COMS 4995 Project Proposal

Parallelized Particle Swarm Optimization

Xijiao Li (xl2950)
Chen Chen (cc4351)

November 22, 2021

Goals

This project's main goal is to implement a parallelized Particle Swarm Optimization algorithm in Haskell. Its implementation will be used to solve two-dimensional optimization problems; for example, to find the local minima of the the Shekel function at (0.5,0.5) captured in Figure 1.

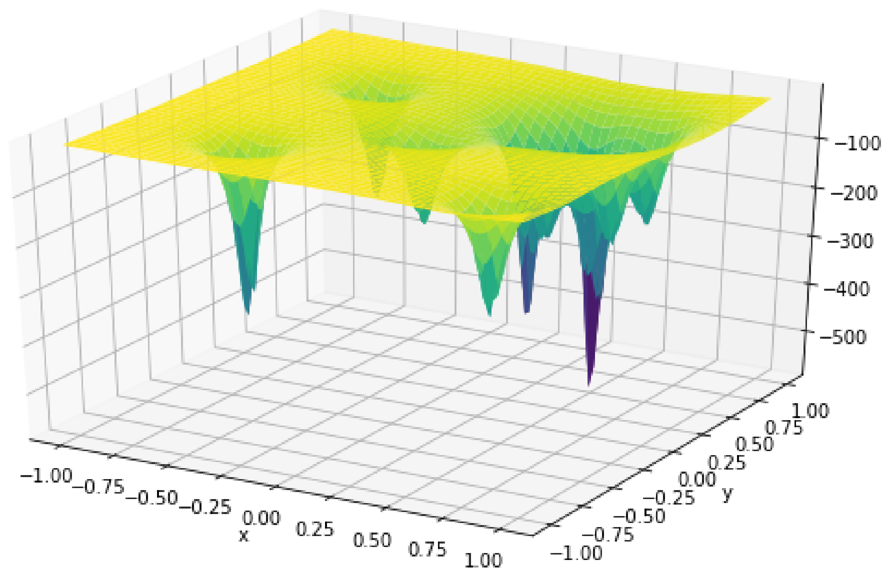

Figure 1: Shekel function with 10 minima

Background

Particle Swarm Optimization (PSO) is a population-based stochastic optimization technique. There are a number of candidates (or particles) which move through the search space in search of the best solution. Every particle position represents a potential solution and the goodness/fitness of that solution is measured by an objective function (the function being optimized).

The algorithm does not require the objective function to be differentiable, so it is applicable to a wide range of optimization problems. It is used in portfolio optimization, heat transfer maximization of energy system, and many other areas.

Upon initialization, each candidate is randomly assigned to a random position and with a random initial velocity. At each time step, every candidate first updates its velocity and then the position:

$$
\begin{aligned}
V_{i,t+1} &= wV_{i,t} + c_1 r_1 (O_{i,t} - P_{i,t}) + c_2 r_2 (O_t - P_{i,t}) \\
P_{i,t+1} &= P_{i,t} + V_{i,t+1}
\end{aligned}
$$

where

- $V_{i,t}$:= the velocity of candidate i at time t
- $P_{i,t}$:= the position of candidate i at time t
- $O_{i,t}$:= the position with min cost ever visited by candidate i at time t
- O_t := the position with min cost ever visited by any candidate at time t
- r_1, r_2, w, c_1, c_2: different weight parameters for each term

One can see from the equation that the movement of the candidates is governed by three factors: the *inertia weight* component ($V_{i,t}$), the *cognitive* component ($O_{i,t} - P_{i,t}$) and the *social* component ($O_t - P_{i,t}$). The inertia weight component allows a candidate to maintain some momentum between iterations. The cognitive component allows the candidate's movement to be influenced by its memory of good positions that it has found in earlier iterations. The social component will cause the good positions found by other candidates of the swarm to influence the given candidate's movement.

Overall the logics of PSO can be found in the pseudo code:

Algorithm 1: PSO algorithm

```
1  begin
2  |  Initialize N candidates on the search space;
3  |  for time step t ← 0 to max_iter do
   |  |  // update all candidates
4  |  |  for every particle i do
5  |  |  |  Update velocity to V_{i,t};
6  |  |  |  Make next movement to position to P_{i,t};
7  |  |  |  Calculate function value at its current position P_{i,t};
8  |  |  |  (Potentially) update O_{i,t} and O_t;
9  |  |  end
10 |  end
11 end
```

The sequential PSO will yield $O(m * n)$ runtime, where m is the total number of iterations and n is the number of candidates.

Design

The project design has three parts: baseline implementation, parallel implementation, and benchmarking.

The baseline implementation would follow the unoptimized, sequential implementation of PSO algorithm. In the conventional imperative approach, the state of the candidates is stored as an array of candidate objects, and at each time step, we traverse through the array and update each candidate's velocity and position. In Haskell, the state of candidates will be stored in the `Candidate` datatype, and the traversal will be done using recursion. The update will not incur any side-effect, since it only depends on the results and statistics from the last time stamp, which must have been determined at that point.

We will parallelize the computation in the `update all candidates` step. Specifically, when updating candidate states, we will splitting the candidates into groups and update them in parallel. Assuming that the number of candidates will be much bigger than the number of CPU threads, we also plan to explore different strategies to split up the workloads to achieve better load balancing.

For the benchmarking part, we plan to evaluate the performance of our baseline and parallelized implementation in terms of runtime and CPU utilization when given different cost functions. We also plan to explore the scalability of our parallelized implementation with number of cores.