Separating Functional and Parallel Correctness using Nondeterministic Sequential Specifications

Jacob Burnim, George Necula, Koushik Sen
University of California, Berkeley
Parallel Programming is Hard

- **Key Culprit:** Nondeterministic interleaving of parallel threads.
  - Painful to reason simultaneously about parallelism and functional correctness.

- **Goal:** Decompose efforts in addressing parallelism and functional correctness.
  - Allow programmers to reason about functional correctness **sequentially**.
  - Independently show correctness of parallelism.
Our Approach

- **Goal**: Decompose efforts in addressing parallelism and functional correctness.
Our Approach

- **Goal**: Decompose efforts in addressing parallelism and functional correctness.
Our Approach

- **Goal:** Decompose efforts in addressing parallelism and functional correctness.

**Parallelism Correctness.** Prove independently of complex & sequential function correctness.
Our Approach

- **Goal:** Decompose efforts in addressing parallelism and functional correctness.

**Parallelism Correctness.** Prove independently of complex & sequential function correctness.

Want to be able to reason about functional correctness *without parallel interleavings.*

Sequential program / specification

Parallel program

Functional specification
Our Approach

- Use **sequential but nondeterministic** specification for a program’s parallelism.
- User annotates **intended nondeterminism**.
Our Approach

- Use **sequential but nondeterministic** specification for a program’s parallelism.
- User annotates **intended nondeterminism**.

Parallelism correct if adds **no unintended nondeterminism**.

Can address functional correctness **without parallel interleavings**.

Parallel program \(\leq\) Nondeterministic sequential program/spec \(\leq\) Functional specification
Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSEQ) Specifications for Parallel Correctness
- Proving Parallel Correctness
- Future Work
- Conclusions
Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```python
for w in queue:
    if lower_bnd(w) >= best:
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```
Motivating Example

Goal: Find minimum-cost solution.
- Simplified branch-and-bound benchmark.

for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w

Input: List of possible solutions.
Output: Solution from input queue with minimum cost.
Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```python
for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```

- Computes cost of solution w. **Expensive.**
Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```python
for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```

Computes cheap lower bound on cost of w.

Computes cost of solution w. **Expensive.**
Motivating Example

- **Goal**: Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```python
for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```

Computes **cheap** lower bound on cost of `w`.

Prune when `w` cannot have minimum-cost.

Computes cost of solution `w`. **Expensive**.
Motivating Example

queue: 
best: ∞
best_soln: •

for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
Motivating Example

queue: (a) bound: 1 cost: 2 (b) bound: 0 cost: 3 (c) bound: 5 cost: 9
best: ∞
best_soln: •

for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w

prune?(a)
Motivating Example

queue: 
best: 2 
best_soln: 

```python
def for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```

<table>
<thead>
<tr>
<th>(a) bound: 1</th>
<th>(b) bound: 0</th>
<th>(c) bound: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost: 2</td>
<td>cost: 3</td>
<td>cost: 9</td>
</tr>
</tbody>
</table>

prune?(a)
update(a)
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w

prune?(a)
update(a)
prune?(b)
Motivating Example

queue:
best: 2
best_soln:

for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w

(a) bound: 1 cost: 2
(b) bound: 0 cost: 3
(c) bound: 5 cost: 9

prune?(a)
update(a)
prune?(b)
update(b)
Motivating Example

queue:
best: 2
best_soln:

for \((w \text{ in queue})\):
  if \((\text{lower_bnd}(w) \geq \text{best})\):
    continue
  cost = \text{compute_cost}(w)
  if cost < best:
    best = cost
    best_soln = w

bound: 1 cost: 2
bound: 0 cost: 3
bound: 5 cost: 9

prune?\((a)\)
n updates (a)
prune?\((b)\)
update\((b)\)
prune?\((c)\)
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w

queue:
  (a) bound: 1
  cost: 2
  (b) bound: 0
  cost: 3
  (c) bound: 5
  cost: 9

best: 2

best_soln:
Motivating Example

- **Goal**: Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```python
for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```
Motivating Example

How do we parallelize this code?

```python
for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```
Parallelizing our Example

- **Goal:** Find min-cost solution in parallel.
  - Simplified branch-and-bound benchmark.

```plaintext
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```
Parallelizing our Example

- **Goal:** Find min-cost solution in parallel.
- Simplified branch-and-bound algorithm.

```python
parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
```

Loop iterations can be run in parallel.

Updates to best are atomic.
Claim: Parallelization is correct.
- If there are any bugs, they are sequential.
- Want to prove parallelization correct.

```
parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
```
Prove Parallelism Correct?

- **Claim**: Parallelization is correct.
  - If there are any bugs, they are sequential.
  - Want to prove parallelization correct.

**Idea**: Specify that parallel version gives same result as sequential.

```python
parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
```
Parallel-Sequential Equivalence?

queue: (a) bound: 1 cost: 2  (b) bound: 0 cost: 2  (c) bound: 5 cost: 9
best: ∞
best_soln: •

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```
Parallel-Sequential Equivalence?

queue:
best: 2
best_soln:

parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w

(a) bound: 1
   cost: 2
(b) bound: 0
   cost: 2
(c) bound: 5
   cost: 9

prune?(a)
update(a)
Parallel-Sequential Equivalence?

queue:
best: 2
best_soln:

parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
Parallel-Sequential Equivalence?

```python
parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
```

Queue:
- (a) bound: 1, cost: 2
- (b) bound: 0, cost: 2
- (c) bound: 5, cost: 9

Best: 2
Best solution:
- (a) bound: 1, cost: 2
- (b) bound: 0, cost: 2
- (c) bound: 5, cost: 9

Prune? (a)
Update (a)
Prune? (b)
Update (b)
Prune? (c)
Parallel-Sequential Equivalence?

Sequential program always finds best_soln = (a).
Parallel-Sequential Equivalence?

Queue: (a) bound: 1, cost: 2 (b) bound: 0, cost: 2 (c) bound: 5, cost: 9

Best: ∞

Best Solution: •

Parallel-for (w in queue):
  if (lower_bound(w) >= best):
    continue
  cost = compute_cost(w)
  Atomic:
    if cost < best:
      best = cost
      best_solution = w
Parallel-Sequential Equivalence?

queue: (a) bound: 1 cost: 2 (b) bound: 0 cost: 2 (c) bound: 5 cost: 9
best: ∞ best_soln: •

**parallel-for** (w in queue):
  if (lower_bnd(w) >= best):
    continue
cost = compute_cost(w)
**atomic:**
  if cost < best:
    best = cost
    best_soln = w

prune?(a)
Parallel-Sequential Equivalence?

queue: (a) bound: 1 cost: 2 (b) bound: 0 cost: 2 (c) bound: 5 cost: 9
best: 2

parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w

prune?(a)
prune?(b)
update(b)
Parallel-Sequential Equivalence?

queue:
best: 2

best_soln:

parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
Parallel-Sequential Equivalence?

queue: (a) bound: 1 cost: 2  (b) bound: 0 cost: 2  (c) bound: 5 cost: 9
best: 2

best_soln: (a) pruning

parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w

prune?(a)
prune?(b)
update(b)
prune?(c)
update(a)
Parallel-Sequential Equivalence?

queue:
(a) bound: 1
   cost: 2
(b) bound: 0
   cost: 2
(c) bound: 5
   cost: 9

best: 2
best_soln: •

Parallel version can also find best_soln = (b).

prune?(a)
update(b)
prune?(c)
update(a)
Parallel-Sequential Equivalence?

- Parallel and sequential **not** equivalent.
  - **Claim:** But parallelism is correct.

```plaintext
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```
Parallel-Sequential Equivalence?

- Parallel and sequential **not** equivalent.
  - **Claim:** But parallelism is correct.

Some nondeterminism is okay.

Specification for the **parallelism** must indicate **intended** or **algorithmic** nondeterminism.

```python
if cost < best:
    best = cost
best_soln = w
```
NDSEQ Specification

- Use nondeterministic sequential (NDSEQ) version of program as spec for parallelism.

```
parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
```

```
nondet-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```
NDSEQ Specification

Use nondeterministic sequential (NDSEQ) version of program as spec for parallelism.

**parallel-for** (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)

  **atomic**:
  if cost < best:
    best = cost
    best_soln = w

**nondet-for** (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)

  if cost < best:
    best = cost
    best_soln = w

Allow sequential code to perform iterations in a nondeterministic order.
NDSEQ Specification

- Specifies:
  - For every parallel execution, there must exist an NDSEQ execution with the same result.

**parallel-for** (w in queue):
  - if (lower_bnd(w) >= best):
    - continue
  - cost = compute_cost(w)
  - atomic:
    - if cost < best:
      - best = cost
      - best_soln = w

**nondet-for** (w in queue):
  - if (lower_bnd(w) >= best):
    - continue
  - cost = compute_cost(w)
  - if cost < best:
    - best = cost
    - best_soln = w
Parallel-NDSEQ Equivalence?

queue:
best: 2

best_soln:

Parallel:

- prune?(a)
- prune?(b)
- update(b)
- prune?(c)
- update(a)

- No equivalent sequential execution.
- An equivalent NDSEQ execution?
Parallel-NDSEQ Equivalence?

queue: 
best: 2
best_soln:

Parallel:
prune?(a)
prune?(b)
update(b)
prune?(c)
update(a)

NDSEQ:
prune?(b)
update(b)
prune?(a)
update(a)
prune?(c)

Equivalent.
NDSEQ Specification

Does this NDSEQ specification really capture correctness of the parallelism?

**parallel-for** (w in queue):
- if (lower_bnd(w) >= best):
  - continue
- cost = compute_cost(w)

**atomic**:
- if cost < best:
  - best = cost
- best_soln = w

**nondet-for** (w in queue):
- if (lower_bnd(w) >= best):
  - continue
- cost = compute_cost(w)

- if cost < best:
  - best = cost
- best_soln = w
Recall: Our Approach

- Use **sequential but nondeterministic** specification for a program’s parallelism.
- User annotates **intended nondeterminism**.

Parallelism correct if adds **no unintended nondeterminism**.

Can address functional correctness **without parallel interleavings**.

Parallel program $\leq$ Nondeterministic but sequential program/spec $\leq$ Functional specification
Recall: Our Approach

- Use **sequential but nondeterministic** specification for a program’s parallelism.
  - User annotates **intended nondeterminism**.

Prove **independently** of complex functional correctness.

Can address functional correctness **without** parallel interleavings.

Parallel program \( \leq \) Nondeterministic but sequential program/spec \( \leq \) Functional specification
Parallel-NDSEQ Equivalence?

queue:
best: $\infty$

\texttt{best\_soln: •}

\textbf{parallel-for} (\texttt{w in queue}):
\texttt{if} (\texttt{lower\_bnd(w)} $\geq$ \texttt{best}):
\texttt{continue}
\texttt{cost} = \texttt{compute\_cost(w)}
\texttt{atomic:}
\texttt{if} \texttt{cost} < \texttt{best}:
\texttt{best} = \texttt{cost}
\texttt{best\_soln} = \texttt{w}

<table>
<thead>
<tr>
<th></th>
<th>bound</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(b)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(c)</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

49
Parallel-NDSEQ Equivalence?

queue:
  (a) bound: 2
  cost: 2
  (b) bound: 2
  cost: 2
  (c) bound: 5
  cost: 9

best: ∞

best_soln: •

parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w

prune?(a)
prune?(b)
Parallel-NDSEQ Equivalence?

queue: 
  (a) bound: 2
  cost: 2
  (b) bound: 2
  cost: 2
  (c) bound: 5
  cost: 9

best: 2

best_soln: (b)

parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w

prune?(a)
prune?(b)
update(a)
Parallel-NDSEQ Equivalence?

queue:
best: 2
best_soln:

parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w

(bound: 2)
  cost: 2

(bound: 2)
  cost: 2

(bound: 5)
  cost: 9

prune?(a)
update(a)
update(b)
Parallel-NDSEQ Equivalence?

queue: 
(bound: 2
cost: 2)

(b) bound: 2
cost: 2

(c) bound: 5
cost: 9

queue:

best: 2

best_soln:

parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
Parallel-NDSEQ Equivalence?

queue: 
(a) bound: 2
(b) bound: 2
(c) bound: 5

best: 2

best_soln:

Parallel code can avoid pruning by interleaving iterations.

NDSEQ version must prune either (a) or (b).

prune?(a)
prune?(b)
update(a)
update(b)
prune?(c)
Parallel-NDSEQ Equivalence?

queue:
(a) bound: 2
(b) bound: 2
(c) bound: 5

best: 2
cost: 2
cost: 2
cost: 9

best_soln:

Parallel code can avoid pruning by interleaving iterations.

NDSEQ should have freedom to not prune.
NDSEQ Specification

Allows NDSEQ version to nondeterministically not prune when pruning is possible.

**parallel-for** (w in queue):
- if (lower_bnd(w) >= best):
  - continue
- cost = compute_cost(w)
- **atomic**:
  - if cost < best:
    - best = cost
  - best_soln = w

**nondet-for** (w in queue):
- if (lower_bnd(w) >= best):
  - if (*): continue
- cost = compute_cost(w)
- if cost < best:
  - best = cost
  - best_soln = w
NDSEQ Specification

- **Claim:** NDSEQ code a good specification for the correctness of the parallelism.

```
parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
```

```
nondet-for (w in queue):
    if (lower_bnd(w) >= best):
        if (*): continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```
Recall: Our Approach

- Use **sequential but nondeterministic** specification for a program’s parallelism.
- User annotates **intended nondeterminism**.

Prove parallel correctness **independent** of complex functional correctness.

Can address functional correctness **without parallel interleavings**.
NDSEQ Functional Correctness

- **Claim:** much easier
  - Consider recursive Boolean programs
  - Consider Model Checking: Reachability
  - Parallel Programs
    - pushdown system with multiple stacks
    - **Undecidable** [Ramalingam '00]
  - **Nondeterministic sequential** programs
    - pushdown systems
    - **Decidable** [Finkel et al. '97, Bouajjani et al. '97, and others]
Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSEQ) Specifications for Parallel Correctness
- Proving Parallel Correctness
- Future Work
- Conclusions
NDSEQ Specification

- Specifies:
  - For every parallel execution, there exists an NDSEQ execution with the same result.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    if (*): continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
    if (*): continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```
Proving NDSEQ Equivalence

- **Prove**: For every parallel execution, there is an NDSEQ one yielding the same result.

**Parallel:**

- prune?(a)
- prune?(b)
- update(b)
- prune?(c)
- update(a)

**best_soln**: (b)
Proving NDSEQ Equivalence

**Prove:** For every parallel execution, there is an NDSEQ one yielding the same result.

**Parallel:**
- prune?(a)
- prune?(b)
- update(b)
- prune?(c)
- update(a)

**NDSEQ:**
- prune?(b)
- update(b)
- prune?(c)
- prune?(a)
- update(a)

**best_soln:** (b)
Proving NDSEQ Equivalence

**Prove**: For every parallel execution, there is an NDSEQ one yielding the same result.

**Parallel**:
- prune?(a)
- prune?(b)
- update(b)
- prune?(c)
- update(a)

**NDSEQ**:
- prune?(b)
- update(b)
- prune?(c)
- prune?(a)
- update(a)

**best_soln**: (b)
Proving NDSEQ Equivalence

Can we prove that such a rearrangement is always possible?

Parallel:
- prune?(a)
- prune?(b)
- update(b)
- prune?(c)
- update(a)

best_soln: (b)

NDSEQ:
- prune?(b)
- update(b)
- prune?(c)
- prune?(a)
- update(a)

best_soln: (b)
Proving NDSEQ Equivalence

- Is it always possible to move a prune? check later in a parallel execution without changing the result?
Is it always possible to move a prune? check later in a parallel execution without changing the result?

Yes – if the check does not prune.
(1) Can \( \text{prune}\,(x) \) move past \( \text{prune}\,(y) \).

state: \( \sigma_1 \)

if \( \text{lower\_bnd}(x) \geq \text{best} \):
    if (*): continue

state: \( \sigma_2 \)

if \( \text{lower\_bnd}(y) \geq \text{best} \):
    if (*): continue
(1) Can \texttt{prune?(x)} move past \texttt{prune?(y)}.

\begin{itemize}
  \item \texttt{state: } $\sigma_1$
  \begin{itemize}
    \item \texttt{if (lower_bnd(x) $\geq$ best)}:
      \begin{itemize}
        \item \texttt{if (*): continue}
      \end{itemize}
  \end{itemize}
  \begin{itemize}
    \item \texttt{if (lower_bnd(y) $\geq$ best)}:
      \begin{itemize}
        \item \texttt{if (*): continue}
      \end{itemize}
  \end{itemize}
  \begin{itemize}
    \item \texttt{state: } $\sigma_2$
  \end{itemize}

  \begin{itemize}
    \item \texttt{state: } $\sigma_1$
    \begin{itemize}
      \item \texttt{if (lower_bnd(y) $\geq$ best)}:
        \begin{itemize}
          \item \texttt{if (*): continue}
        \end{itemize}
    \end{itemize}
    \begin{itemize}
      \item \texttt{if (lower_bnd(x) $\geq$ best)}:
        \begin{itemize}
          \item \texttt{if (*): continue}
        \end{itemize}
    \end{itemize}
    \begin{itemize}
      \item \texttt{state: } $\sigma_2$
    \end{itemize}
  \end{itemize}
\end{itemize}
Proving NDSEQ Equivalence

- (2) Can prune?(x) move past update?(y).

\[
\begin{align*}
\text{state: } \sigma_1 \\
\text{if (lower_bnd(x) >= best):} \\
\quad \text{if (*): continue} \\
\text{best = *} \\
\text{best_soln = *} \\
\text{state: } \sigma_2
\end{align*}
\]
(2) Can prune?(x) move past update?(y).

- state: $\sigma_1$
  - if (lower_bnd(x) >= best):
    - if (*): continue
    - best = *
    - best_soln = *
  - state: $\sigma_2$

- state: $\sigma_1$
  - best = *
  - best_soln = *
  - if (lower_bnd(x) >= best):
    - if (*): continue
  - state: $\sigma_2$
This is proof by **reduction** [Lipton '75].

[Elmas, et al., POPL 09] has proved atomicity by reduction with SMT solvers.

parallel-for (w in queue):
    if (lower_bnd(w) >= best):
        if (*): continue
    cost = compute_cost(w)
    atomic:
        if cost < best:
            best = cost
            best_soln = w
Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSEQ) Specifications for Parallel Correctness
- Proving Parallel Correctness
- Future Work + Conclusions
Future Work

- Prove parallel-NDSEQ equivalence for real benchmarks.
  - Automated proofs using SMT solving.

- Combine with tools for verifying sequential programs with nondeterminism.
  - Model checking techniques (e.g., CEGAR)

- Also interested in dynamically checking NDSEQ specifications.
NDSEQ and Debugging

- Given parallel execution exhibiting error:
  - Can we produce an NDSEQ trace exhibiting the same wrong behavior?
  - If so, bug is sequential and programmer can debug on a sequential (but NDSEQ) trace.
  - Can we efficiently produce NDSEQ trace given static proof of parallel correctness?

- Dynamically checking NDSEQ specs?
  - Ideally, efficiently: (1) finds equivalent NDSEQ trace, or (2) localizes parallel bug.
Questions?

Email jburnim@cs.berkeley.edu