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 \(\rightarrow\) Functional specification

Satisfies?

Nondeterministic sequential specification

Parallel program \(\rightarrow\)Satisfies?\(\rightarrow\) Functional specification

Satisfies?

Satisfies?
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

Nondeterministic sequential specification

Satisfies?

Functional specification

Satisfies?
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(*))
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
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: \( \text{lowest\_cost} = \infty \)

Parallel program:

```java
coforeach \( i \) in [1..N]

\[ s = \text{search}(i) \]

\text{synchronized\_by(lock)}

\text{if } \text{cost}(s) < \text{lowest\_cost}

\text{lowest\_cost} = \text{cost}(s)

\text{best\_soln} = s
```

\( \text{lowest\_cost} \) and \( \text{best\_soln} \) are global to all threads
Example: Simple branch-and-bound

Goal: Find minimum-cost solution

Initially: lowest_cost = ∞

Parallel program:

```plaintext
coforeach i in [1..N]  
  s = search(i)  
  synchronized_by(lock)  
    if cost(s) < lowest_cost  
      lowest_cost = cost(s)  
      best_soln = s
```

Functionality 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

Non-deterministic sequential specification

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.
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

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
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

<table>
<thead>
<tr>
<th>Time</th>
<th>(1) search(1)</th>
<th>best_soln = s1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2) search(2)</td>
<td>// no update</td>
</tr>
</tbody>
</table>

Result: best_soln = s1

A parallel execution (no equivalent sequential execution)

Thread 1

<table>
<thead>
<tr>
<th>Time</th>
<th>(2) search(2)</th>
<th>best_soln = s2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1) search(1)</td>
<td>// no update</td>
</tr>
</tbody>
</table>

Result: best_soln = s2

Thread 2
Parallel program ≠ NDSeq spec

NDSeq specification too strict!

- Must allow to choose different optimal solutions

Only possible sequential execution

- search(1)
  - best_soln = s1
- search(2)
  - // no update

Result: best_soln = s1

A parallel execution (no equivalent sequential execution)

- search(2)
  - best_soln = s2
- search(1)
  - // no update

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

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:

```plaintext
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:

\[
\text{coforeach } i \in [1..N] \\
\text{\hspace{1cm} } b = \text{lower\_bound}(i) \\
\text{\hspace{1cm} } \text{if } b \geq \text{lowest\_cost} \\
\text{\hspace{1cm} } \quad \text{end\_iteration} \\
\text{\hspace{1cm} } s = \text{search}(i) \\
\text{\hspace{1cm} } \text{synchronized\_by(lock)} \\
\text{\hspace{1.5cm} } \text{if } \text{cost}(s) < \text{lowest\_cost} \\
\text{\hspace{2cm} } \text{lowest\_cost} = \text{cost}(s) \\
\text{\hspace{2cm} } \text{best\_soln} = s
\]

NDSeq specification:

\[
\text{\textbf{nd-foreach} } i \in [1..N] \\
\text{\hspace{1cm} } b = \text{lower\_bound}(i) \\
\text{\hspace{1cm} } \text{if } b \geq \text{lowest\_cost} \\
\text{\hspace{1cm} } \quad \text{end\_iteration} \\
\text{\hspace{1cm} } s = \text{search}(i) \\
\text{\hspace{1cm} } \text{if } \text{cost}(s) < \text{lowest\_cost} \\
\text{\hspace{2cm} } \text{lowest\_cost} = \text{cost}(s) \\
\text{\hspace{2cm} } \text{best\_soln} = s
\]
Parallel program = NDSeq spec?

Parallel program:

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

best_soln = s
```

NDSeq specification:

```plaintext
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?
Initially:
lowest_cost = ∞

Search space:

A parallel execution

Thread 1
lower_bound(1)
Decides to do search(1)

Thread 2
lower_bound(2)
search(2)
lowest_cost = 5
Updates lowest_cost

search(1)
// no update
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:
lower_bound(1)
lower_bound(2)
search(2)
lowest_cost = 5
search(1) // no search

Only possible sequential executions:
lower_bound(1)
search(1)
lowest_cost = 5
lower_bound(2)
// no search

lower_bound(2)
lower_bound(1)
// no search
Parallel program ≠ NDSeq spec

NDSeq specification too strict!

- Must allow to NOT prune a redundant search

A parallel execution

- lower_bound(1)
  - search(1)
    - // no update
  - lower_bound(2)
    - search(2)
      - lowest_cost = 5

Only possible sequential executions

- lower_bound(1)
  - search(1)
    - lowest_cost = 5
  - // no search

- lower_bound(2)
  - search(2)
    - lowest_cost = 5
  - // no search

lower_bound(1)

lower_bound(2)
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:

\[ \text{nd-foreach } i \text{ in } [1..N] \]

\[
\begin{align*}
  b &= \text{lower_bound}(i) \\
  \text{if } \left( \ast \right) \\
  \text{if } b \geq \text{lowest_cost} \\
  &\text{end iteration}
\end{align*}
\]

\[
\begin{align*}
  s &= \text{search}(i) \\
  \text{if } \text{cost}(s) < \text{lowest_cost} \\
  \text{lowest_cost} &= \text{cost}(s) \\
  \text{best_soln} &= s
\end{align*}
\]

New NDSeq execution

lower_bound(2)
search(2)
\text{lowest_cost} = 5

lower_bound(1)
search(1)
\text{// no update}

Redundant search with side effect

Assign false
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

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

Satisfies!
Embedding NDSeq spec. in parallel program

```plaintext
nd-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
```

if (true) by default
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
Traditional conflict serializability

Interleaved execution

Initial State $s_0$ → Final State $s_1$

A serialized thread

Equivalent, serialized execution

Initial State $s_0$ → Final State $s_1$
Problem with traditional conflict serializability

Thread 1

\[ s_1 = \text{search}(1) \]
\[ \textbf{if} \ (\ast) \ // \text{Default: if(true)} \]
\[ \textbf{if} \ b \geq \text{lowest\_cost} \]

\[ b = \text{lower\_bound}(1) \]

\[ \text{if cost}(s_1) < \text{lowest\_cost} \]
\[ // \text{no update} \]

Thread 2

\[ \ldots \]
\[ \text{lowest\_cost} = \text{cost}(s_2) \]
\[ \ldots \]

(Time)
Problem with traditional conflict serializability

Thread 1

\[ b = \text{lower-bound}(1) \]

\[
\text{if } (* )  // \text{Default: if(true)} \]

\[
\text{if } b \geq \text{lowest\_cost} 
\]

\[ s1 = \text{search}(1) \]

\[
\text{if } \text{cost}(s1) < \text{lowest\_cost} \]

\[ // \text{no update} \]

Thread 2

\[ \ldots \]

\[ \text{lowest\_cost} = \text{cost}(s2) \]

\[ \ldots \]

(Time)

Not serializable!
Cycle of conflict edges

Conflict
Conflict

Problem with traditional conflict serializability!
Problem with traditional conflict serializability

Thread 1

\[
s_1 = \text{search}(1)
\]

\[
\text{if} \ (* *) \ // \text{Default: if(true)}
\]

\[
\text{if} \ b \geq \text{lowest_cost}
\]

Not serializable!
Cycle of conflict edges

\[
s_1 = \text{search}(1)
\]

\[
\text{if} \ \text{cost}(s_1) < \text{lowest_cost}
\]

// no update

Can we flip * to false?

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

\[
\text{lowest_cost} = \text{cost}(s_2)
\]

......

(Time)
Using if(*) to rule out false conflict

Thread 1

\[ b = \text{lower\_bound}(s1) \]

\[ \text{if } (* ) \quad // \text{Resolve: if(false)} \]

\[ \text{if } b \geq \text{lowest\_cost} \]

 Serializable!

\[ s1 = \text{search}(1) \]

\[ \text{if } \text{cost}(s1) < \text{lowest\_cost} \quad // \text{no update} \]

1. Resolve * to false

Safe: Body has no dependent

2. Eliminated since the first access no longer exists

\[ \text{lowest\_cost} = \text{cost}(s2) \]

(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)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Line of code</th>
<th>Number of parallel constructs</th>
<th>Number of if(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>series</td>
<td>800</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>crypt</td>
<td>1.1K</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>1.9K</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3.6K</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pi3</td>
<td>150</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>keysearch3</td>
<td>200</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>250</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>phylogeny</td>
<td>4.4K</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>stack</td>
<td>40</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>queue</td>
<td>60</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>meshrefine</td>
<td>1K</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
## Results 2 (No false alarms)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size of execution trace</th>
<th>Number of distinct warnings</th>
<th>Traditional</th>
<th>Our technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>series</td>
<td>11k</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>crypt</td>
<td>504K</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>6170K</td>
<td>1</td>
<td>1 (real bug)</td>
<td></td>
</tr>
<tr>
<td>montecarlo</td>
<td>1897K</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pi3</td>
<td>1062K</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>keysearch3</td>
<td>2059K</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>mandelbrot</td>
<td>1707K</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>phylogeny</td>
<td>470K</td>
<td>6</td>
<td>6 (real bug)</td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>1744</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>queue</td>
<td>868</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>meshrefine</td>
<td>747K</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
• **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