

cliquepar: Project Proposal

Sammy Abdella (sn3136)

COMS W4995: Parallel Functional Programming

What I'll build. I'll write a Haskell program that finds a maximum clique in a graph. A clique is a group of vertices where every pair has an edge between them. The program will print one of the largest cliques and its size. I'll use undirected graphs with no edges from a vertex to itself and no repeated edges. My goal is to test when splitting this search across CPU cores makes it faster.

How the search will work. I'll start with a recursive search that keeps the current clique and a list of possible next vertices. It picks a vertex and tries two choices: include it or skip it. Both choices remove that vertex from the remaining list. The include choice also keeps only its neighbors, so every remaining vertex still connects to every vertex in the current clique. When no candidates remain, the search returns the current clique. Comparing the two choices gives the largest result. I'll then add a rule that skips a branch if the current size plus the number of remaining candidates can't beat the best size found in that task. This saves work without losing a larger clique.

How I'll make it parallel. Each vertex will start a search task that includes it and considers only its neighbors with larger vertex numbers. Each nonempty clique belongs to exactly one task, based on its smallest vertex. I'll first run these tasks one at a time with `map`. Then I'll use `parMap rdeepseq` from `Control.Parallel.Strategies` to let the runtime evaluate tasks in parallel and fully evaluate their results. Both versions will use the same tasks, search order, and rule for skipping branches. Each task will keep its own best result, and I'll choose the largest result at the end. Recursion inside each task will stay sequential.

Input data and checks. I'll create text files with the vertex count on the first line and one edge per remaining line, using vertex numbers from 1 to that count. I'll test a graph with zero vertices, which should return size 0. A graph with vertices but no edges should return size 1. A triangle should return size 3. I'll also test complete graphs and graphs with separate parts. On small graphs, I'll check the answer by trying every subset. For every result, I'll check that its vertices are distinct and every pair shares an edge.

I'll also generate random graphs, starting with 15 to 25 vertices. For each pair of vertices, I'll add an edge with a chosen probability, such as 0.2, 0.5, or 0.8. I'll save several graphs for each size and probability so both versions use identical inputs. I'll increase sizes gradually, since the number of possible vertex subsets grows quickly.

How I'll compare running times. I'll compile with optimization and threaded runtime support, then compare the parallel version with one, two, and four runtime workers where the computer supports them. I'll also time the sequential version on one worker. I'll load and fully evaluate each graph before timing, and fully evaluate the result before stopping the timer. I'll run each setting five times in separate program runs and report the median elapsed time. I'll show how the parallel version scales from one worker and how it compares with the sequential version.

Work plan and limits. I'll finish the sequential search and correctness checks first, add the branch skipping rule, then add parallel tasks and measure performance. Small tasks may cost more to schedule than they save, and one large task may keep the other cores waiting. If needed, I'll try splitting the first two search levels into tasks and use that same split in both versions. I'll report slowdowns as well as speedups and include the saved inputs and run instructions.