A Lattice-Based Approach to Deterministic Parallelism

Lindsey Kuper and Ryan R. Newton
Indiana University

POPL 2013 Student Session
25 January 2013
What does this program evaluate to?

```
let _ = put l 3 in
let par v = get l
    _ = put l 4
    in v
```
Disallow multiple writes?

let _ = put l 3 in
  let par v = get l
    _ = put l 4
  in v
Disallow multiple writes?

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

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

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

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

Zoran Budimlić, Michael Burke, Vincent Carel, Kathleen Knobe, Geoff Lowney, Ryan Newton, Juan Palomino, David Pezutto, Vivek Sarkar, Frank Schlimbach, Sargor Tafazal

Abstract

We introduce the Concurrent Collections (CoC) programming model. CoC supports flexible combinations 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 deterministic computation. We evaluate the performance of CoC implementations in several applications and show that CoC offers performance and scalability superior 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 implicit 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 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 Threading [3]. CoC falls into the same family as datashare 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 dependences. 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.

Concurrent collections will require reaching the large community of non-professional programmers—scientists, artists, 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 dataflow-like parallel model...
From Concurrent Collections...

The key language feature that enables determinism is the single assignment condition. The single assignment condition guarantees monotonocity 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
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 schemes are given. We hope in this way to make a case for a more formal (e.g., mathematical) approach to the design of languages for system 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 in the sample program $S$ on fig. 1. The conventions are close to Algol and we only insist upon one new feature: the program $S$ consists of a set of declarations and a body. Variables of type Integar (channels) are declared at line 3), and for any simple integer-constant expression $k$, we declare a constant $x_{k}$ to be a channel. Then processes $p$ and $q$ are declared, much like procedures. Aside from local parameters (passed by value in this example, like $I$ at line 11), we can declare in the heading of the process how it is to be linked to other processes: at line 12, $z$ is said 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 insertions of unit and unit input lines (e.g. at (6)) or send a variable on a line of compatible type (e.g., at (11)). The process starts blocked on a unit output line, and the same for the input line. 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 (a done in the body of the main program at line 15) where the actual names of the channels are bound to the formal parameters of the processes. The index operator $p_i$ initiates the concurrent activation of the processes. A style of programming is close to any system using EVENT mechanisms ([1],[2],[3],[4],[5]). A pictorial representation of the program in the scheme $P$ on fig. 2, where the nodes represent processes and the arcs communication channels between these processes.

That sort of things would we like to prove on a program like $S$. Firstly, that all processes in $S$ run forever. Secondly, more precisely, that if prints out at line 15 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 erroneous reason, the whole system would stop.

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

1. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way: some autonomous computing stations are connected to each other in a network and then communicate exchanging information through these lines. A given station computes on data coming along its input lines,

**Fig. 1.** Sample parallel program $S$.  

**Fig. 2.** The schema $P$ for the program $S$.  

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.

Kahn, 1974
Monotonicity causes deterministic parallelism!
Parameterizing our language: LVars

0 1 2...
\( T \)
\( \perp \)

IVar

\((0, 0)\) \(\ldots\) \((0, 1)\)
\((1, 0)\) \((1, 1)\)
\((\perp, 0)\) \((\perp, 1)\)
\((0, \perp)\) \((1, \perp)\)
\(\perp\)

Pair of IVars

getFst "tripwire"
getSnd

Counter
Parameterizing our language: LVars

Pair of IVars
Parameterizing our language: LVars

Pair of LVars
Parameterizing our language: LVars

Pair of IVars

\( (0, 0), (0, 1), \ldots, (1, 0), (1, 1), \ldots \)
Parameterizing our language: LVars

let _ = put p {⊥, 4} in
let par v₁ = getFst p
  _ = put p {3, 4}
in ...v₁...
Parameterizing our language: LVars

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

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

Monotonicity causes deterministic parallelism

Monotonically increasing writes + threshold reads = deterministic parallelism
More in our TR

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

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