Deterministic Parallel Computing in Managed-Code Languages

Motivation

Parallel programming introduces significant complexity in writing code due to race conditions and memory corruption. A common approach to avoid this is through mutual exclusion and locking, but this introduces additional issues such as the possibility of deadlock, resource starvation, and wasted cycles. Furthermore, the dearth of tools for debugging parallel code complicates the development process even further, since the results of program execution may not always be repeatable.

Determinator is a deterministic operating system being developed by the DeDis group at Yale. It guarantees that anything an application computes is exactly repeatable, and this can significantly reduce the complexity of writing and debugging parallel programs. Determinator does this through a deterministic parallel programming model known as "workspace consistency," and this project aims to apply the same principle to high-level language environments and specifically managed-code languages such as JavaScript, Java, and Python, where runtime information and type-safe execution facilities are available.

Background

Instead of operating system-based determinism, we want to utilize language-based determinism in parallel programs. Prior work has shown promising results, but in the context of purely functional languages i.e. parallel extensions for Haskell. In contrast, a majority of the applications today use object-oriented languages, so we will focus on those kinds of high-level languages.

Generally, a language allows concurrent processes to execute in separate address spaces as completely isolated instances. A process may spawn threads which all share an address space and may all modify shared memory. Problems occur when two threads modify the same part of memory, because determinism can’t be guaranteed if reads/writes conflict and the thread can see different states depending on non-serialized memory reads. A common solution to this problem is to use locks so that only one thread may access a resource at a time,
which guarantees order of access, but this puts the burden on the programmer to lock and unlock properly and manage the acquisition of multiple locks.

Processes operate on separate address spaces and don’t conflict with each other, but memory may be marked as “shared” to allow multiple processes to access and modify the same region. Our goal is to make processes retain their independence and isolation, but to diff-and-merge their separate heaps when they explicitly synchronize via fork/join, barriers, or message passing, to create the illusion of a shared address space.

To ensure determinism in this model, we will use workspace consistency as defined in the “Workspace Consistency: A Programming Model for Shared Memory Parallelism” paper. The basic principle is that data accesses can either be normal reads/writes or special “release/acquire” actions. “Release” is when a thread makes recent state changes available to other threads, whereas “acquire” is when other threads incorporate the state changes of other threads into their own state. Release/acquire occurs at points of synchronization, and writes propagate as slowly as possible, with conflicts left up to the implementation. In addition, we require that releases be paired with acquires so that each release transmits updates to a newly spawned process/thread.

One final aspect is that of heap encoding/decoding as JSON. The goal is that a Java thread should be able to spawn a Python process, and receive the result as a JSON representation of the heap, so that the Java thread can parse the heap for specific changes and incorporate them (diff-and-merge) to enable inter-language deterministic parallelism.

Timeline

1. Starting with Python, write a program that spawns multiple processes, has all of them do some computation (Black-Scholes, map-reduce type operations), and then serialize the heap and merge in the results using some heap-merging algorithm.

2. Using Java, spawn Python threads to do some computation, receive the results as a JSON object representing the heap, and merge in the results. Try the same with a Python process spawning Java threads.

3. Condense the JSON encoding so instead of the entire heap, we only have to pass back the parts where the process executed writes. Ideally this should be done without specific programmer annotation, or minimal annotation in the source code.

Deliverables

There will be plenty of source code to reflect any changes in the languages’ runtimes and also the code to implement workspace consistency at the language level. Each language will
require a heap-diff and heap-merge algorithm to incorporate changes from other processes. Additional code will depend on how heap encoding is done. There will also be a final report describing the implementation and algorithms used, as well as performance tests to compare deterministic parallel code to non-deterministic parallel code and to non-parallel code, run using standard benchmarks.

**Sources**

Amittai Aviram, Bryan Ford, and Yu Zhang:
“Workspace Consistency: A Programming Model for Shared Memory Parallelism”

Amittai Aviram, Bryan Ford:
“Deterministic OpenMP for Race-Free Parallelism”

Yu Zhang, Bryan Ford:
“A Virtual Memory Foundation for Scalable Deterministic Parallelism”

Amittai Aviram, Shu-Chun Weng, Sen Hu, Bryan Ford:
“Efficient System-Enforced Deterministic Parallelism”

Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve and Marc Snir:
“Parallel Programming Must Be Deterministic by Default”

H.W. Loidl et al.:
“Comparing Parallel Functional Languages: Programming and Performance”