Project Proposal

COMS 4995 Parallel Functional Programming
Yixuan Li(yl3803), Jiaqian Li(cl4283), Phoebe Wang(kw3036)

We have two potential approaches for our project, and we would greatly appreciate your feedback on which one would be more suitable for us to pursue:

- **Traveling Salesperson Problem (TSP)**
- **Boolean Satisfiability Problem Solver (SAT Solver)**

1 Option 1: Traveling Salesperson Problem (TSP)

1.1 Algorithm Description

The Traveling Salesperson Problem (TSP) is an NP-hard optimization problem. Given a list of cities and the cost of traveling between each pair of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting point.

Our project will implement two parallel solutions to TSP: **Brute Force Parallelization** for finding the exact optimal solution and the **Divide-and-Conquer Strategy** designed to provide an approximate solution for large-scale TSP problems.

1.2 Brute Force Parallelization

- Explore all permutations of city ordering to determine the shortest path.
  - **Parallelism**: Evaluate each permutation concurrently to calculate the total distance.

1.3 Divide-and-Conquer Strategy

- Divide the set of cities into smaller subsets.
  - Random Partitioning - Randomly assign cities to different subsets.
  - Proximity-Based Partitioning -
    - Nearest Neighbor Expansion: Start from a random city and grow subsets by adding the nearest unvisited cities until the subset reaches a desired size.
    - **Parallelism**: Select multiple starting cities and grow each subset independently.
    - K-Means Clustering with parallelism taught in class.
- Solve TSP for each subset independently.
  - **Parallelism**: Each subset's permutation will be calculated concurrently.
- Combine the solutions by connecting the closest pair of cities between subsets.
  - **Parallelism**: Compute all distances between all pairs of points from the two subsets in parallel to select the smallest distance.

Additionally, if time allows, we plan to experiment with other algorithms that guarantee to find the optimal solution but are computationally expensive, such as the Branch and Bound Method, and Dynamic Programming (Held-Karp Algorithm).

1.4 Parallelization Plan

Our parallelization plan will use Haskell's `Control.Parallel.Strategies` library:

- Brute Force Parallelization
  - Generate all permutations of cities and calculate the total distance for each route.
  - Use `parMap` and `rpar` to evaluate the distance for each permutation in parallel.
- Divide-and-Conquer Strategy

- Parallelized partitioning techniques like K-Means clustering and Nearest Neighbor Expansion using `parMap` and `rpar`.
- Solve TSP for each group in parallel using the brute force method above.
- Parallelize distance calculations for merging subsets using `parMap` and `rpar`.

1.5 Input Data

Our program will use a distance matrix to represent the input. The matrix will define pairwise distances between cities, where distance$[i][j]$ is the distance from city i to city j. The program will support reading matrices from an external file.

Datasets:

- We will use publicly available datasets from TSP Library for testing and benchmarking.
- Initial testing will involve a small dataset with 5 cities.
- We will later expand to a larger dataset with 48 cities (US state capitals).

1.6 Reference

[1] Traveling salesman problem - Wikipedia
[2] Algorithms for the Traveling Salesman Problem
[3] Ishikawa, Kazunori, et al. "Solving for Large-Scale Traveling Salesman Problem with Divide-and-Conquer." *SCIS & ISIS 2010*, 8-12 Dec. 2010, Okayama Convention Center, Okayama, Japan.

2 Option 2: Boolean Satisfiability Problem Solver (SAT Solver)

2.1 Algorithm Description

The **Boolean Satisfiability Problem (SAT)** is a fundamental decision problem. Given a Boolean formula in conjunctive normal form (**CNF**, **AND** of **OR**s), our objective is to determine if there exists an assignment of truth values to variables that make the formula evaluate to true. By the *Cook-Levin theorem*, (circuit)-SAT, and consequently SAT, is proven to be NP-complete, which means that we do not expect an efficient algorithm to solve it in the worst-case scenario. However, in many practical applications, SAT instances often exhibit structural properties or redundancies that make them more tractable for modern solvers.

Modern SAT solvers build on DPLL by incorporating Conflict-Driven Clause Learning (CDCL). The high-level idea of the algorithm is shown below:

Algorithm 1 CDCL SAT solver (φ)

```
1:  dl ← 0; conflicts ← 0; answer ← UNKNOWN
2:  if (UnitPropagation(φ) = CONFLICT) then
3:    answer ← UNSATISFIABLE
4:  while (answer = UNKNOWN) do
5:    dl ← 0;
6:    while (conflicts < limit ∧ answer = UNKNOWN) do
7:      if (AllVariablesAssigned(φ)) then
8:        answer ← SATISFIABLE
9:      else
10:       dl ← dl + 1
11:       AssignBranchingVariable(φ)
12:       while (UnitPropagation(φ) = CONFLICT ∧ answer = UNKNOWN) do
13:         β ← ConflictAnalysis(φ)
14:         if (β < 0) then
15:           answer ← UNSATISFIABLE
16:         else
17:           Backtrack(φ,β)
18:           dl ← β
19:           conflicts ← conflicts + 1
20:   Restart(φ, limit)
21: return answer
```

Our objective is to build a parallel CDCL SAT solver.

2.2 Parallelization Plan

The key idea behind a parallel SAT solver is to **partition the search space effectively** and **manage workloads dynamically to address imbalances**. Solving a SAT instance involves exploring a large search space defined by all possible variable assignments. Directly dividing this space can lead to imbalanced workloads.

Initially, the search space is divided into guiding paths, which are sets of variable assignments that restrict the solver to specific disjoint subspaces. To handle imbalances, we use **dynamic work allocation** with **task splitting** and **work stealing**:

- **Task Splitting**: If the initial work pool is empty but some slave processes are still active, the master requests an active slave to split its current workload. The slave then divides its subspace into smaller parts using heuristics (e.g., unassigned variables or branching points).
- **Work Stealing**: Idle slave processes can steal tasks from the work pool or request tasks directly from active slaves.

To implement a parallel SAT solver in Haskell, several libraries are essential for managing concurrency, parallelism, and data structures.

- `Control.Concurrent` and `Control.Concurrent.Async`: provide lightweight threads and synchronization tools, such as `MVar`, which are used to create a shared work pool for task allocation and dynamic work stealing. They also offer high-level abstractions like `async` and `mapConcurrently`, which enable parallel execution of slave processes and efficient management of their results.
- `Control.Parallel` and `Control.Parallel.Strategies`: these libraries support parallel computations by providing evaluation strategies like `parListChunk`, which can be used to partition and evaluate tasks concurrently.

2.3 Input Data

To test a SAT solver, small handcrafted **CNF** formulas (5–10 variables) can be used for debugging. We also generate larger random k**-SAT** instances near the phase transition for performance testing. Additionally, datasets from SAT competitions and real-world applications can benchmark the solver's practical performance.

SAT solvers use the **DIMACS CNF** format. The first line specifies `p cnf <num_variables> <num_clauses>`, followed by clauses with literals as integers (positive for true, negative for false), each ending with 0. For example, the formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$ is represented as:

```
p cnf 3 2
1 2 0
-1 3 0
```

2.4 Reference

[1] Martins, R., Manquinho, V., & Lynce, I. (2012). An overview of parallel SAT solving. *Constraints, 17(3),* 304–347. Springer.
[2] Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM, 5(7),* 394–397. ACM New York, NY, USA.
[3] SATLIB - Benchmark Problems