PFP Project Proposal: parallel-sorting

Xinchen Zhang (UNI: xz3052)

11/17/2025

Project Overview

In this project, I will implement both sequential and parallel versions of two classical divide-and-conquer sorting algorithms: **Merge Sort** and **QuickSort**. The goal is to compare their performance under different parallel evaluation strategies in Haskell and analyze their scaling behavior across multiple cores.

Merge Sort recursively divides the input list and merges the sorted halves, while QuickSort partitions the list around a pivot and recursively sorts each partition. These algorithms provide rich opportunities for parallelism because the subproblems created during recursion are largely independent.

The parallel versions will explore several strategies, including spark-based divide-and-conquer parallelism, `parList`, `parBuffer`, and controlled recursion depth. The sequential versions will serve as baselines, allowing a detailed comparison of runtime, speedup, and memory behavior.

Motivation

Merge Sort and QuickSort are both classical divide-and-conquer algorithms whose recursive structure creates independent subproblems. These subproblems can be evaluated concurrently, providing a natural foundation for exploring the parallel constructs introduced in this course, such as `par`, `pseq`, `parList`, and `parBuffer`. Their algorithmic structure allows detailed observation of how different parallel strategies affect performance.

Also, sorting offers a simple and robust scaling mechanism through the size of the input list. This allows experiments that avoid extremely short runtimes, where measurement noise dominates, as well as excessively long runtimes that are impractical for repeated tests. Additionally, different input distributions (random, sorted, reverse-sorted) produce distinct computational patterns, especially for QuickSort, resulting in more diverse and informative performance evaluations.

Finally, sorting is a compute-bound problem with minimal I/O. The performance results therefore reflect genuine parallel execution characteristics rather than external bottlenecks

such as disk access or heavy memory traffic. Because these algorithms are well-understood, verifying correctness is straightforward, enabling the project to focus on parallel performance rather than functional correctness.

Expected Result

A final report including:

1. Detailed explanation of all algorithm variants, including sequential baselines, optimized versions, and multiple parallel implementations.
2. A description of the parallelization strategies used, including spark creation, granularity control, recursion depth selection, and evaluation strategy choices.
3. Performance tables summarizing execution time across input sizes, thread counts, and repeated runs.
4. Memory and garbage collection statistics, including maximum residency, Gen0/Gen1 counts, GC time, and the impact of varying nursery sizes.
5. ThreadScope visualizations and analysis of spark behavior, load balance, idle times, garbage collection pauses, and anomalies at specific thread counts.
6. Discussion of spark granularity and task size, identifying under-parallelization vs. over-sparking and determining optimal depth thresholds.
7. Clear final conclusions summarizing:
    - which parallel strategy performs best and why,
    - which algorithm scales better under which conditions,
    - lessons learned about the Haskell runtime system.

Proposed Timeline

1. [**11/18 – 11/24**]: Implement sequential baselines.
    - Implement basic Merge Sort and QuickSort using Haskell.
    - Verify correctness on multiple input sizes.
    - Add optimized sequential versions with improved strictness and merging.
2. [**11/25 – 11/29**]: Implement initial parallel versions.
    - Add depth-1 parallel divide-and-conquer using `par` and `pseq`.
    - Implement `parList` and chunk-based parallel sorting.
    - Begin preliminary tuning of spark granularity.

3. [**11/30 – 12/03**]: Advanced parallel strategies and tuning.
    - Implement depth-k parallel recursion for granularity control.
    - Add a `parBuffer` version to study bounded parallelism.
    - Run initial experiments to identify bottlenecks.
    - Collect early ThreadScope traces.
4. [**12/04 – 12/05**]: Final experiments and report drafting.
    - Run full experiment suite across input sizes and thread counts.
    - Generate performance tables, speedup graphs, and Amdahl curves.
    - Produce ThreadScope visualizations and write analysis.
    - Finalize conclusions and complete the final report.