Parallel Genetic Algorithm for 8-Queen Problem

Zhangyi Pan(zp2223), Shuhong Zhang(sz2949)

1. 8-Queen Problem
   8-Queen problem is a simple and popular game for AI agents. In 8-Queen game, we have a 8*8 board and 8 queens that can be placed in any grid on the board. However, we assume the queens can attack each other from vertical, horizontal and diagonal direction. As a peace lover, we want to place the queens in places such that none of them can attack each other, which means we will have only 1 queen for each line and each row, and no two queens can lie on the same diagonal line.

2. Problem Setting
   Now we know what an 8-Queen game is, and we will need to formulate a setting for an AI solver. We can formulate the game as following:
   - State: We will have some placement of the 8 queens on board
   - Transition: We will generate a new state with different placement of queens
   - Fitness function: We want to define a function which can tell us how good our current state is. Here we define the fitness function to be the number of non-attacking queens pairs. That is, for our ideal state, in which none of the queens are attacking each other, the fitness function should be f=7+6+5+4+3+2+1 = 28.
   - Goal test: In AI problems, we have to know whether we have reached our goal. In our problem, we know we have reached an end if we have our fitness function f(cur_state) = 28

3. Genetic Algorithm
   In this project, we want to use Genetic Algorithm for parallelization. Genetic Algorithm is an algorithm inspired by Natural Selection. In a genetic algorithm, we first generate a population of states, and generate the next generation of states, by randomly selecting parent states and combining them to generate a successor. The selection of parent states is usually based on some probability. In our case, we want the probability to be proportional to our fitness function, thus we are more likely to select parents closer to a goal. To combine the parents, we randomly pick a crossover point and then combine the parents over the cross point. Also, in order to generate new samples, we want the successor to “mutate”, that means for each element of the state, we have a small probability for it to change to some other value. In our game, the algorithm will terminate if we have a goal state in the population.

4. Genetic Algorithm for Parallelization
   There are two different ways to do parallelization. The former one is to parallelize combination and mutation. Crossover is a genetic operator for combination. Typical data structures, such as vectors, binary trees, can do crossover. For simplicity, we take k-point crossover as an example. It picks k points from both parents' chromosomes and switches the points respectively. After switching, two offspring are generated. Mutation is

a genetic operator for diversity. From one generation to the next one, it is possible to mutate on some points of its chromosomes. In data structure, it leads to flipping on some positions of data. Since mutations on different positions or chromosomes are individual to each other, parallelization for mutations is practically impossible.
Another way is called parallel Genetic Algorithm, in which we have several independent settings and let them evolve locally without sharing with each other. After finishing the evolving process for each individual setting, we combine their results as a new generation and continue performing Genetic Algorithm with the bagged generation to generate a final result.

5. Parallelization Measurement
   Due to the convenience of Haskell's runtime measurement tool, we can measure the overall runtime and activity of total cores as well as every single core directly. We plan to compare the strategies among non-parallelization, parallelization-on-crossover, parallelization-on-mutation, parallelization-on-both. Then calculate speedup for the last three strategies with respect to the first one.