COMS W4995 005 Parallel Functional Programming
Ashley Garcia (UNI: ag4647)

Project Overview and Motivation

A large part of my engineering experience revolves around version control and semantic reasoning about code, and my routine use of Git has made me acutely aware of the limitations of traditional line-based diffs. Git's object model uses Merkle-tree structure to detect changes by hashing the contents of files and directories. Git is very fundamentally textual, it cannot detect whether a program's structure changed, only if the text has changed. I think that for many modern workflows (agentic lol), understanding structural changes in code is far more meaningful than tracking raw text edits. Semantic diffs are very computationally expensive though because they operate over entire AST structures. I think it's a good subject for parallelism because in this problem/solution, different subtrees are independent, hash computations are pure and bottom-up, and the task of diffing distinct child pairs is a divide-and-conquer operation.

With that in mind, I wanted to work on a parallel AST-based semantic diff engine for a small, expression-oriented toy programming language. Instead of comparing text, the system will parse two program versions (taking in two input files) into Abstract Syntax Trees (ASTs), convert them into Merkle trees, and compute differences by comparing subtree hashes in parallel. Subtrees with identical hashes can be pruned immediately; those that differ can be diffed recursively, with child comparisons performed concurrently.

Objective

The goal of this project is to implement a compact, well-scoped semantic diff tool that highlights structural differences between two programs in a toy language. The final deliverable will parse both programs into ASTs, hash every subtree, and produce a structured summary of which AST subtrees changed. The project focuses specifically on how parallel hashing and parallel recursive diffing can reduce runtime.

Toy Language Specification

A concern I have is the language. I have to build a parser which can be a lot of work, so, I have to get a balanced language, not too rich and not too simple. If my toy language is too rich, hashing code and diff logic become complicated, error messages become confusing, and debugging will be awful, but if the toy language is too simple, my ASTs are tiny and I don't get interesting parallel behavior and speedup numbers look trivial. My goal is to have something that is expressive enough to generate interesting tree shapes (nested expressions, conditionals, small let-bindings) but not so expressive that I accidentally end up re-implementing half a compiler or drowning in edge cases. Something in the space of a small functional core calculus feels appropriate: integers, variables, binary operations, if expressions, a let construct.

Sequential Baseline

Once the language is defined, the sequential baseline is pretty straightforward:

1. Parse both program versions into ASTs using Megaparsec
2. Compute Merkle hashes bottom-up (constructor + literal + children)
3. Diff the trees structurally - if two nodes have identical hashes, prune; otherwise recursively compare children

COMS W4995 005 Parallel Functional Programming
Ashley Garcia (UNI: ag4647)

Other Stuff

Hashing child nodes is embarrassingly parallel* and subtree diffs form a divide-and-conquer recursion tree where each node can initiate individual work so parallelism fits pretty well. My parallelization strategy focuses on two main components:

1. Parallel Merkle Hashing: Each node's children can be hashed using strategies like `childHashes 'using' parList rdeeseq`
2. Parallel structural diff: when two nodes differ, corresponding child pairs can be diffed concurrently
   `parMap rdeepseq (uncurry diff) (zip cs1 cs2)`

Loads won't be perfectly uniform due to the irregular structure of ASTs, so I'll also experiment with different evaluation strategies (parList, parBuffer, chunking) to handle unbalanced recursion. I also think this will produce non-trivial performance behavior because some diffs will prune early, others will branch heavily, and performance may depend strongly on AST shape, so I'll get to benchmark meaningfully.

I'm intentionally keeping the scope narrow so I can highlight the parallel parts clearly, and if I'm short on time, I can always scale down the grammar or cut optional features without compromising the integrity of the parallel algorithm.

Evaluation

For evaluation, I'll generate synthetic ASTs of varying sizes (balanced and unbalanced) in addition to parsing real toy programs. I'll benchmark:

- hashing alone
- diffing alone
- full end-to-end comparison

and run everything on 1, 2, 4, and 8 cores to measure speedup and estimate the parallel fraction via Amdahl's Law. This should give me enough data to talk about irregular parallelism, granularity, overhead, and why certain shapes of ASTs parallelize better than others.

I'll be running all benchmarks on my laptop - MacBook Pro with an M4 chip (10 cores, 10 hardware threads).

*I thought this was funny