A Lattice-Based Approach to Deterministic Parallelism with Shared State

Lindsey Kuper and Ryan R. Newton
Indiana University
University of California, Berkeley
August 16, 2012
let _ = put l 3 in
let par v = get l
        _ = put l 4
    in v
What do we want?
What do we want?

- A **deterministic** program is one that always produces the same observable result on multiple runs.
What do we want?

- A **deterministic** program is one that always produces the same observable result on multiple runs.
- A **deterministic-by-construction** programming model is one that only allows deterministic programs to be written.
What do we want?

- A **deterministic** program is one that always produces the same observable result on multiple runs.

- A **deterministic-by-construction** programming model is one that only allows deterministic programs to be written.

- Examples: Kahn process networks, Intel Concurrent Collections, Haskell’s monad-par, ...
let _ = put \( l \ 3 \) in

let par \( v = \) get \( l \)

\[ _ = \text{put} \ l \ 4 \]

in \( v \)
let _ = put l 3 in
let par v = get l
  _ = put l 4
  in v

Serialize?
let _ = put l 3 in
let par v = get l in
let _ = put l 4
in v
Serialize?

```
let _ = put l 3 in
let par v = get l
_ = put l 4
in v
```
Disallow shared state?

```
let _ = put l 3 in
let \(\text{par } v\) = get \(l\)
    _ = put \(l\) 4
    in \(v\)
```
Disallow shared state?

\[
\begin{align*}
\text{let } _ &= \text{put } l \ 3 \ \text{in} \\
\text{let par } v &= \text{get } l \\
_ &= \text{put } l \ 4 \\
\text{in } v
\end{align*}
\]
Disallow shared state?

let _ = put \( l \ 3 \) in

let par \( v = \) get \( l \)

_ = put \( l \ 4 \)

in \( v \)
let _ = put l 3 in
let par \( v = \) get l
  _ = put l 4
in \( v \)
Disallow multiple assignment?

```
let _ = put l 3 in
let par v = get l
  _ = put l 4  
  X

in v
```
A few single-assignment languages
A few single-assignment languages

- Historically:
  - Compel (Tesler and Enea, 1968)
A few single-assignment languages

- Historically:
  - Compel (Tesler and Enea, 1968)
  - Id, I-Structures and IVars (Arvind et al., 1989)
A few single-assignment languages

- Historically:
  - Compel (Tesler and Enea, 1968)
  - Id, I-Structures and IVars (Arvind et al., 1989)

- Today:
  - Intel Concurrent Collections (Budimlić et al., 2010)
    - Specifically, Featherweight CnC
A few single-assignment languages

Historically:
- Compel (Tesler and Enea, 1968)
- Id, I-Structures and IVars (Arvind et al., 1989)

Today:
- Intel Concurrent Collections (Budimlić et al., 2010)
  - Specifically, Featherweight CnC
- monad-par for Haskell (Marlow et al., 2011)
Disallow multiple assignment?

```
let _ = put l 3 in
let par v = get l
  _ = put l 4 ×
  in v
```
Deterministic programs that single-assignment forbids

let _ = put l 3 in
let par v = get l
    _ = put l 3
    in v
Deterministic programs that single-assignment forbids

\[
\begin{align*}
\text{let } _ &= \text{put } l \ 3 \ \text{in} \\
\text{let } \text{par } v &= \text{get } l \\
\text{in } v
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{par } _ &= \text{put } l \ (4, \bot) \\
&\quad _ = \text{put } l \ (\bot, 3) \\
&\quad \text{in let } v = \text{get } l \ \text{in } v
\end{align*}
\]
Kahn process networks (Kahn, 1974)
Kahn process networks (Kahn, 1974)
Kahn process networks (Kahn, 1974)

Diagram showing network connections between nodes labeled $C_1$, $C_2$, $C_3$, $C_4$, and $C_5$. Arrows indicate direction of process flow.
Kahn process networks (Kahn, 1974)
Kahn process networks (Kahn, 1974)
Kahn process networks (Kahn, 1974)
Kahn process networks (Kahn, 1974)
Kahn process networks (Kahn, 1974)
Kahn process networks (Kahn, 1974)

\[ \text{hist}(\text{in}(C_3)) : [3, 0, 5, \ldots] \]
Kahn process networks (Kahn, 1974)

\[ \text{hist(in(C3)): [3, 0, 5, ...]} \quad \text{hist(out(C3)): [6, 1, 120, ...]} \]
Monotonicity
Monotonicity
Monotonicity

$f$ is monotonic iff $x \leq y \implies f(x) \leq f(y)$
Monotonicity in KPNs

\[ f \text{ is monotonic iff } x \leq y \implies f(x) \leq f(y) \]
Monotonicity in KPNs

\[ f \text{ is monotonic iff } x \leq y \implies f(x) \leq f(y) \]
Monotonicity in KPNs

\[ f \text{ is monotonic iff } x \leq y \implies f(x) \leq f(y) \]
Monotonicity in KPNs

\[ f \text{ is monotonic iff } x \leq y \implies f(x) \leq f(y) \]

\[
\begin{array}{c}
C_1 \quad \xrightarrow{0} \quad C_3 \quad \xrightarrow{3} \quad C_5 \\
\text{in}(C_3) \quad \text{out}(C_3)
\end{array}
\]
Monotonicity in KPNs

\[ f \text{ is monotonic iff } x \leq y \implies f(x) \leq f(y) \]

Diagram:

\[ C_1 \xrightarrow{5 \ 0 \ 3} C_3 \xrightarrow{\text{in}(C_3) \ \text{out}(C_3)} C_5 \]
Monotonicity in KPNs

\[ f \text{ is monotonic iff } x \leq y \implies f(x) \leq f(y) \]
Monotonicity in KPNs

$f$ is monotonic iff $x \leq y \implies f(x) \leq f(y)$
Monotonicity in KPNs

$f$ is monotonic iff $x \leq y \implies f(x) \leq f(y)$
Monotonicity in KPNs

\[ f \text{ is monotonic iff } x \leq y \implies f(x) \leq f(y) \]

For KPNs, the \( \leq \) relation is just \textit{prefix-of}:

\[
\begin{align*}
[3, 0] \text{ prefix-of } [3, 0, 5] & \implies [6, 1] \text{ prefix-of } [6, 1, 120] \\
& \ldots
\end{align*}
\]
Monotonicity causes deterministic parallelism!
Back to single-assignment languages

let _ = put $l_1$ 4 in
    let _ = put $l_2$ 3 in
        let par _ = put $l_4$ 3
            _ = put $l_3$ 5
        in get $l_4$
Back to single-assignment languages

let _ = put \( l_1 4 \) in
    let _ = put \( l_2 3 \) in
        let par _ = put \( l_4 3 \)
            _ = put \( l_3 5 \)
    in get \( l_4 \)

Store:

<table>
<thead>
<tr>
<th>( l_1 )</th>
<th>4</th>
</tr>
</thead>
</table>
let _ = put $l_1$ 4 in
let _ = put $l_2$ 3 in
let par _ = put $l_4$ 3
_ = put $l_3$ 5
in get $l_4$

Store:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$</td>
<td>4</td>
</tr>
<tr>
<td>$l_2$</td>
<td>3</td>
</tr>
</tbody>
</table>
Back to single-assignment languages

```
let _ = put $l_1$ 4 in
let _ = put $l_2$ 3 in
let par _ = put $l_4$ 3
_ = put $l_3$ 5
in get $l_4$
```

Store:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$</td>
<td>4</td>
</tr>
<tr>
<td>$l_2$</td>
<td>3</td>
</tr>
<tr>
<td>$l_3$</td>
<td>5</td>
</tr>
</tbody>
</table>
let _ = put $l_1$ 4 in

let _ = put $l_2$ 3 in

let par _ = put $l_4$ 3

_ = put $l_3$ 5

in get $l_4$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$</td>
<td>4</td>
</tr>
<tr>
<td>$l_2$</td>
<td>3</td>
</tr>
<tr>
<td>$l_3$</td>
<td>5</td>
</tr>
<tr>
<td>$l_4$</td>
<td>3</td>
</tr>
</tbody>
</table>
Back to single-assignment languages

let _ = put \( l_1 \ 4 \) in

let _ = put \( l_2 \ 3 \) in

let par _ = put \( l_4 \ 3 \)

_ = put \( l_3 \ 5 \)

in get \( l_4 \)

Store:

| \( l_1 \) | 4 |
| \( l_2 \) | 3 |
| \( l_3 \) | 5 |
| \( l_4 \) | 3 |

For stores, the \( \leq \) relation is \( \subseteq \):

\[
\{ l_1 \rightarrow 4, \ l_2 \rightarrow 3 \} \subseteq \{ l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_3 \rightarrow 5 \} \implies
\]

\[
\{ l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_4 \rightarrow 3 \} \subseteq \{ l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_3 \rightarrow 5, \ l_4 \rightarrow 3 \}
\]
Generalizing our notion of monotonicity

For stores, the \( \leq \) relation is \( \subseteq \):

\[
\{ l_1 \rightarrow 4, \ l_2 \rightarrow 3 \} \subseteq \{ l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_3 \rightarrow 5 \} \implies \\
\{ l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_4 \rightarrow 3 \} \subseteq \{ l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_3 \rightarrow 5, \ l_4 \rightarrow 3 \}
\]

- Given stores \( S \) and \( S' \), we say that \( S \leq S' \) iff:
  - \( \text{dom}(S) \subseteq \text{dom}(S') \), and
  - for all locations \( l \) in \( \text{dom}(S) \), \( S(l) = S'(l) \)
Generalizing our notion of monotonicity

For stores, the \( \leq \) relation is \( \subseteq \): 

\[
\{l_1 \rightarrow 4, \ l_2 \rightarrow 3\} \subseteq \{l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_3 \rightarrow 5\} \implies \quad \{l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_4 \rightarrow 3\} \subseteq \{l_1 \rightarrow 4, \ l_2 \rightarrow 3, \ l_3 \rightarrow 5, \ l_4 \rightarrow 3\}
\]

- Given stores \( S \) and \( S' \), we say that \( S \leq S' \) iff:
  - \( \text{dom}(S) \subseteq \text{dom}(S') \), and
  - for all locations \( l \) in \( \text{dom}(S) \), \( S(l) \subseteq S'(l) \)
Idea: restrict reads

let _ = put l 3 in

let par v = get l 4

_ = put l 4

in v
Idea: restrict reads

let _ = put l 3 in

let par \( v = \) get \( l(4) \)

\( _ = \) put \( l(4) \)

in \( v \)
Idea: restrict reads

let _ = put l 3 in
let par v = get l \(4\)
  _ = put l 4
in v

let _ = put l 3 in
let par v = get l 4
  _ = put l 4
  _ = put l 5
in v
Idea: restrict reads

```
let _ = put l 3 in
  let par v = get l 4
    _ = put l 4
    in v
let _ = put l 3 in
  let par v = get l 4
    _ = put l 4
    _ = put l 5
    in v
```
Idea: restrict reads

```
let _ = put l 3 in
  let par v = get l (4)
    _ = put l 4
    in v
```

```
let _ = put l 3 in
  let par v = get l 4
    _ = put l 4
      _ = put l 5
    in v
```
Idea: restrict reads

```
let _ = put l 3 in
  let par v = get l 4
    _ = put l 4
  in v

let _ = put l 3 in
  let par v = get l 4
    _ = put l 4
    _ = put l 5
  in v

return 4
```
Monotonically increasing writes
+ restricted reads
= deterministic-by-construction parallelism
Parameterizing our language: “LVars”

**IVar**

**Pair of IVars**

**Counter**
Parameterizing our language: “LVars”

Pair of IVars
Parameterizing our language: “LVars”

Pair of lVars
Parameterizing our language: “LVars”

Pair of LVars

getFst

getSnd

"tripwire"
Parameterizing our language: “LVars”

\[
\text{let } p = \text{new in}
\]
\[
\text{let } _ = \text{put } p \{(\bot, 4)\} \text{ in}
\]
\[
\text{let par } v_1 = \text{getFst } p
\]
\[
\text{let } _ = \text{put } p \{(3, 4)\}
\]
\[
\text{in ... } v_1 ...
\]
More in our paper draft and TR
More in our paper draft and TR

- Complete syntax and semantics
More in our paper draft and TR

- Complete syntax and semantics
- Proof of determinism
  - A “frame-rule-like” property
  - Location renaming is surprisingly tricky!
More in our paper draft and TR

- Complete syntax and semantics
- Proof of determinism
  - A “frame-rule-like” property
  - Location renaming is surprisingly tricky!
- Subsuming existing models
  - KPNs, CnC, monad-par
More in our paper draft and TR

- Complete syntax and semantics
- Proof of determinism
  - A “frame-rule-like” property
  - Location renaming is surprisingly tricky!
- Subsuming existing models
  - KPNs, CnC, monad-par
- Support for controlled nondeterminism
  - “probation” state
Thanks!

Email: lkuper@cs.indiana.edu
Twitter: @lindsey
Web: cs.indiana.edu/~ lkuper
Research group: lambda.cs.indiana.edu