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

Parallel program \[\text{Satisfies?}\] Functional specification \(\phi\)

Parallel program \[\text{Satisfies?}\] Nondeterministic sequential specification \[\text{Satisfies?}\] Functional specification \(\phi\)
Goal: Decompose effort in addressing parallelism and functional correctness

Parallelism Correctness. Handle independently of complex & sequential functional properties.

Functional Correctness. Reason about sequentially, without thread interleavings.

Parallel program

Satisfies?

Nondeterministic sequential specification

Satisfies?

Satisfies?

Functional specification $\phi$
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.

```python
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.

```python
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.

```python
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**.

```python
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**.

```python
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.

```python
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
```

```python
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]):

\[
\begin{align*}
    c &= \text{min\_cost} \\
    b &= \text{lower\_bound}(i) \\
    \text{if } b &\geq c:
        \text{continue}
\end{align*}
\]

cost = \text{compute\_cost}(i)

synchronized (lock):

\[
\begin{align*}
    \text{if } \text{cost} &< \text{min\_cost}:
        \text{min\_cost} = \text{cost} \\
        \text{min\_item} = i
\end{align*}
\]

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?

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

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

prune?(1)
prune?(2)
update(2)
Parallel-Sequential Equivalence?

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

items: (1) bound: 5 cost: 5
(2) bound: 5 cost: 5
min_item: (2)
min_cost: 5
Parallel-Sequential Equivalence?

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).

parallel:
  bound: 5
  cost: 5

sequential:
  bound: 5
  cost: 5
  min_item: (2)
  min_cost: 5

items:
  (1) bound: 5
cost: 5
  (2) bound: 5
cost: 5
Specifying Parallel Correctness

- Parallel program has freedom to:

```python
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.

```python
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

  Can **choose not** to prune item.
  min_cost = cost
  min_item = i

**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
```

```
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
```

Satisfies? Yes.
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)

Initial State $s_0$ --- $\rightarrow$ --- $\rightarrow$ --- $\rightarrow$ --- $\rightarrow$ --- $\rightarrow$ --- $\rightarrow$ --- Final State $s_1$

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

Idea: Serializability?
Conflict-Serializability is Too Strict

Thread 1:

\[
\begin{align*}
c &= \text{min\_cost} \\
b &= \text{lower\_bound}(i) \\
\text{if } \star [\text{true}] : \\
    &\quad \text{if } b \geq c: \text{ // false} \\
\text{cost} &= \text{compute\_cost}(i) \\
\text{if } \text{cost} < \text{min\_cost} : \\
    &\quad \text{ // false}
\end{align*}
\]

Thread 2:

\[
\begin{align*}
\text{min\_cost} &= \text{cost} \\
\end{align*}
\]

Classic Theorem:
Cycle of conflict edges =>
Not serializable!
Relaxing Conflict-Serializability

Thread 1:

\[ c = \text{min\_cost} \]
\[ b = \text{lower\_bound}(i) \]
\[ \text{if } * \text{ [true]}: \]
\[ \quad \text{if } b \geq c: /// \text{false} \]
\[ \text{cost} = \text{compute\_cost}(i) \]
\[ \text{if } \text{cost} < \text{min\_cost}: /// \text{false} \]

Can we set * to false?

Check: Does body have any side effects on execution?

Thread 2:

\[ \text{min\_cost} = \text{cost} \]

...
Relaxing Conflict-Serializability

Thread 1:
\[ c = \text{min\_cost} \]
\[ b = \text{lower\_bound}(i) \]
\[ \text{if } * \ [\text{false}]: \]
\[ \quad \text{if } b \geq c: // \text{false} \]
\[ \text{cost} = \text{compute\_cost}(i) \]
\[ \text{if} \ \text{cost} < \text{min\_cost}: // \text{false} \]

Can we set * to false?

Check: Does body have any side effects on execution?

Thread 2:
\[ \text{min\_cost} = \text{cost} \]

...
Relaxing Conflict-Serializability

Thread 1:

\[ c = \text{min\_cost} \]
\[ b = \text{lower\_bound}(i) \]
\[ \text{if } * \text{ [false]}: \]
\[ \text{if } b \geq c: \text{// false} \]

\[ \text{cost} = \text{compute\_cost}(i) \]
\[ \text{if } \text{cost} < \text{min\_cost}: \text{// false} \]

Local \( c \) is no longer used, so conflicting read of \( \text{min\_cost} \) is irrelevant.

Thread 2:

... 

\[ \text{min\_cost} = \text{cost} \]

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

Iteration 1:
\[ c = \text{min\_cost} \]
\[ b = \text{lower\_bound}(i) \]
if * [false]:
\[ \text{cost} = \text{compute\_cost}(i) \]
if cost < \text{min\_cost}:
  // false

Read different value for min\_cost, but overall behavior is the same.

Iteration 2:
\[ \text{min\_cost} = \text{cost} \]
Traditional conflict serializability:

Not serializable!
Cycle of conflicts.

+ 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.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th># of Parallel Constructs</th>
<th># of if(*)</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>sunflow</td>
<td>24K</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>xalan</td>
<td>302K</td>
<td>1</td>
<td>3</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>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>Benchmark</td>
<td>Size of Trace</td>
<td>Serializability Warnings</td>
<td>Serializability Warnings</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Traditional</td>
<td>Our Technique</td>
</tr>
<tr>
<td>stack</td>
<td>1,744</td>
<td>5 (false)</td>
<td>0</td>
</tr>
<tr>
<td>queue</td>
<td>846</td>
<td>9 (false)</td>
<td>0</td>
</tr>
<tr>
<td>meshrefine</td>
<td>747K</td>
<td>30 (false)</td>
<td>0</td>
</tr>
<tr>
<td>sunflow</td>
<td>24,250K</td>
<td>28 (false)</td>
<td>3 (false)</td>
</tr>
<tr>
<td>xalan</td>
<td>16,540K</td>
<td>6 (false)</td>
<td>2 (false)</td>
</tr>
<tr>
<td>keysearch3</td>
<td>2,059K</td>
<td>2 (false)</td>
<td>0</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>1,707K</td>
<td>1 (false)</td>
<td>0</td>
</tr>
<tr>
<td>phylogeny</td>
<td>470K</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>series</td>
<td>11K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>crypt</td>
<td>504K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>6,170K</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>montecarlo</td>
<td>1,897K</td>
<td>2 (false)</td>
<td>0</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Size of Trace</td>
<td>Serializability Warnings</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>--------------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Traditional</td>
<td>Our Technique</td>
</tr>
<tr>
<td>stack</td>
<td>1,744</td>
<td>5 (false)</td>
<td>0</td>
</tr>
<tr>
<td>queue</td>
<td>846</td>
<td>9 (false)</td>
<td>0</td>
</tr>
<tr>
<td>meshrefine</td>
<td>747K</td>
<td>30 (false)</td>
<td>0</td>
</tr>
<tr>
<td>DaCapo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sunflow</td>
<td>24,250K</td>
<td>28 (false)</td>
<td>3 (false)</td>
</tr>
<tr>
<td>xalan</td>
<td>16,540K</td>
<td>6 (false)</td>
<td>2 (false)</td>
</tr>
<tr>
<td>PJ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>keysearch3</td>
<td>2,059K</td>
<td>2 (false)</td>
<td>0</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>1,707K</td>
<td>1 (false)</td>
<td>0</td>
</tr>
<tr>
<td>phylogeny</td>
<td>470K</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>series</td>
<td>11K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JGF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypt</td>
<td>504K</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>6,170K</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>montecarlo</td>
<td>1,897K</td>
<td>2 (false)</td>
<td>0</td>
</tr>
</tbody>
</table>
Limitations

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

- Commutativity:

```
increment(x);
...
increment(y);
...
```

```
increment(x);
...
increment(y);
...
```
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.