NDSeq: Specifying and Checking Parallelism Correctness Using Nondeterministic Sequential Programs

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

Our challenge: Simplify testing and verification of parallel programs

- **Parallel:** More difficult than sequential
  - – Simultaneous reasoning about functional correctness and nondeterministic thread interleavings

Our goal: Decompose efforts in addressing parallelism and functional correctness

Parallel program

Satisfies?

Functional specification

Parallel program

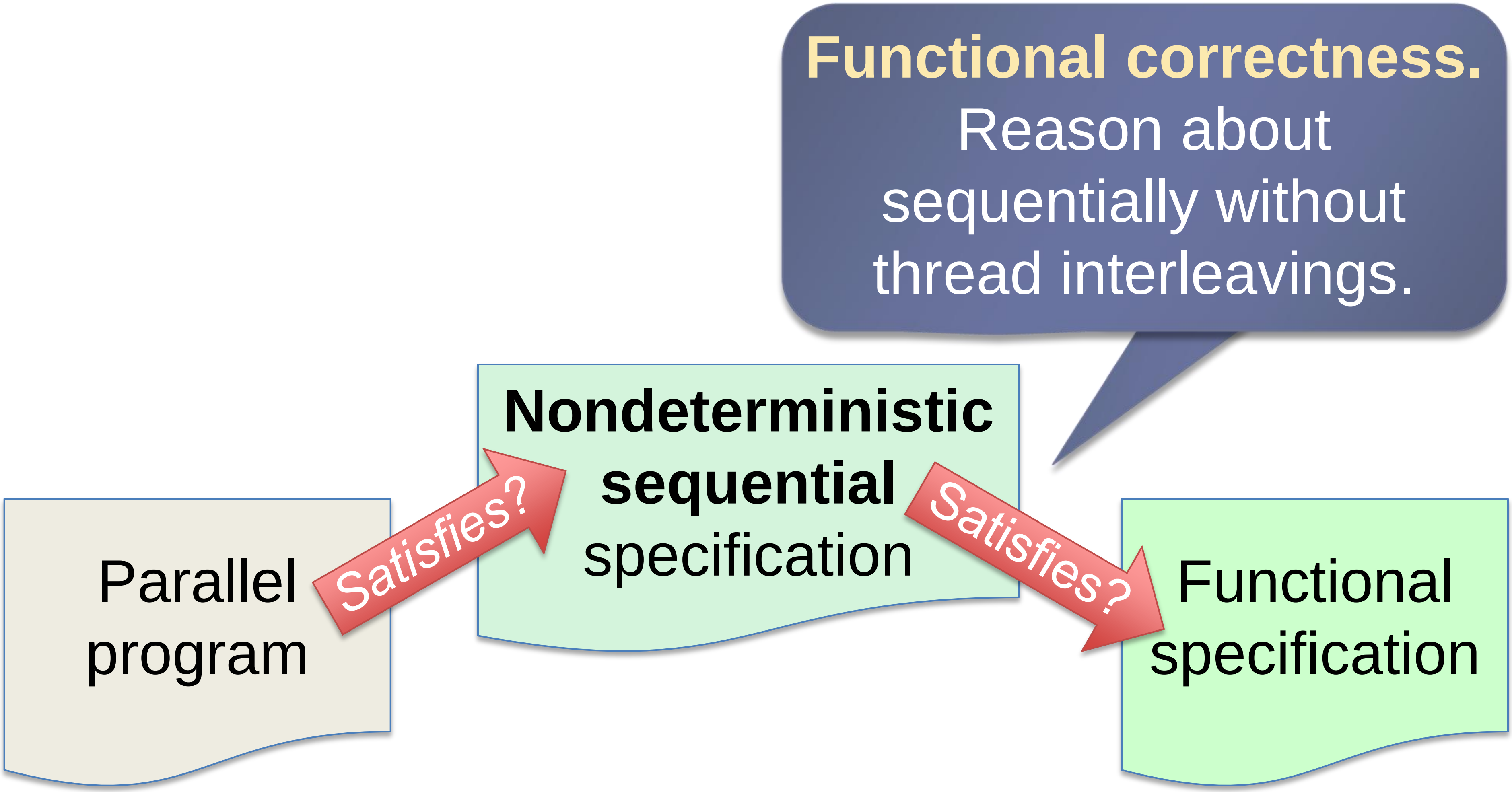
Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification

Our goal: Decompose efforts in addressing parallelism and functional correctness



Functional correctness. Reason about sequentially without thread interleavings.

Parallel program

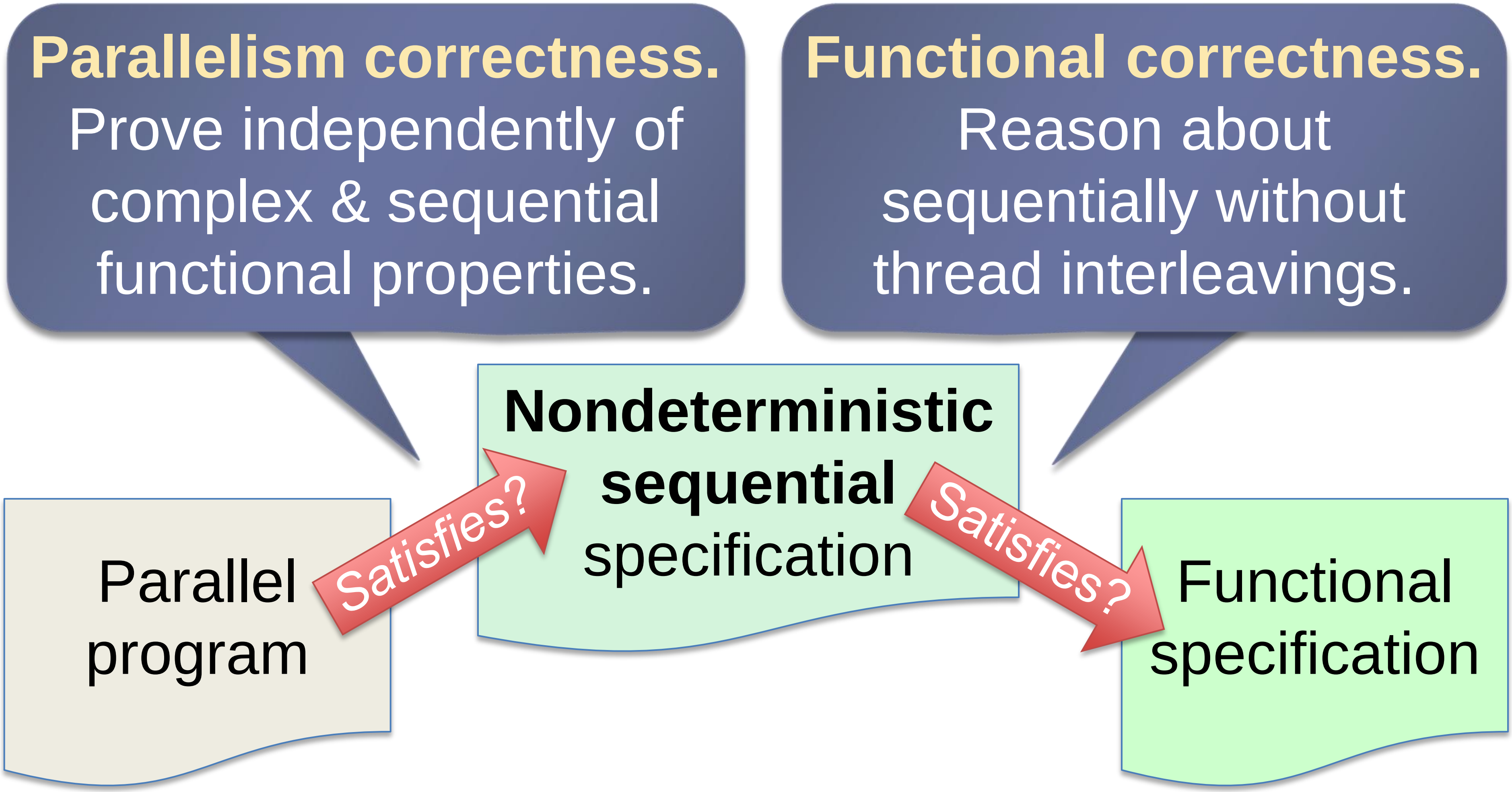
Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification

Our goal: Decompose efforts in addressing parallelism and functional correctness



Parallelism correctness. Prove independently of complex & sequential functional properties.

Functional correctness. Reason about sequentially without thread interleavings.

Parallel program

Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification

Our approach: Nondeterministic sequential (NDSeq) specifications

In this talk (on a running example):

1. Easy-to-write, lightweight specification

- Few simple annotations to indicate **intended nondeterminism** (nd-foreach, if(*))

Parallel program

Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification

Our approach: Nondeterministic sequential (NDSeq) specifications

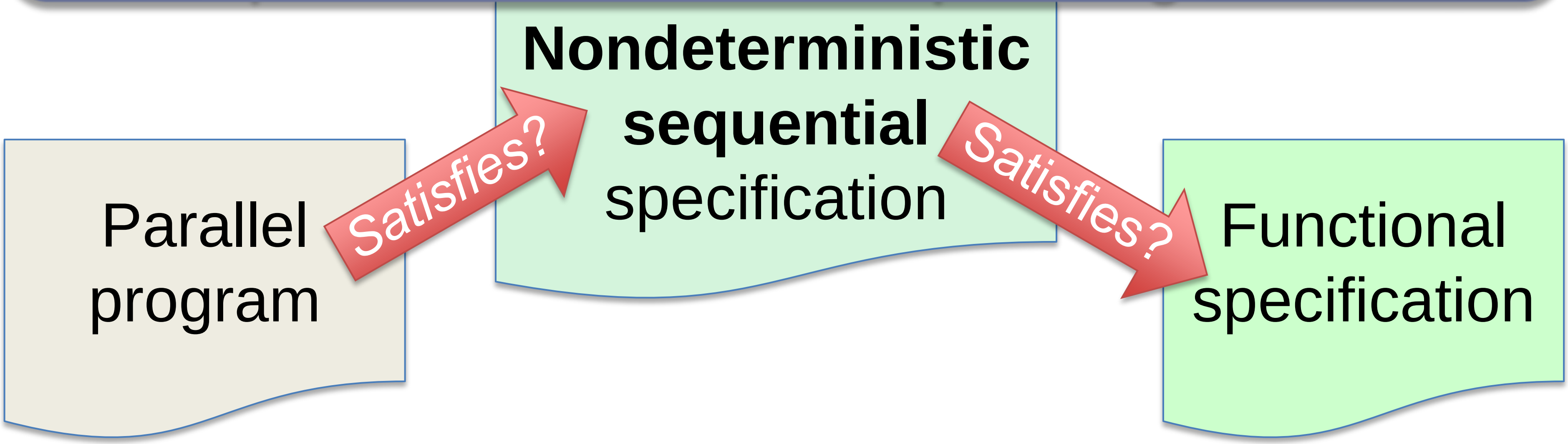
In this talk (on a running example):

1. Easy-to-write, lightweight specification

- Few simple annotations to indicate **intended nondeterminism** (nd-foreach, if(*))

2. Runtime checking algorithm for testing

- Improves traditional technique using annotations



Parallel program

Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification

Example: Simple branch-and-bound

Goal: Find minimum-cost solution

Initially: lowest_cost = ∞

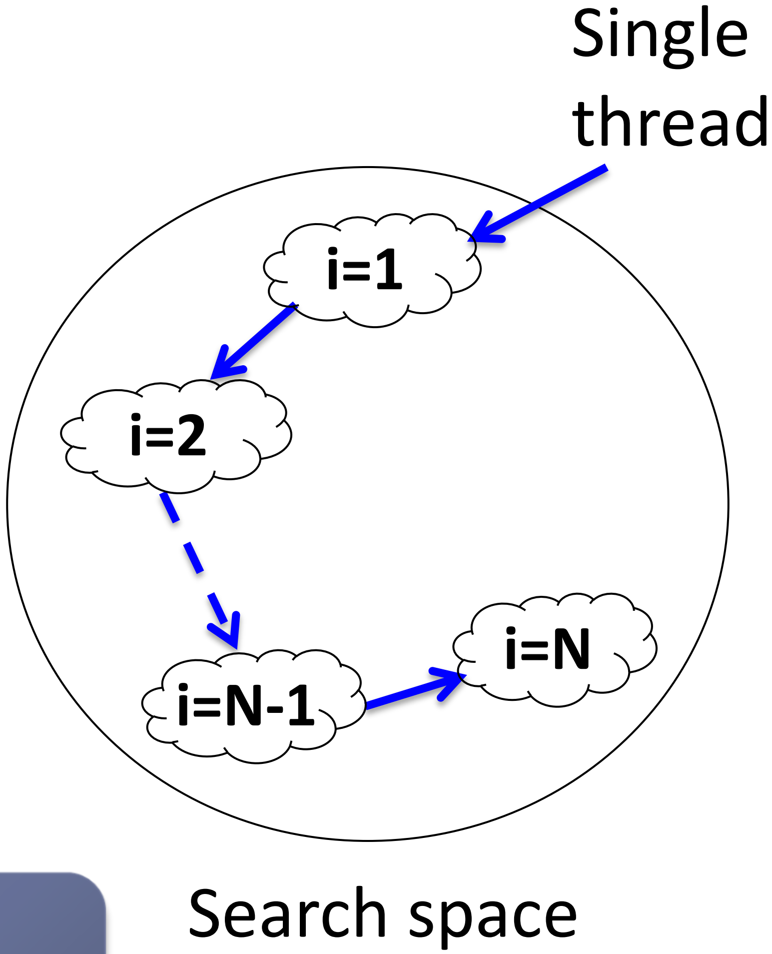
Sequential program:

```
foreach i in [1..N]

    s = search(i)

    if cost(s) < lowest_cost
        lowest_cost = cost(s)
        best_soln = s
```

Outputs:

Best solution: **best_soln**

Minimum cost: **lowest_cost**



Example: Simple branch-and-bound

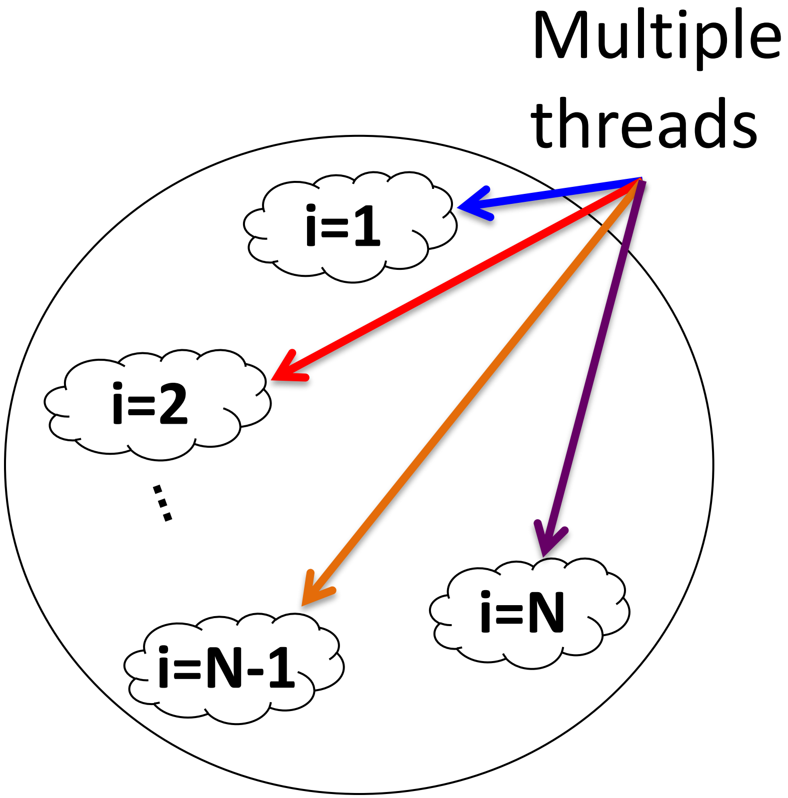
Goal: Find minimum-cost solution

Initially: lowest_cost = ∞

Parallel program:

```
coforeach i in [1..N]

    s = search(i)

    synchronized_by(lock)
        if cost(s) < lowest_cost
            lowest_cost = cost(s)
            best_soln = s
```

lowest_cost and **best_soln** are global to all threads



Search space

Example: Simple branch-and-bound

Goal: Find minimum-cost solution

Initially: lowest_cost = ∞

Parallel program:

```
coforeach i in [1..N]

  s = search(i)

  synchronized_by(lock)
    if cost(s) < lowest_cost
      lowest_cost = cost(s)
      best_soln = s
```

Functional correctness:
As difficult to prove
as sequential.

PLUS thread interleavings.

assert (cost(best_soln) is lowest_cost and minimum in search space)

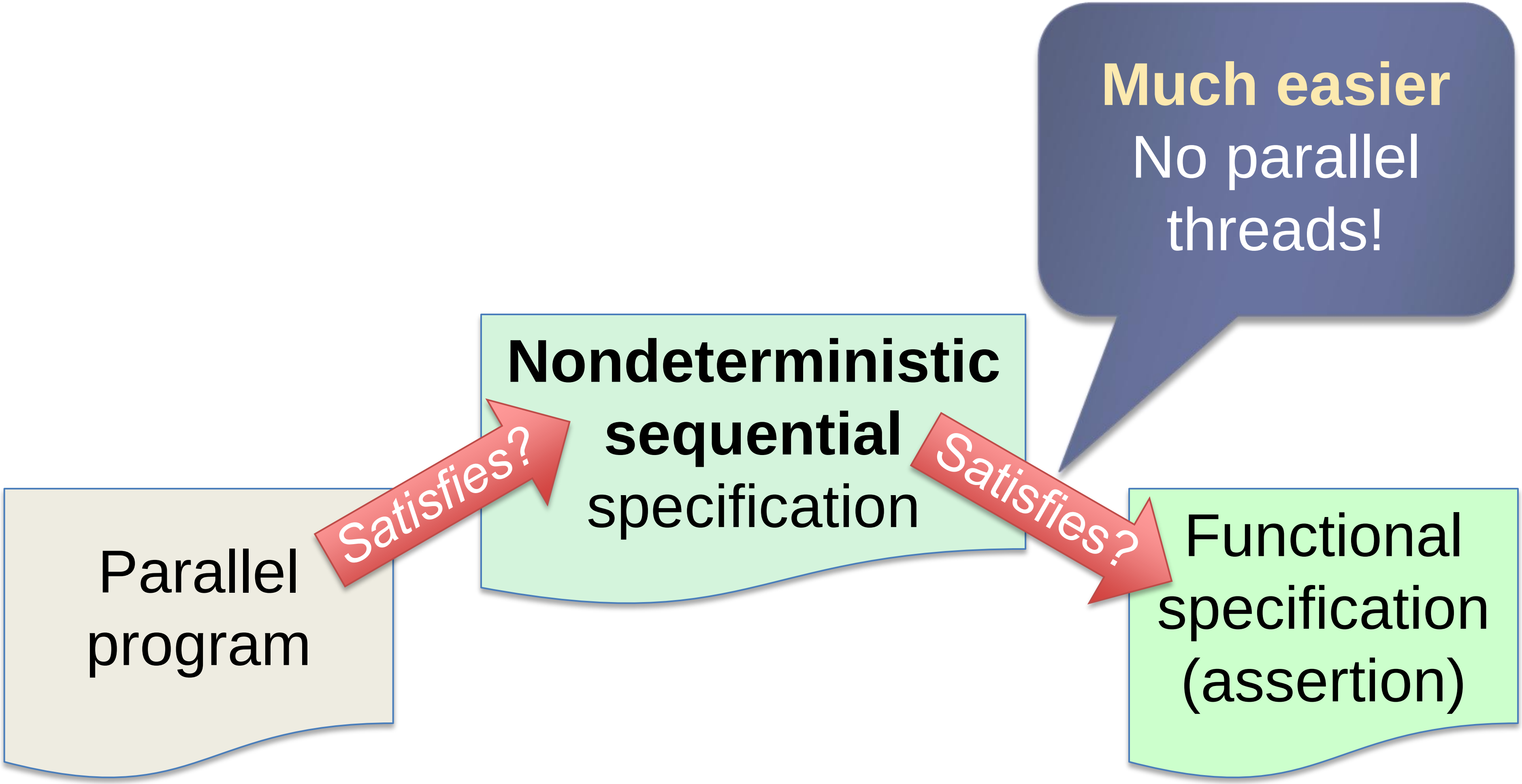
Our goal

Parallel program

Satisfies?

Functional specification (assertion)

Our approach



Parallel program

Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification (assertion)

Much easier

No parallel threads!

Our approach

Correct parallelism: For each interleaved behavior of parallel program, exists a sequential behavior of NDSeq specification giving the same result.

Independently of functional correctness.

Parallel program

Nondeterministic sequential specification

Functional specification (assertion)

Our approach

Correct parallelism: For each interleaved behavior of parallel program, exists a sequential behavior of NDSeq specification giving the same result.

Independently of functional correctness.

Parallel program:

```
coforeach i in [1..N]

  s = search(i)

  synchronized_by(lock)
    if cost(s) < lowest_cost
      lowest_cost = cost(s)
      best_soln = s
```

Satisfies?

NDSeq specification:

```
foreach i in [1..N]

  s = search(i)


    if cost(s) < lowest_cost
      lowest_cost = cost(s)
      best_soln = s
```

Parallel program = NDSeq spec ?

Initially:
lowest_cost = ∞

Search space:

(1) cost(s1): 5

(2) cost(s2): 5

Parallel program = NDSeq spec ?

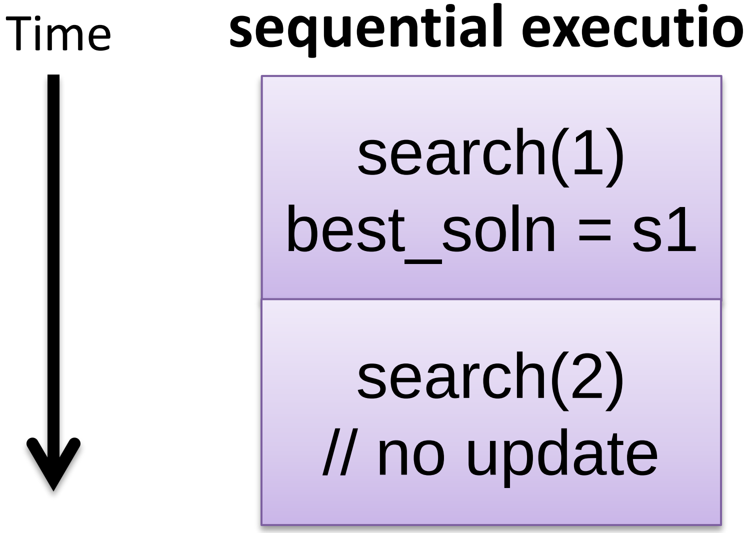
Initially:
lowest_cost = ∞

Search space:

(1) cost(s1): 5

(2) cost(s2): 5

Only possible sequential execution


Time

search(1)
best_soln = s1

search(2)
// no update

Result: best_soln = s1

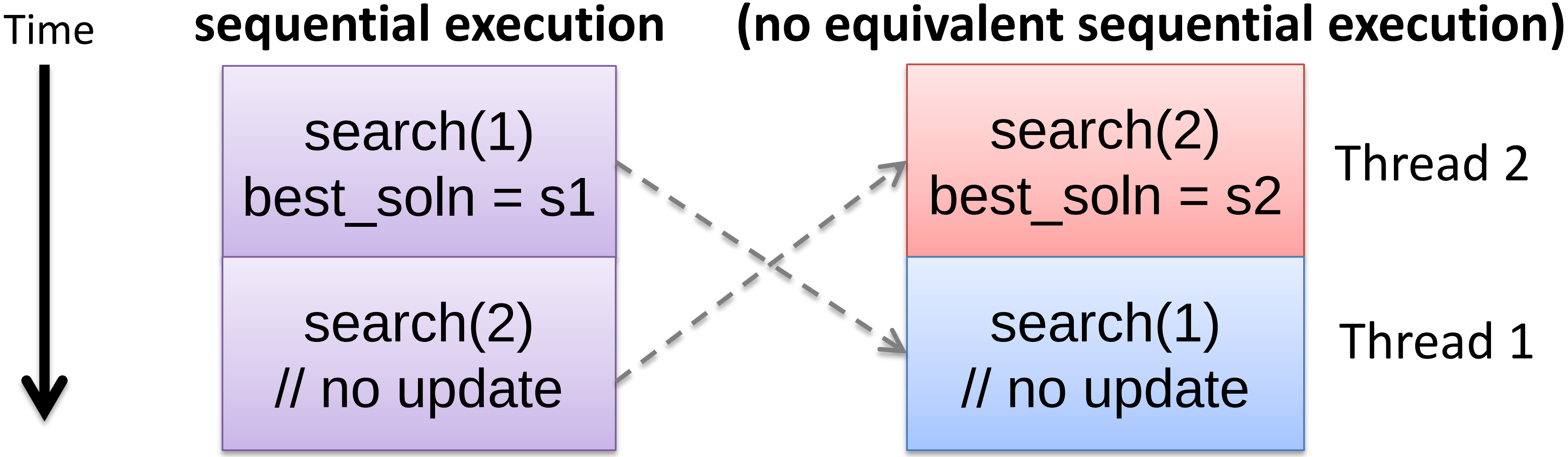
Parallel program = NDSeq spec ?

Initially:
lowest_cost = ∞

Search space:

(1) cost(s1): 5

(2) cost(s2): 5



Only possible sequential execution

A parallel execution (no equivalent sequential execution)

Time

search(1)
best_soln = s1

search(2)
// no update

search(2)
best_soln = s2

Thread 2

search(1)
// no update

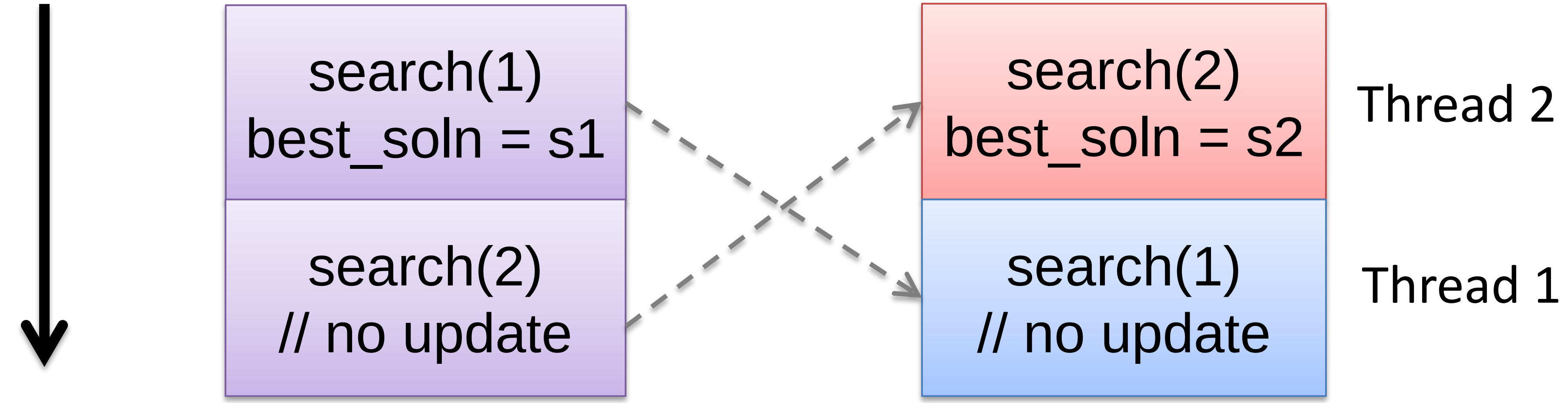
Thread 1

Result: best_soln = s1 $\neq$ **Result:** best_soln = s2

Parallel program ≠ NDSeq spec

NDSeq specification too strict !

- Must allow to choose different optimal solutions



Only possible sequential execution

A parallel execution (no equivalent sequential execution)

Time

search(1)
best_soln = s1

search(2)
// no update

search(2)
best_soln = s2

Thread 2

search(1)
// no update

Thread 1

Result: best_soln = s1

Result: best_soln = s2

Introducing nondeterminism sequentially

Programmer annotates loop:

- Allow sequential code to perform iterations in nondeterministic order.

NDSeq specification:

```
foreach i in [1..N]

    s = search(i)

    if cost(s) < lowest_cost
        lowest_cost = cost(s)
        best_soln = s
```

New NDSeq specification

```
nd-foreach i in [1..N]

    s = search(i)

    if cost(s) < lowest_cost
        lowest_cost = cost(s)
        best_soln = s
```

Parallel program = NDSeq spec

NDSeq specification gives same results as parallel program!

- With only one thread!

Parallel program:

```
coforeach i in [1..N
    s = search(i)

    synchronized_by(lock)
        if cost(s) < lowest_cost
            lowest_cost = cost(s)
            best_soln = s
```

Satisfies!

NDSeq specification:

```
nd-foreach i in [1..N]
    s = search(i)


    if cost(s) < lowest_cost
        lowest_cost = cost(s)
        best_soln = s
```

Example with optimization code

Parallel program:

```
coforeach i in [1..N]
    b = lower_bound(i)
    if b >= lowest_cost
        end_iteration
    s = search(i)

    synchronized_by(lock)
        if cost(s) < lowest_cost
            lowest_cost = cost(s)
            best_soln = s
```

Cheap !
Prune if search is redundant

Expensive !

Parallel program = NDSeq spec

Parallel program:

```
coforeach i in [1..

    b = lower_bound(i)
    if b >= lowest_cost
        end_iteration

    s = search(i)

    synchronized_by(lock)
        if cost(s) < lowest_cost
            lowest_cost = cost(s)
            best_soln = s
```

Satisfies!

NDSeq specification:

```
nd-foreach i in [1..N]

    b = lower_bound(i)
    if b >= lowest_cost
        end_iteration

    s = search(i)


    if cost(s) < lowest_cost
        lowest_cost = cost(s)
        best_soln = s
```

Parallel program = NDSeq spec ?

Parallel program:

```
coforeach i in [1..

    b = lower_bound(i)
    if b >= lowest_cost
        end_iteration

    s = search(i)

        best_soln = s
```

Satisfies?

NDSeq specification:

```
nd-foreach i in [1..N]

    b = lower_bound(i)
    if b >= lowest_cost
        end_iteration

    s = search(i)

        best_soln = s
```

What if search(i) has side effect on functionality?

Parallel program = NDSeq spec ?

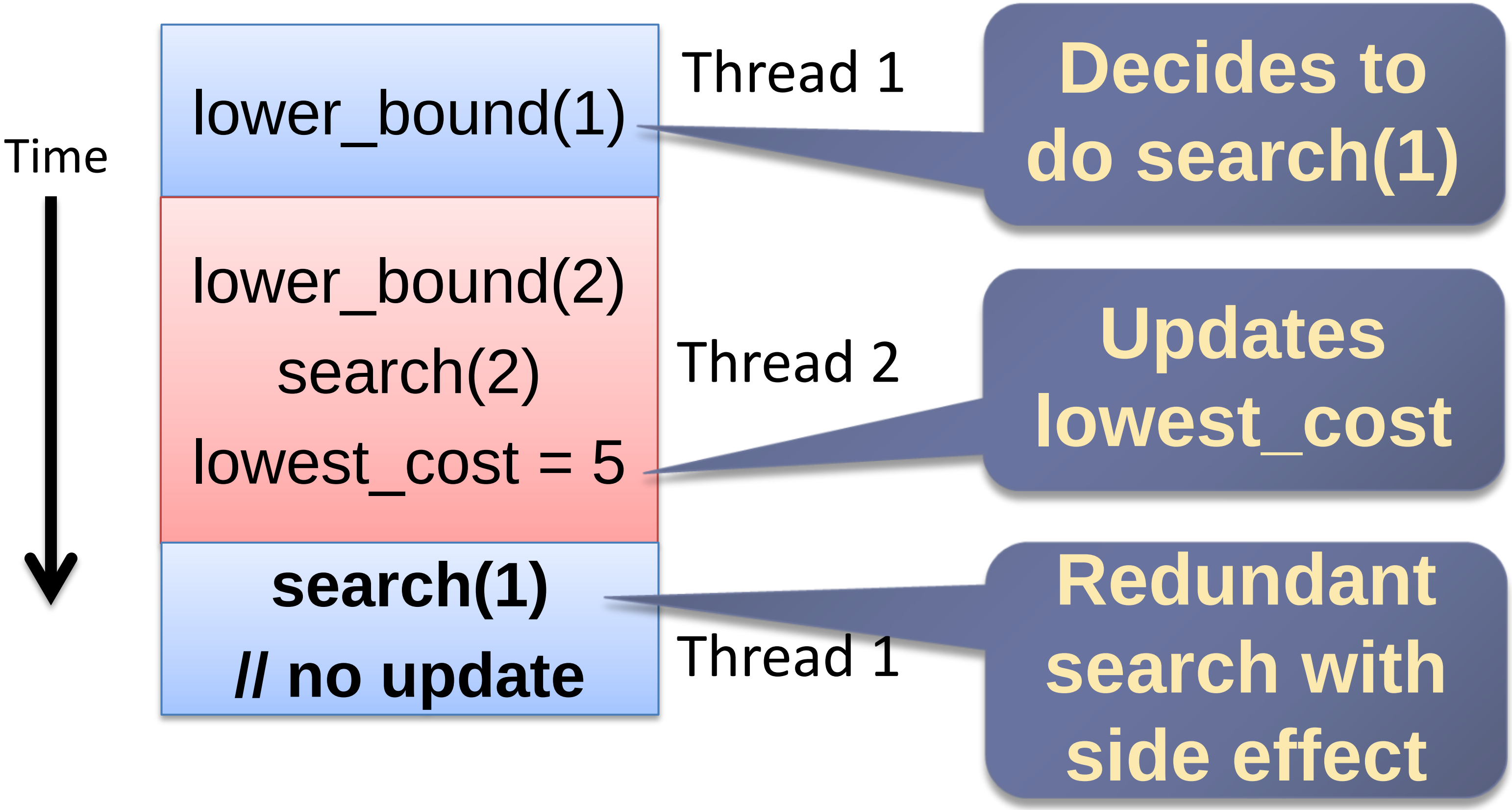
Initially:
lowest_cost = ∞

Search space:

(1) cost(s1): 5
bound: 5

(2) cost(s2): 5
bound: 5

A parallel execution



Time

lower_bound(1)

Thread 1

Decides to do search(1)

lower_bound(2)
search(2)
lowest_cost = 5

Thread 2

Updates lowest_cost

search(1)
// no update

Thread 1

Redundant search with side effect

Parallel program = NDSeq spec ?

Initially:
lowest_cost = ∞

Search space:

(1) cost(s1): 5
bound: 5

(2) cost(s2): 5
bound: 5

A parallel execution

Time ↓

lower_bound(1)

lower_bound(2)
search(2)
lowest_cost = 5

search(1)
// no update

Only possible sequential executions

lower_bound(1)
search(1)
lowest_cost = 5

lower_bound(2)
// no search

lower_bound(2)
search(2)
lowest_cost = 5

lower_bound(1)
// no search

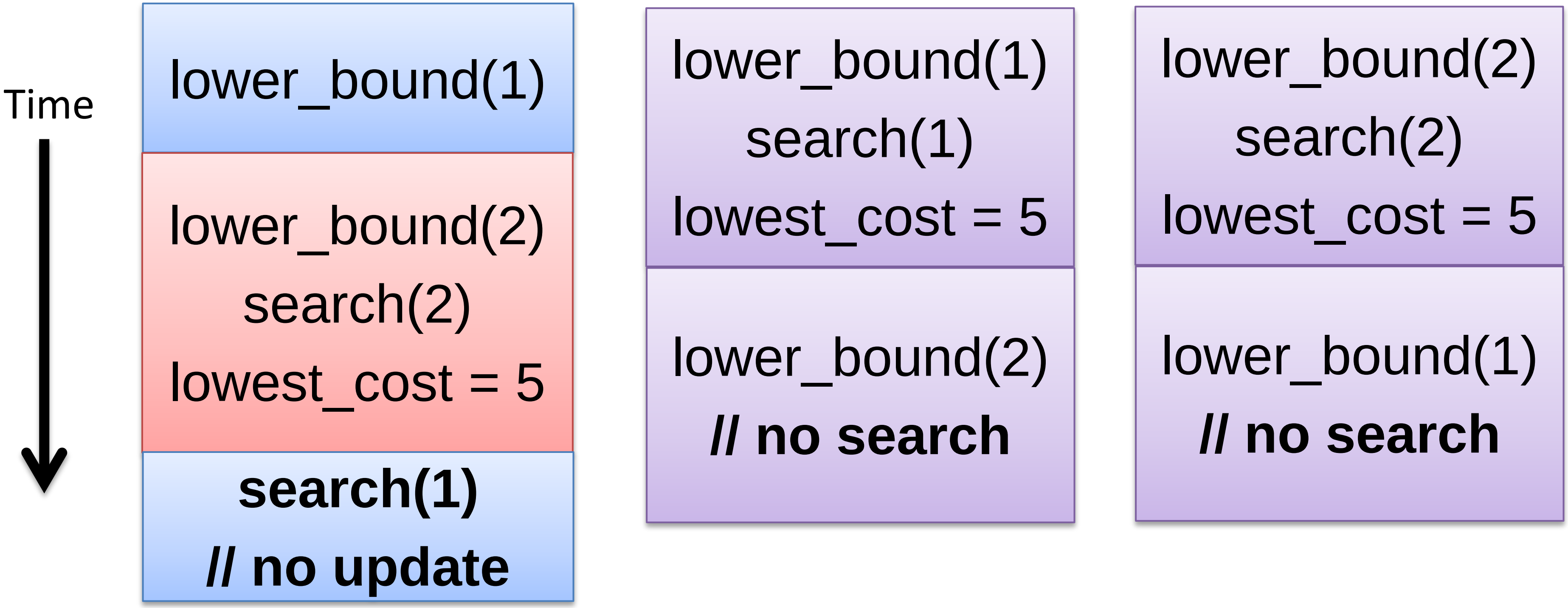
Parallel program ≠ NDSeq spec

NDSeq specification too strict !

- Must allow to NOT prune a redundant search

A parallel execution

Only possible sequential executions



Time

lower_bound(1)

lower_bound(2)
search(2)
lowest_cost = 5

search(1)
// no update

lower_bound(1)
search(1)
lowest_cost = 5

lower_bound(2)
// no search

lower_bound(2)
search(2)
lowest_cost = 5

lower_bound(1)
// no search

Expressing nondeterminism sequentially

Programmer adds if (*)

NDSeq specification:

```
nd-foreach i in [1..N]

  b = lower_bound(i)

  if b >= lowest_cost
    end_iteration

  s = search(i)

  if cost(s) < lowest_cost
    lowest_cost = cost(s)
    best_soln = s
```

New NDSeq specification:

```
nd-foreach i in [1..N]

  b = lower_bound(i)
  if (*)
    if b >= lowest_cost
      end_iteration

  s = search(i)

  if cost(s) < lowest_cost
    lowest_cost = cost(s)
    best_soln = s
```

Expressing nondeterminism sequentially

***** : Pick **true or false** nondeterministically!

New NDSeq specification:

```
nd-foreach i in [1..N]

    b = lower_bound(i)
    if (*)
        if b >= lowest_cost
            end_iteration

    s = search(i)

    if cost(s) < lowest_cost
        lowest_cost = cost(s)
        best_soln = s
```

Programmer asserts:
Skipping body of **if(*)** is safe for functionality (it is optimization)

Expressing nondeterminism sequentially

* : Pick **true or false** nondeterministically!

New NDSeq specification:

```
nd-foreach i in [1..N]

    b = lower_bound(i)
    if (*)
        if b >= lowest_cost
            end_iteration

    s = search(i)

    if cost(s) < lowest_cost
        est_cost = cost(s)
        _soln = s
```

New NDSeq execution

Time ↓

lower_bound(2)
search(2)
lowest_cost = 5

lower_bound(1)
search(1)
// no update

Assign **false**

Redundant search with side effect

Parallel program = NDSeq specification

Parallel program:

```
coforeach i in [1..N]

   b = lower_bound(i)

   if b >= lowest_cost
      end_iteration

   s = search(i)

   synchronized_by(lock)
      if cost(s) < lowest_cost
         lowest_cost = cost(s)
         best_soln = s
```

Satisfies!

NDSeq specification:

```
nd-foreach i in [1..N]

   b = lower_bound(i)
   if (*)
      if b >= lowest_cost
         end_iteration

   s = search(i)


   if cost(s) < lowest_cost
      lowest_cost = cost(s)
      best_soln = s
```

Embedding NDSeq spec. in parallel program

if (true)
by default

```
nd-coforeach i in [1..N]

    b = lower_bound(i)
    if (*)
        if b >= lowest_cost
            end_iteration

    s = search(i)

    synchronized_by(lock)
        if cost(s) < lowest_cost
            lowest_cost = cost(s)
            best_soln = s
```

Our approach: Nondeterministic sequential (NDSeq) specifications

In this talk:

1. Easy-to-write, lightweight specification

- Few simple annotations to indicate **intended nondeterminism** (nd-foreach, if(*))

2. Runtime checking algorithm for testing

- Improves traditional technique using annotations

Parallel program

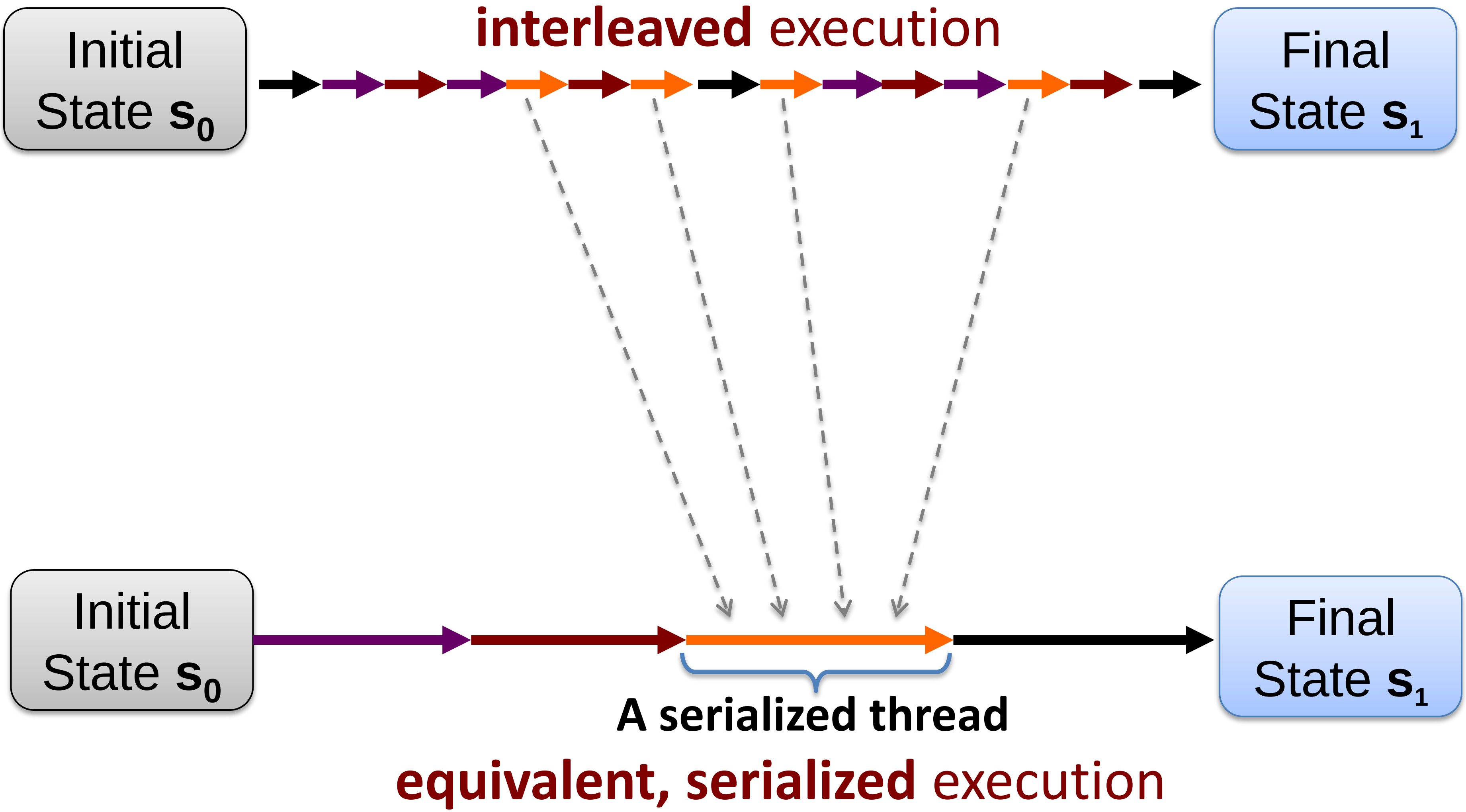
Satisfies?

Nondeterministic sequential specification

Satisfies?

Functional specification

Traditional conflict serializability



interleaved execution

Initial State $\mathbf{s_0}$

Final State $\mathbf{s_1}$

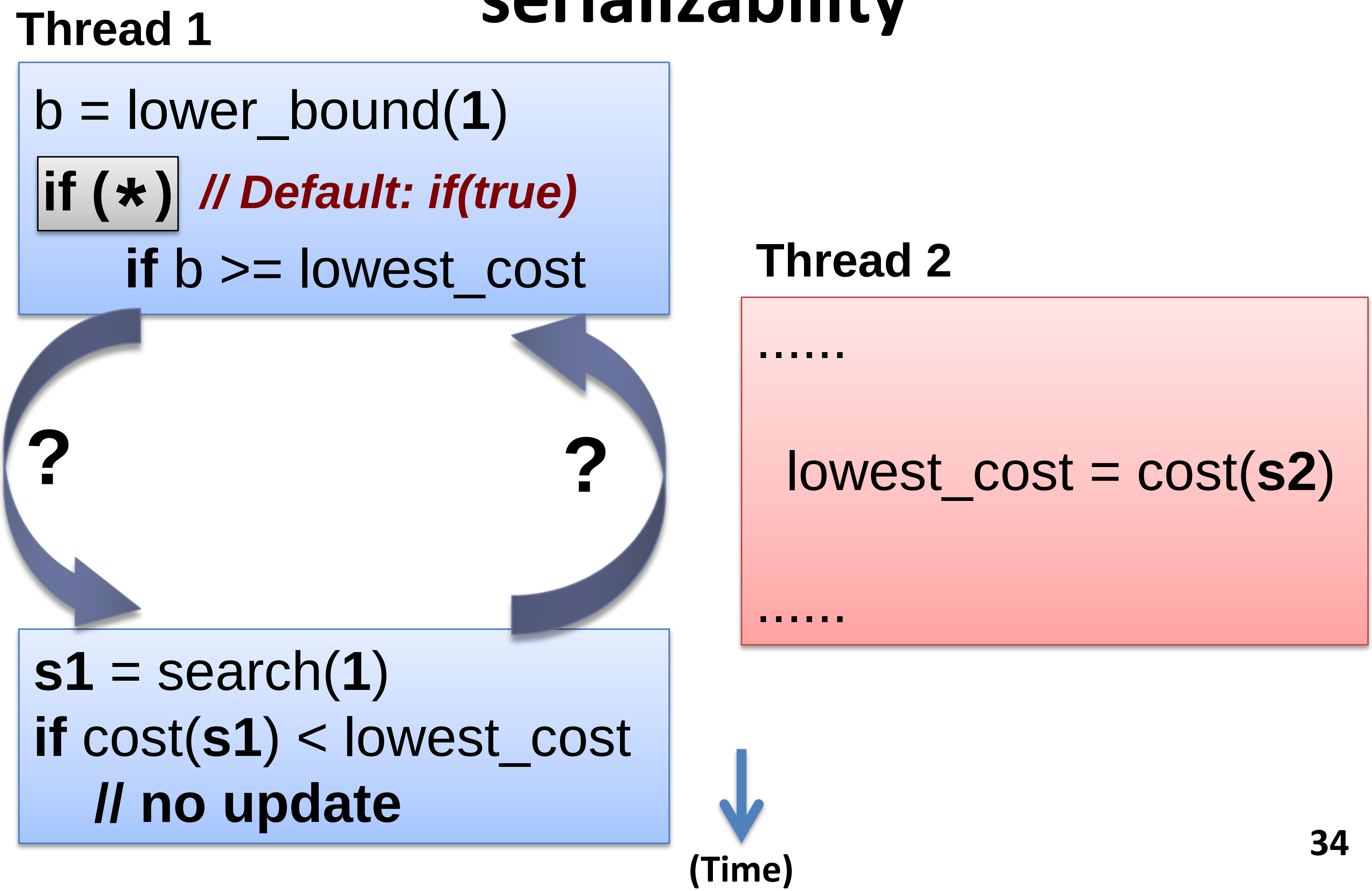
Initial State $\mathbf{s_0}$

Final State $\mathbf{s_1}$

A serialized thread

equivalent, serialized execution

Problem with traditional conflict serializability



Thread 1

```
b = lower_bound(1)
if (*) // Default: if(true)
    if b >= lowest_cost
```

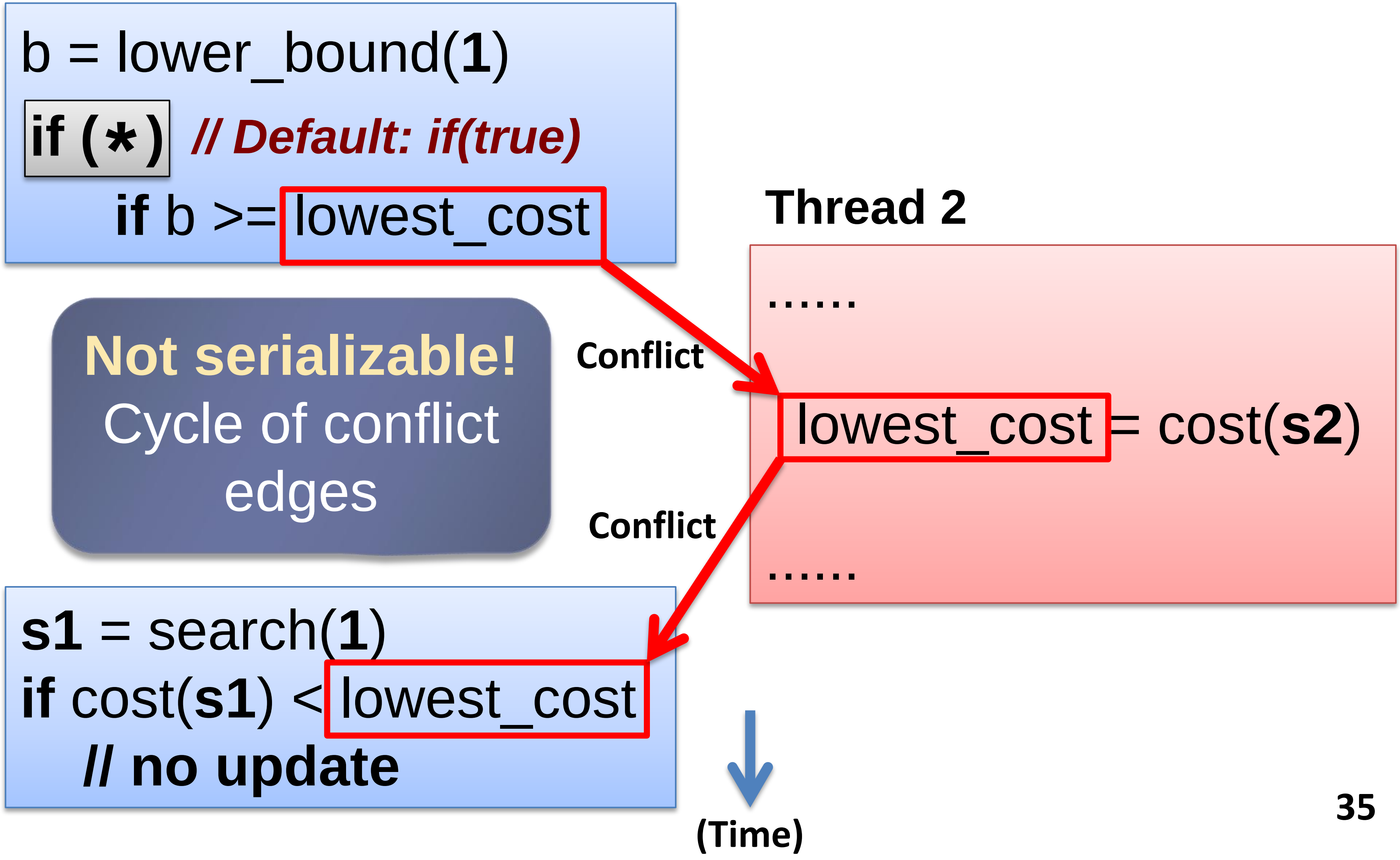
```
s1 = search(1)
if cost(s1) < lowest_cost
    // no update
```

Thread 2

```
……
  lowest_cost = cost(s2)
……
```

(Time)

Problem with traditional conflict serializability



Thread 1

```
b = lower_bound(1)
if (*)  // Default: if(true)
    if b >= lowest_cost
```

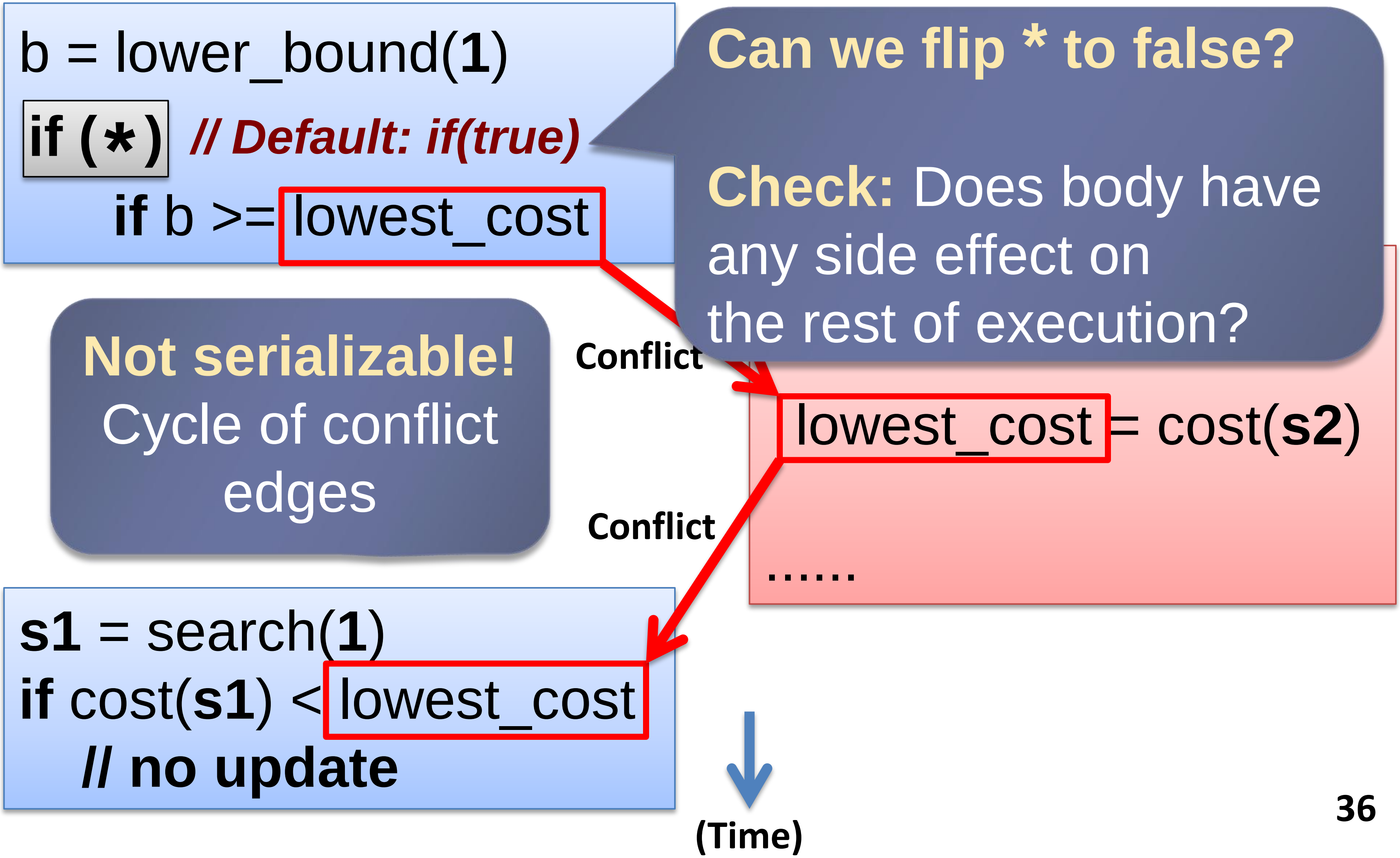
Not serializable!
Cycle of conflict edges

```
s1 = search(1)
if cost(s1) < lowest_cost
    // no update
```

Thread 2

```
……
lowest_cost = cost(s2)
……
```

Problem with traditional conflict serializability



Thread 1

```
b = lower_bound(1)
if (*)  // Default: if(true)
    if b >= lowest_cost
```

Can we flip * to false?

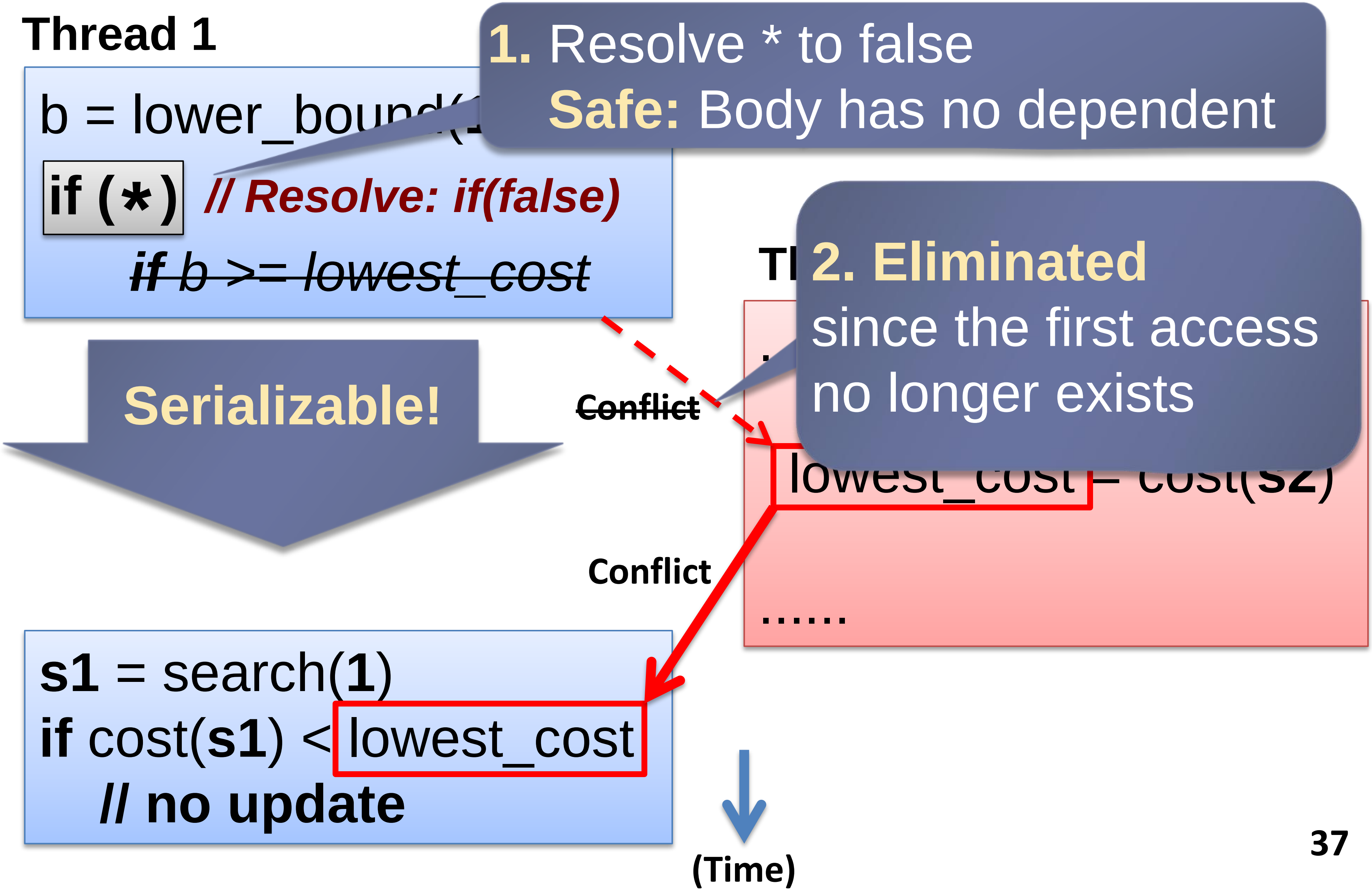
Check: Does body have any side effect on the rest of execution?

Not serializable!
Cycle of conflict edges

```
lowest_cost = cost(s2)

......
```

```
s1 = search(1)
if cost(s1) < lowest_cost
    // no update
```

Using if(*) to rule out false conflict



Thread 1

b = lower_bound(1

if (*) ***// Resolve: if(false)***

~~if b >= lowest_cost~~

Serializable!

1. Resolve * to false
Safe: Body has no dependent

2. Eliminated
since the first access
no longer exists

~~Conflict~~

Conflict

Th

lowest_cost = cost(**s2**)

……

s1 = search(**1**)
if cost(**s1**) < lowest_cost
// no update

(Time)

Traditional conflict serializability:

Thread 1 (a)

Thread 2

Thread 1 (b)

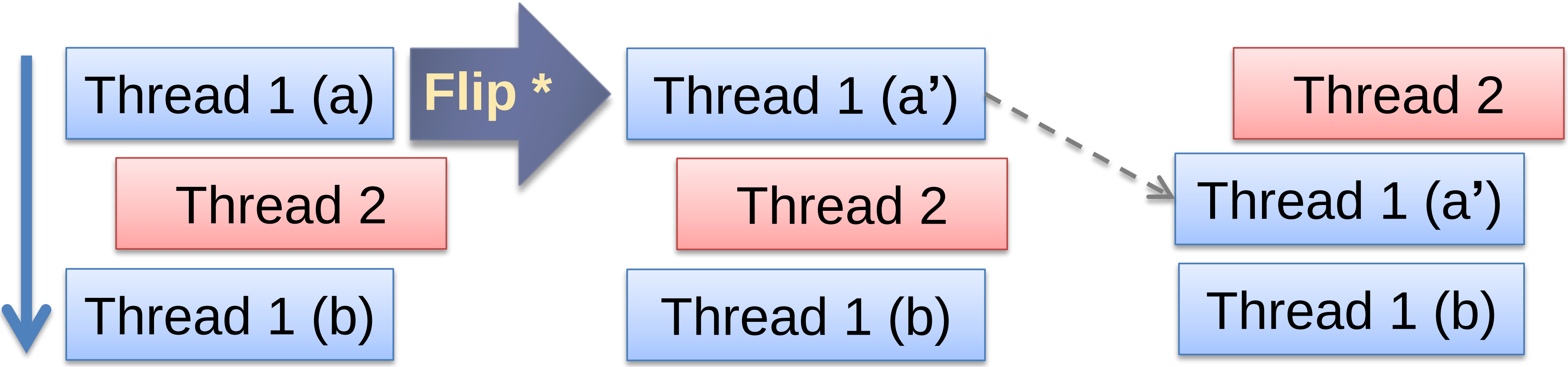
Not serializable!
Cycle of conflict edges

Thread 2

Thread 1 (a)

Thread 1 (b)

Flipping * + traditional conflict serializability:



Thread 1 (a)

Thread 2

Thread 1 (b)

Flip *

Thread 1 (a’)

Thread 2

Thread 1 (b)

Thread 2

Thread 1 (a’)

Thread 1 (b)

Experimental evaluation for Java

- **Wrote and checked NDSeq specifications for:**
  - Java Grande, Parallel Java, Lonestar, and nonblocking data structures
    - **Size:** 40 to 4K lines of code
- **Two claims:**
  1. Easy to write NDSeq specifications
  2. Our technique serialize significantly more executions than traditional methods

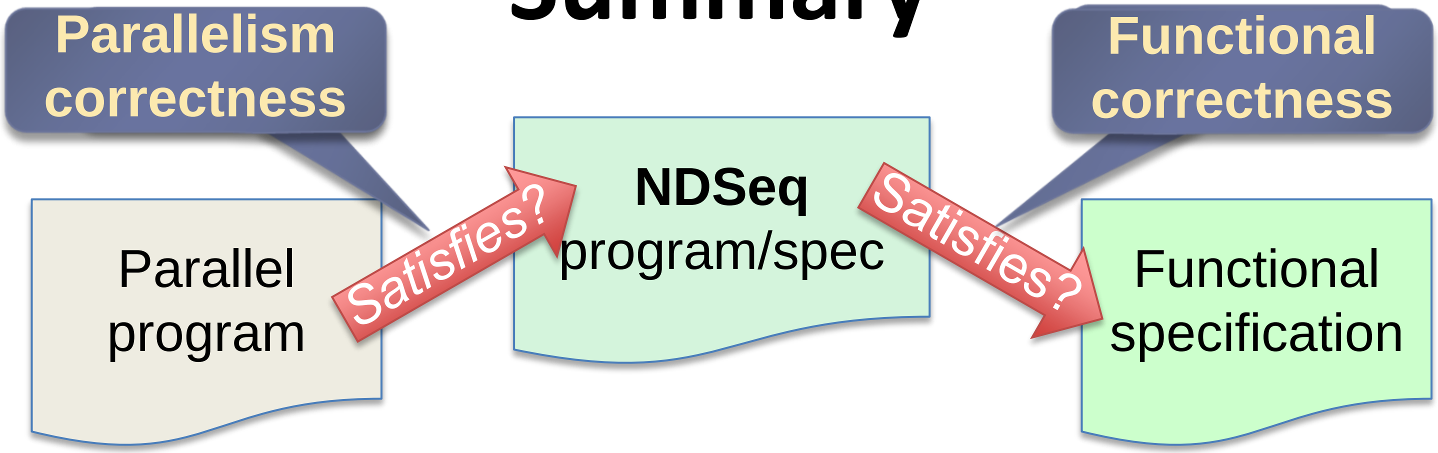
Results 1 (Easy to write specs)

	Benchmark	Line of code	Number of parallel constructs	Number of if(*)
Java Grande	series	800	1	0
	crypt	1.1K	2	0
	raytracer	1.9K	1	0
	montecarlo	3.6K	1	0
Parallel Java	pi3	150	1	0
	keysearch3	200	2	0
	mandelbrot	250	1	0
	phylogeny	4.4K	2	3
	stack	40	1	2
	queue	60	1	2
	meshrefine	1K	1	2

Results 2 (No false alarms)

	Benchmark	Size of execution trace	Number of distinct warnings	
			Traditional	Our technique
Java Grande	series	11k	0	0
	crypt	504K	0	0
	raytracer	6170K	1	1 (real bug)
	montecarlo	1897K	2	0
Parallel Java	pi3	1062K	0	0
	keysearch3	2059K	0	0
	mandelbrot	1707K	0	0
	phylogeny	470K	6	6 (real bug)
	stack	1744	5	0
	queue	868	9	0
	meshrefine	747K	30	0

Summary



- **Key idea:** Specify parallelism correctness using sequential but nondeterministic version of program.
- **Lightweight annotations (nd-foreach, if (*)):** Specify various kinds of intended nondeterminism
  - Without parallel threads and functional specification.
- **Novel runtime checking algorithm**
  - Traditional conflict serializability + Flipping if (*)'s