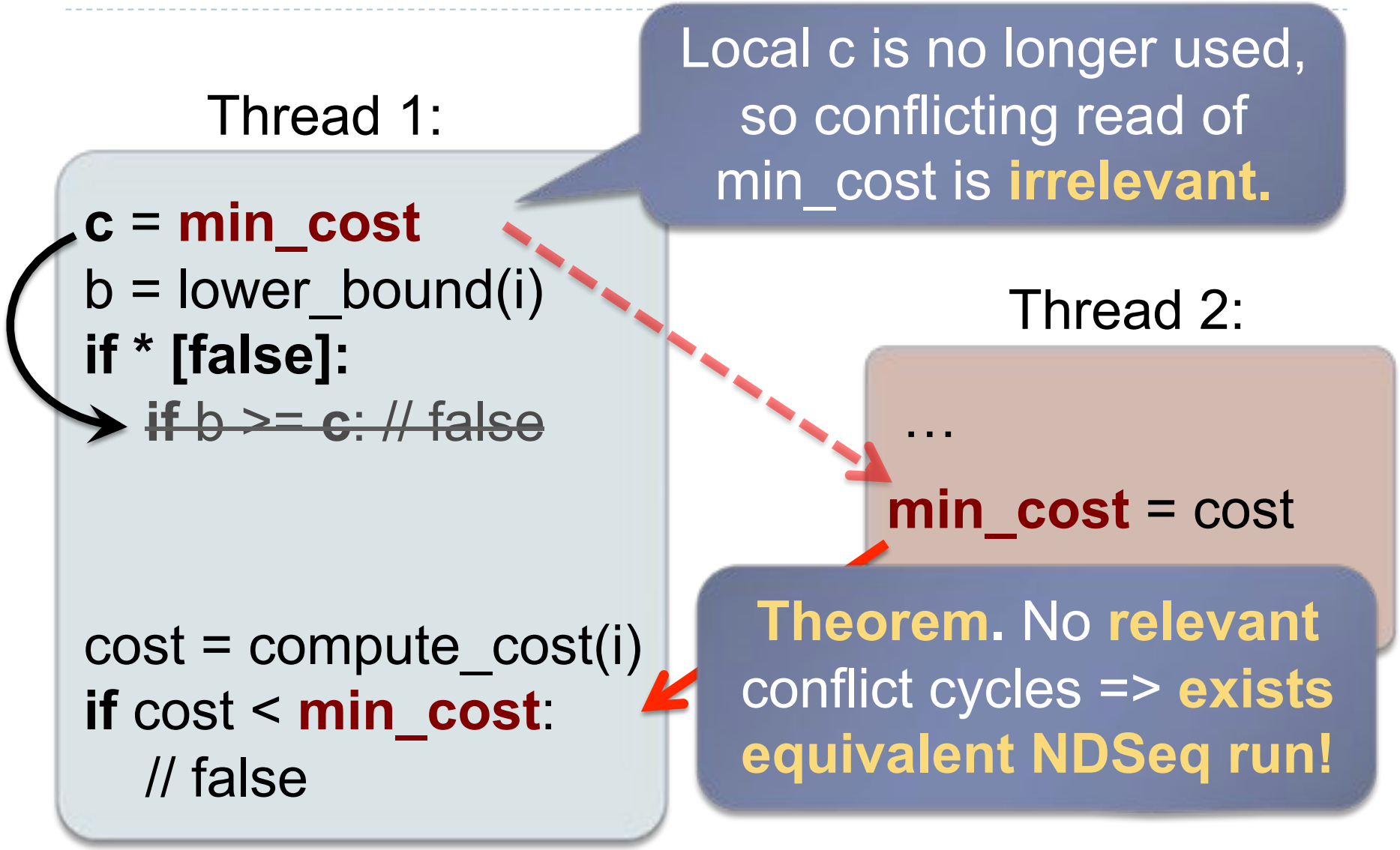
Relaxing Conflict-Serializability



Relaxing Conflict-Serializability



Relaxing Conflict-Serializability



Relaxing Conflict-Serializability

Theorem. No **relevant** conflict cycles => **exists equivalent NDSeq run!**

Iteration 2:

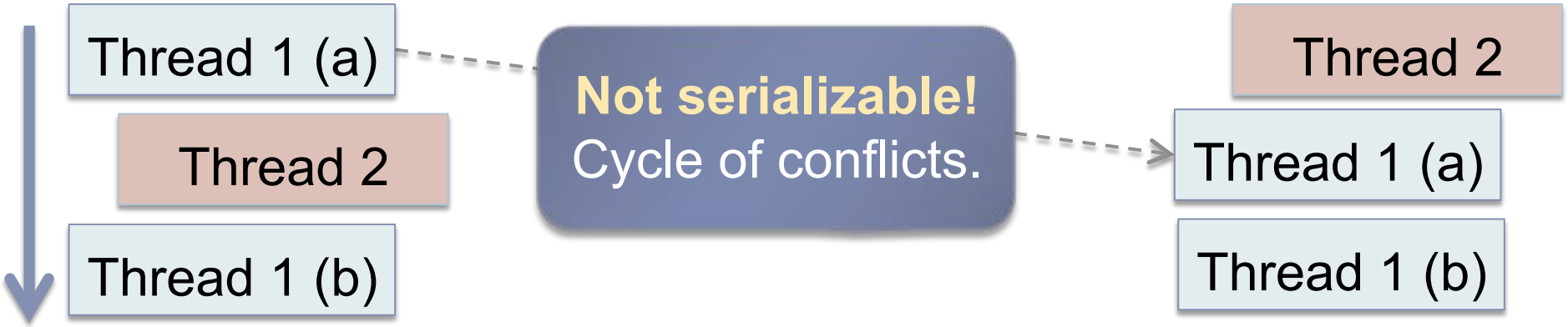
```
…
min_cost = cost
…
```

Iteration 1:

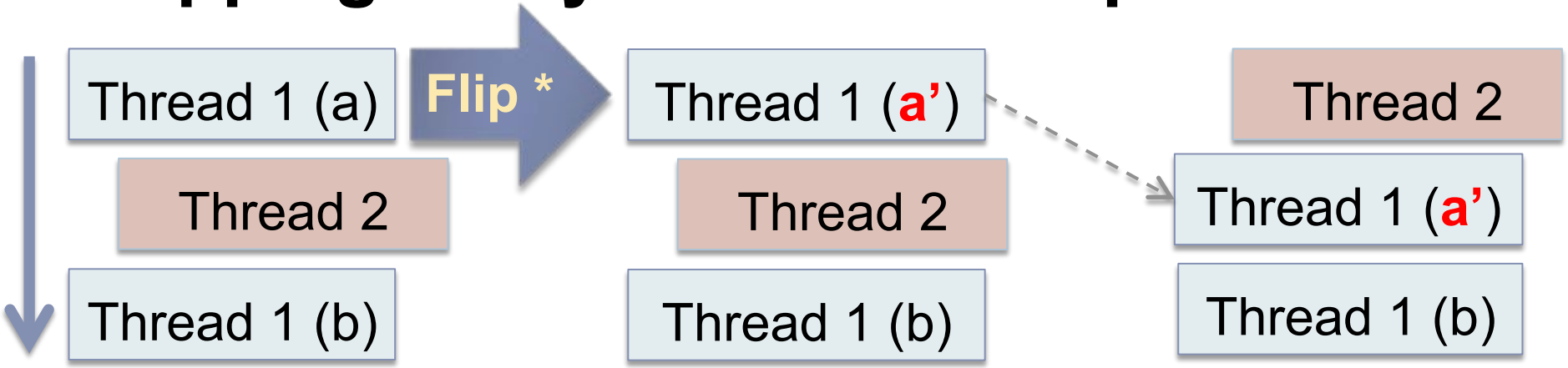
```
c = min_cost
b = lower_bound(i)
if * [false]:


cost = compute_cost(i)
if cost < min_cost:
    // false
```

Read **different** value for min_cost, but **overall** behavior is the same.

Traditional conflict serializability:



+ flipping * + dynamic data dependence:



Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSeq) Specifications for Parallel Correctness
- Runtime Checking of NDSeq Specifications
- **Experimental Results**
- Conclusion

Experimental Evaluation

- Wrote and tested NDSeq specifications for:
  - Java Grande, Parallel Java, Lonestar, DaCapo, and nonblocking data structure.
  - **Size:** 40 to 300K lines of code.
  - Tested 5 parallel executions / benchmark.
- **Two claims:**
  1. Easy to write NDSeq specifications.
  2. Our technique serializes significantly more executions than traditional methods.

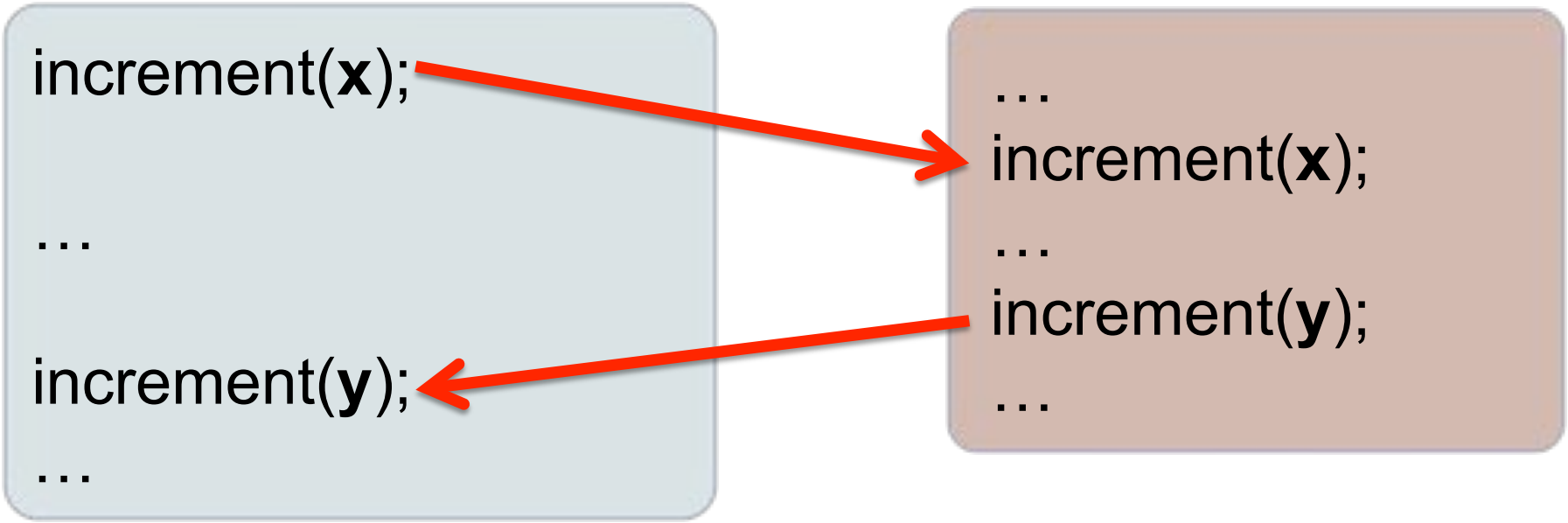
	Benchmark	Lines of Code	# of Parallel Constructs	# of if(*)
	stack	40	**1**	**2**
	queue	60	**1**	**2**
	meshrefine	1K	**1**	**2**
DaCapo	sunflow	24K	**4**	**4**
DaCapo	xalan	302K	**1**	**3**
PJ	keysearch3	200	**2**	0
PJ	mandelbrot	250	**1**	0
PJ	phylogeny	4.4K	**2**	**3**
JGF	series	800	**1**	0
JGF	crypt	1.1K	**2**	0
JGF	raytracer	1.9K	**1**	0
JGF	montecarlo	3.6K	**1**	0

	Benchmark	Size of Trace	Serializability Warnings	
			Traditional	Our Technique
	stack	1,744	**5 (false)**	0
	queue	846	**9 (false)**	0
	meshrefine	747K	**30 (false)**	0
DaCapo	sunflow	24,250K	**28 (false)**	**3 (false)**
DaCapo	xalan	16,540K	**6 (false)**	**2 (false)**
PJ	keysearch3	2,059K	**2 (false)**	0
PJ	mandelbrot	1,707K	**1 (false)**	0
PJ	phylogeny	470K	**6**	**6**
JGF	series	11K	0	0
JGF	crypt	504K	0	0
JGF	raytracer	6,170K	**1**	**1**
JGF	montecarlo	1,897K	**2 (false)**	0

	Benchmark	Size of Trace	Serializability Warnings	
			Traditional	Our Technique
	stack	1,744	**5 (false)**	0
	queue	846	**9 (false)**	0
	meshrefine	747K	**30 (false)**	0
DaCapo	sunflow	24,250K	**28 (false)**	**3 (false)**
DaCapo	xalan	16,540K	**6 (false)**	**2 (false)**
PJ	keysearch3	2,059K	**2 (false)**	0
PJ	mandelbrot	1,707K	**1 (false)**	0
PJ	phylogeny	470K	**6**	**6**
JGF	series	11K	0	0
JGF	crypt	504K	0	0
JGF	raytracer	6,170K	**1**	**1**
JGF	montecarlo	1,897K	**2 (false)**	0

Limitations

- Implementation
  - Dynamic data dependence ==> high overhead.
  - Instrumentation may miss some reads/writes.
- Commutativity:



Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSeq) Specifications for Parallel Correctness
- Runtime Checking of NDSeq Specifications
- Experimental Results
- **Conclusion**

Summary

- Separate parallel & functional correctness.
  - Lightweight NDSeq specs for parallelism.
  - Sequentially verify functional correctness.
- Runtime checking of NDSeq specs.
  - Generalize conflict-serializability using if(*) and dynamic data dependence.
- Future/Current Work:
  - Automatically inferring NDSeq specifications.
  - Static verification of parallel correctness.
  - Debugging on NDSeq.

Questions?

Many thanks to Intel, Microsoft, other Parlab sponsors,
and NSF for supporting this work.