Lab 3: BST Comparison:
Performance analysis and detailed approach

Qiying Wu

October 27, 2024

Abstract

In this lab, we explore the concurrent computation of binary search tree (BST) hashes and tree comparison using Go's concurrency tools, specifically goroutines, channels, and synchronization primitives. The goal is to optimize the processes of hashing, identifying duplicate hash groups, and performing pairwise tree comparisons to achieve high efficiency while maintaining accuracy. Each BST is represented by a unique hash generated from an in-order traversal using a custom hashing function. Multiple configurations of worker threads handle hash computation and map updates, providing insights into the impact of parallelism on performance. This report evaluates the scalability of different parallelization approaches across several worker configurations, with a focus on balancing computational and synchronization overhead.

Introduction

Binary Search Trees (BSTs) are a fundamental data structure in computer science, used extensively in applications requiring efficient data retrieval and organization. This lab focuses on implementing a parallelized approach to compare BSTs using a custom hash-based methodology. Given a set of BSTs, we aim to compute a unique hash for each tree using an in-order traversal, group trees with identical hashes, and further compare these groups to identify structurally identical trees. Leveraging Gos concurrency model, the implementation employs goroutines and channels for inter-process communication and synchronization mechanisms such as mutexes and semaphores to ensure data consistency during concurrent hash computations and updates.

The lab is structured in three parts:

- **Hash Computation**: We compute the hash of each BST in parallel by spawning a designated number of worker goroutines, based on the -hash-workers flag. This step evaluates the efficiency of parallel hash computation and measures the time taken to generate hashes for all BSTs.

- **Hash Grouping**: Trees with identical hashes are grouped to identify potential duplicates. Different configurations of data worker goroutines are managed by the -data-workers flag to control access to the map holding hash groups. This part evaluates synchronization overhead and seeks to balance hash computation with data aggregation.

- **Tree Comparison**: For each group with duplicate hashes, tree comparisons are performed in parallel using -comp-workers. This step refines the grouping by identifying structurally identical trees, based on a pairwise comparison strategy.

  The experimental setup uses multiple input files with varying BST complexities to analyze the performance of each concurrent approach. The results are assessed for both correctness and performance under different configurations, providing insights into how concurrency levels and synchronization impact the speed and scalability of BST hashing and comparison.

This report presents the design and implementation of these different approaches, along with performance analyses based on their execution times for various datasets. We also discuss the challenges encountered during the optimization process, including memory management and non-deterministic behavior caused by atomic operations in CUDA. By comparing the performance of these implementations, we aim to highlight the trade-offs between different levels of abstraction in GPU programming and the impact of hardware-level optimizations on the performance of parallel algorithms like KMeans.

Background and Hashing Methodology

0.1 Binary Search Tree Structure

0.2 Hash Computation Method

Implementation

0.3 Hash Computation

0.3.1 Sequential Hash Computation

0.3.2 Parallel Hash Computation

0.4 Hash Grouping

0.4.1 Sequential Hash Grouping

0.4.2 Parallel Hash Grouping

0.5 Tree Comparison

0.5.1 Sequential Tree Comparison

0.5.2 Parallel Tree Comparison

Experimental Setup

0.6 Input Data and Flags

0.7 Performance Metrics

1 Results and Analysis

1.1 Performance Analysis

1.1.1 Hash Computation Time

1.1.2 Hash Grouping Time

1.1.3 Tree Comparison Time

1.2 Correctness Verification

Discussion

Conclusion

2 *Performance Analysis

Speedup Graphs

I will be using the total execution time for analysis, here's a cleaned up table for the total execution time without iteration data, and I will use threshold 1e-5.

References

Technical Sources:

- Atomic Add based on atomicCAS()
- atomicAdd() for double on GPU