LVars:

lattice-based data structures for deterministic parallelism

Lindsey Kuper and Ryan R. Newton
Indiana University

Hacker School
June 10, 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

\[
\begin{align*}
\text{let } & \_ = \text{put } l \ 8 \ \text{in} \\
& \text{let } \text{par } v = \text{get } l \\
& \_ = \text{put } l \ 8 \\
& \text{in } v
\end{align*}
\]
Deterministic programs that single-assignment forbids

\[
\begin{align*}
\text{let } &\_ = \text{put } l \ 8 \ \text{in} \\
\text{let } &\text{par } v = \text{get } l \\
\_ &\text{ = put } l \ 8 \\
\text{in } &v
\end{align*}
\]

\[
\begin{align*}
\text{let } &\text{par }\_ = \text{put } l \ (4, \bot) \\
\_ &\text{ = put } l \ (\bot, 3) \\
\text{in } &\text{get } l
\end{align*}
\]
Deterministic programs that single-assignment forbids

```plaintext
let \_ = put \_ 8 in
    let par \_ v = get \_
        \_ = put \_ 8
    in \_ v

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

let par \_ = insert \_ "1111"
    \_ = insert \_ "1100"
    in get \_
```
From Kahn process networks...

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboratory, Domaine de Voluceau, 78150
Boulogne-sur-mer, 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 schemata 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 system designers as to what are the best primitives for writing systems 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 $\delta$ shown in Fig. 1. The conventions are close to Algol and we only insist upon the new features. The program $\delta$ consists of a set of declarations and a body. Variables of type integer channel are declared at line (1), and for any simple channel we declare a channel. Then processes $f_0$ and $h_0$ are declared, much like parameters. Aside from usual parameters (passed by value in this example, here $c$ at line (3)), we can declare in the heading of the process how it is linked to other processes: at line (2) $f_0$ is stated to communicate via two input lines that can carry integers, and one similar output line.

The body of a process is an usual Algol program except for invocation of unit output line (e.g., at (4)) or send a variable on a line of compatible type (e.g., at (3)). The process starts blocked on an input line and, in the case of integer channel, it cannot communicate until it is unblocked. For any other process, but nothing can prevent a process from performing a send on a line.

In other words, processes communicate via first-in-first-out (fifo) queues. Calling instances of the processes are done in the body of the main program at line (5) where the actual names of the channels are bound to the formal parameters of the processes. The inf operator permutes the consequent activation of the processes.

Such a style of programming is close to any systems using EVENT mechanisms ([13,15,17,18]). A pictorial representation of the program is in the scheme $\delta$ in Fig. 2, where the nodes represent processes and the arcs communication channels between these processes.

What sort of things would we like to prove on a program like $\delta$? Firstly, that all processes in $\delta$ run forever. Secondly, more precisely, that $h_0$ prints out at line (3) an alternating sequence of $0$'s and $1$'s forever. Third, that if one of the processes were to stop at some time for an external reason, the whole system would stop.

The ability to state formally this kind of property of a parallel program and to prove it within a formal logical framework is the central motivation for the theoretical study of the next sections.

Kahn, 1974
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.
$f$ is monotonic iff, for a given $\le$, 

$$x \le y \implies f(x) \le f(y)$$
Concurrent Collections

Zoran Budimlić, Michael Burke, Vincent Carle, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jon Paleberg, David Peizotto, Vivek Sarkar, Frank Schliman, Sahak Tatarian

1 Rice University 2Intel Corporation 3UCLA

Abstract

We introduce the Concurrent Collections (CoC) programming model. CoC supports flexible continuations of task and data parallelism while retaining determinism. CoC is implicitly parallel, with the user providing high-level operations along with semantic ordering constraints that together form a CoC graph.

We formally describe the execution semantics of CoC and prove that the model guarantees determinate computation. We evaluate the performance of CoC implementations on several applications and show that CoC offers performance and scalability equivalent to or better than that offered by low-level parallel programming models.

1 Introduction

With multicore 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 seen widespread adoption. This is beginning to change. Systems like MapReduce are succeeding based on explicit parallelism. Other systems like Nvidia CUDA are partway there, providing a restricted programming model to the user but also exposing too many of the hardware details. The payoff for a high-level programming model is clear—it can provide semantic guarantees and can simplify the understanding, debugging, and testing of a parallel program.

In this paper we introduce the Concurrent Collections (CoC) programming model, built on past work on Streams [3]. CoC falls into the same family as dataflow and stream-processing languages—a program is a graph of kernels, communicating with one another. In CoC, these computations are called steps, and are related by control and data dependencies. CoC is provably deterministic. This limits CoC’s scope, but compared to its more narrow counterparts (StreamIT, NP-Click, etc.), CoC is suited for many applications—incorporating static and dynamic forms of task, data, loop, pipeline, and tree parallelism.

Truly mainstream parallelism will require reaching the large community of non-professional programmers—scientists, animators, and financial analysts—but reaching them requires a separation of concerns between application logic and parallel implementation. We say that the former is the concern of the domain expert and the latter of 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 CoC is that it is simultaneously a databrace-like parallel model
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 enables deterministic parallelism!
Parameterizing our language: LVars

IVar

Pair of IVars

Counter

0 1 2 ...
T
⊥

(⊥, 0) (⊥, 1) ... (0, ⊥) (1, ⊥) ...
(0, 0) (0, 1) ... (1, 0) (1, 1) ...

getFst "tripwire"
getSnd
Parameterizing our language: LVars

Pair of IVars
Parameterizing our language: LVars

Pair of IVars
Parameterizing our language: LVars

Pair of LVars

getSnd

"tripwire"

getFst
Parameterizing our language: LVars

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

getFst p △ get p {(n, ⊥) | n ∈ N}

Pair of IVars
Two take-aways

Monotonicity enables deterministic parallelism

Monotonically increasing writes
+ threshold reads
= deterministic parallelism
Where to find out more

- composition.al
  - Recent post: “How to read from an LVar: an illustrated guide”
- Our draft paper and tech report
  - Paper: Complete syntax and semantics
  - TR: Complete proof of determinism
- GitHub: github.com/iu-parfunc/lvars
  - Prototype LVar library in Haskell
  - Mechanized semantics in PLT Redex
- Talk to me!
Thanks!