Contentious Data Structures

Leveraging Immutability for Parallelism

Sean Laguna

Oct 20, 2016

Intro

Motivation

- many programs not trivially parallelizable, but also not terribly complicated
- most tools either not accommodating too bare metal (synchronization primitives)
- solutions (Clojure Erlang, Chapel) too committal or invasive

Observations on Easing Concurrency

- understanding the relevant/slated operations helps
- data layout shadows algorithm via access pattern
- records help mediate between function and state, but incur cost, which quickly becomes prohibitive

Requirements for a Method

- less boilerplate than synchronization primitives
  - no learning a new series of pitfalls
- more flexible than "big data"-style solutions
  - handles data contention (in particular, races)
- efficient to the point of benefit
  - speedup over serial implementation on a 2-core machine?

Pinpointing the Task

In:

- description of stepwise or iterative task(s), algebraic properties of them
- data on which to operate, in a provided container
- number of desired threads

Out:

- concurrent execution of task(s) on that many threads

Needs of Scientific Programs

- handle large amounts of data with computational interdependence; scaling
- runtime options to match users needs; expressiveness
- small methodological changes should not require big design changes or manifest verification difficulties
- hardware multiplicity: distributed, manycore, gpu

Supporting Material

(Functionally) Persistent Data Structures

- not storage persistence, but uncanny similarities
- Okazaki, Hickey
- data structures don't have to be immutable to look immutable

Bit-partitioned Tries

- underlying structure that provides effect
- yields a vector or “hash map”

dependency graphs

Identify...

- opportunities for parallelism
- points that necessitate concurrency control

atomic operations

- lock-free MPMC work queue
- IDs for versioning container
- managing the interrelationships (much more later)

3 “views” of the method

1. theoreric (dependency graph and algebra)
2. memory-based (data layout and transformation)
3. operative (implementation and execution)

Related Efforts

- Clojure: reducers/transducers, etc; Erlang
- CITRUS; cuckoo hashing
- PRCU, RLU (read-copy-update successors); thread-local storage
- transactional memory
  - notably the identification and accommodation of pathological execution ordering

Inner Workings

Overview

- each thread gets an ID, and it's own snapshot of the data structure at each step
- each thread will perform its work with that snapshot as if the values were correct
- conflict detection and resolution will propagate any late updates, using knowledge of the partition and operations at hand to finalize all values

Remember... Persistent updates create new copies of the vector. So, they will look immutable. Transient updates will change the snapshot

The Contentious Vector

- transient vector with extra stuff

Test

```
#! /usr/bin/Rscript --vanilla
library(knitr)
library(rmarkdown)
file <- list.files(pattern='.Rmd')
rmarkdown::render(file)
```