A Lattice-Based Approach to Deterministic Parallelism

Lindsey Kuper and Ryan R. Newton
Indiana University

MPI-SWS
30 January 2013
What does this program evaluate to?

```
let _ = put l 0 in
  let par v = get l
    _ = put l 8
    in v
```
Disallow multiple writes?

let _ = put l 0 in
let par v = get l
    _ = put l 8
    in v
Disallow multiple writes?

let _ = put l 0 in

let par v = get l

_ = put l 8 \times

in v

Tesler and Enea, 1968
Arvind et al., 1989

“IVars”
Deterministic programs that single-assignment forbids

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

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

let par _ = put l (4, ⊥)
  _ = put l (⊥, 3)
in get l
Deterministic programs that single-assignment forbids

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

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

let par _ = insert l "1111"
  _ = insert l "1100"
  in get l
Concurrent Collections

Zoran Budimlić, Michael Burke, Vincent Caro, Kathleen Kuchta, Geoff Lowney, Ryan Newton, Jano Peldschus, David Peinan, Vivek Sarkar, Frank Schlimbach, Saghaoln Toomey

1. Introduction

With multicores processors, parallel computing is going mainstream. Yet most software is still written in traditional serial languages with explicit threading. High-level parallel programming models, after four decades of proposals, have still not been widespread adoption. This is beginning to change. Systems like Microsoft are streaming based on implicit parallelism. Other systems like Nvidia CUDA are pursuing these, providing a restricted programming model to the user but also exposing too many of the hardware details. The payoff for high-level parallelism is clear— ECs can provide semantic guarantees and can simplify the understanding, debugging, and testing of a parallel program.

In this paper we introduce the Concurrent Collections (CC) programming model, built on past work on Threading Local. CC is based on the same family of dataflow and stream processing languages—a program is a graph of threads, communicating with one another. In CC, these computations are called steps, and are related by control and data dependence. CC is possibly determinate. This limits CC’s scope, but compared to its more narrow counterparts (StreamIt, NVT, etc), CC is suited for many applications—incorporating static and dynamic forms of task, data, loop, pipeline, and tree parallelism.

Daily mainstream parallelism will require reaching the large community of non-professional programmers—scientists, engineers, and financial analysts—but reaching these requires a separation of concern between applications high-level and parallel implementations. We say that the former is the concern of the domain expert and the latter is the performance tuning expert. The tuning expert is given the maximum possible freedom to map the computation onto the target architecture and is not required to have an understanding of the domain. A strength of CC is that it is simultaneously a database-like, parallel model.
Lemma 3.2. (Monotonicity) If $\sigma \rightarrow \sigma'$, then $\sigma \leq \sigma'$.

The key language feature that enables determinism is the single assignment condition. The single assignment condition guarantees monotonicity of the data collection $A$. We view $A$ as a partial function from integers to integers and the single assignment condition guarantees that we can establish an ordering based on the non-decreasing domain of $A$. 

Budimlić et al., 2010
Monotonicity

$f$ is monotonic iff, for a given $\leq$,

\[ x \leq y \implies f(x) \leq f(y) \]
...to KPNs

Gilles KAHN

INRIA Laboratoire, Domaine de Voluceau, 78150
Boulogne, France

and

Commissariat à l’Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemas are given. We hope in this way to make a case for a more formal (i.e., mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing system programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on Fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type Integer channel are declared at line (1), and for any simple variable declaration, a channel is declared as a channel. These processes, if and if-else are

```plaintext
Begin
  if X then unit(0) else unit(0)
End
```

Process $g$: $g$ : Integer out V; Integer out W; Begin $g$ : Integer out V; Integer out W; Begin integer T:

```plaintext
if A then send 1 on V else send 1 on W;
B := B + T;
End
```

End

Comment : body of main program

```plaintext
f(X,Y,Z) par g(X,Y,Z,T) par h(X,Y,Z,T)
End
```

Fig.1. Simple parallel program S.

Fig.2. The schema $F$ for the program S.
Monotonicity means that receiving more input at a computing station can only provoke it to send more output. Indeed this a crucial property since it allows parallel operation: a machine need not have all of its input to start computing, since future input concerns only future output.

The kind of parallel programming we have studied in this paper is severely limited: it can produce only determinate programs.
Monotonicity enables deterministic parallelism!
Parameterizing our language: LVars

IVar

Pair of IVars

Counter
Parameterizing our language: LVars

Pair of LVars
Parameterizing our language: LVars

Pair of LVars
Parameterizing our language: LVars

Pair of IVars

getFst

getSnd

"tripwire"
Parameterizing our language: LVars

let _ = put p { (⊥, 4) } in
  let par v1 = getFst p
  _ = put p { (3, 4) }
  in ...v1 ...

Pair of IVars
Parameterizing our language: LVars

let \_ = put \( p \{ (\bot, 4) \} \) in
let par \( v_1 = \text{getFst} \ p \)
\_ = put \( p \{ (3, 4) \} \)
in \ldots v_1 \ldots

\text{getFst} \ p \overset{\Delta}{=} \text{get} \ p \{ (n, \bot) \mid n \in \mathbb{N} \}
Two take-aways

Monotonicity enables deterministic parallelism

Monotonically increasing writes + threshold reads = deterministic parallelism
Determinism for $\lambda_{V\sigma}$

\[
\sigma' = \sigma''
\]
Determinism for $\lambda_{\text{LVar}}$

Diamond

$$\sigma_a \quad \sigma_b \quad \sigma_c$$

Determinism

$$\sigma' = \sigma''$$
Determinism for $\lambda_{LVar}$

Diamond

$\sigma$

$\sigma_a$, $\sigma_b$, $\sigma_c$

“Independence”

Determinism

$\sigma$

$\sigma'$ $\sigma''$

$\sigma' = \sigma''$
Why we need Independence

To show: There exists $\sigma_c$ such that

$$\langle S ; e_1 e_2 \rangle$$

$$\langle S_{a_1 \cup S} S_{a_2} ; e_{a_1} e_{a_2} \rangle$$

$$\langle S_{b_1 \cup S} S_{b_2} ; e_{b_1} e_{b_2} \rangle$$

$$\sigma_c$$
Why we need Independence

By induction hypothesis, there exist \( \sigma_{c_1}, \sigma_{c_2} \) such that

\[
\langle S; e_1 \rangle \quad \langle S; e_2 \rangle
\]

\[
\langle S_{a_1}; e_{a_1} \rangle \quad \langle S_{b_1}; e_{b_1} \rangle \quad \langle S_{a_2}; e_{a_2} \rangle \quad \langle S_{b_2}; e_{b_2} \rangle
\]

\[
\sigma_{c_1} \quad \sigma_{c_2}
\]

\[
(= \langle S_{c_1}; e_{c_1} \rangle \text{ or error}) \quad (= \langle S_{c_2}; e_{c_2} \rangle \text{ or error})
\]

To show: There exists \( \sigma_c \) such that

\[
\langle S; e_1 e_2 \rangle
\]

\[
\langle S_{a_1} \sqcup_S S_{a_2}; e_{a_1} e_{a_2} \rangle \quad \langle S_{b_1} \sqcup_S S_{b_2}; e_{b_1} e_{b_2} \rangle
\]

\[
\sigma_c
\]
Determinism for $\lambda_{LVar}$

Diamond

Independence

\[
\langle S; e \rangle \leftrightarrow \langle S'; e' \rangle \quad \frac{\langle S \sqcup S''; e \rangle \leftrightarrow \langle S' \sqcup S''; e' \rangle}{\langle S''; e' \rangle}
\]

Determinism

$\sigma' = \sigma''$
Independence

\[ \langle S; e \rangle \leftrightarrow \langle S'; e' \rangle \]

\[ \langle S \sqcup S S''; e \rangle \leftrightarrow \langle S' \sqcup S S''; e' \rangle \]
Independence

“That looks kind of like a frame rule.”
— Amal, March 2012
Independence

Frame

\[
\frac{\{p\} \; c \; \{q\}}{\{p \ast r\} \; c \; \{q \ast r\}}
\]

Independence

\[
\langle S; \; e \rangle \leftrightarrow \langle S'; \; e' \rangle \\
\langle S \sqcup_S S''; \; e \rangle \leftrightarrow \langle S' \sqcup_S S''; \; e' \rangle
\]
More in our TR

- Complete syntax and semantics
- Proof of determinism
- Subsuming existing models
  - KPNs, monad-par
- Support for controlled nondeterminism
  - “probation” state
LATTICE-BASED DETERMINISTIC PARALLELISM
Thanks!

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

---

LATTICE-BASED
DETERMINISTIC PARALLELISM