Parallel Functional Programming
Final Project Report
Group members: Fernando Notari, Maximo Jalife

A Parallel Traveling Salesman Problem Solver with Genetic Algorithms

Problem Overview

The objective of our project is to come up with good candidate solutions to the Traveling Salesman Problem (TSP). The problem statement is:

"*Given a set of N cities and a unit measure of distance, what is the shortest possible path which visits all N cities and returns to the starting point?*"

It is easy to come up with a random candidate solution to the question: start at a random city. If the next city has been visited, skip it. Else, visit it. When all cities have been visited, return to the beginning. The problem comes when verifying if this generated solution is the shortest possible path, as we would have to traverse all others to know for sure. There are $\frac{(N-1)!}{2}$ possible paths (as some paths are rotations of others), but since our proposed solver does not check if two paths are equal when rotated we will effectively be dealing with $N!$ possibilities.

Genetic Algorithm Approach

In order to create candidate solutions to the TSP, we will employ a genetic algorithm (GA). As a class of algorithms, GAs are inspired by natural selection: they take as input a population, evolve it over the course of multiple generations, and therefore output a fitter population than the initial. In our case, an individual will be a tour through the cities and therefore a candidate solution. There are many parameters which may be tuned in the algorithm:

- The number of generations/terminating conditions: in our case we will use a fixed number of generations. A possible terminating condition would be stopping if improvement plateaus.
- The number of crossovers per generation: this can be a fixed number or vary according to population size.
- The crossover function: we made use of an ordered crossover[1] in order to ensure each city is only visited once.
- The mutation probability: this parameter may vary from 1 to 100 and is normally kept low to preserve the best aspects of the prior generation. It helps the algorithm escape local maximums.

[1] https://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf

- The mutation function: our project randomly swaps two cities in an offspring before in order to mutate it. This function may be changed in the process of tuning it.
- The size of the initial population: we let the initial population equal the number of cities, though in general the larger this size the better (pending hardware considerations).
- The m

Observations: our algorithm incorporates elitism: we only randomly cross the top 10% of the population in terms of fitness. We also keep all individuals in each generation to further expand the search space.

Logistical considerations

In order to assess the quality of our results, we used only city maps from TSPLIB[2] made available by Heidelberg University. We read in the cities from the problem file and compare it against their provided best solution.

Parallelization Attempts

We attempted to parallelize this algorithm in two separate instances: first when performing the crossovers, and second when calculating fitness. While the former achieved a negligible improvement in runtime (and core management efficiency), the former allowed the program to speed up significantly. The more cores used in running the program, the faster it became. This speaks to its scalability; the program benefits from increased computational power.

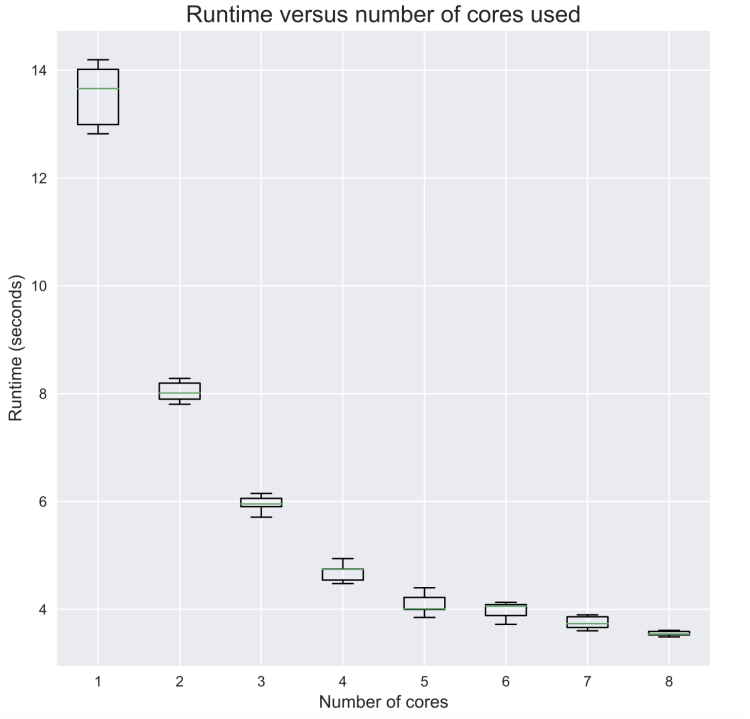

FIGURE 1: Runtime versus number of cores

[2] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

This result was also true for different maps. The graph above is for one such map with 280 cities, and we can see the same pattern in different sized maps:

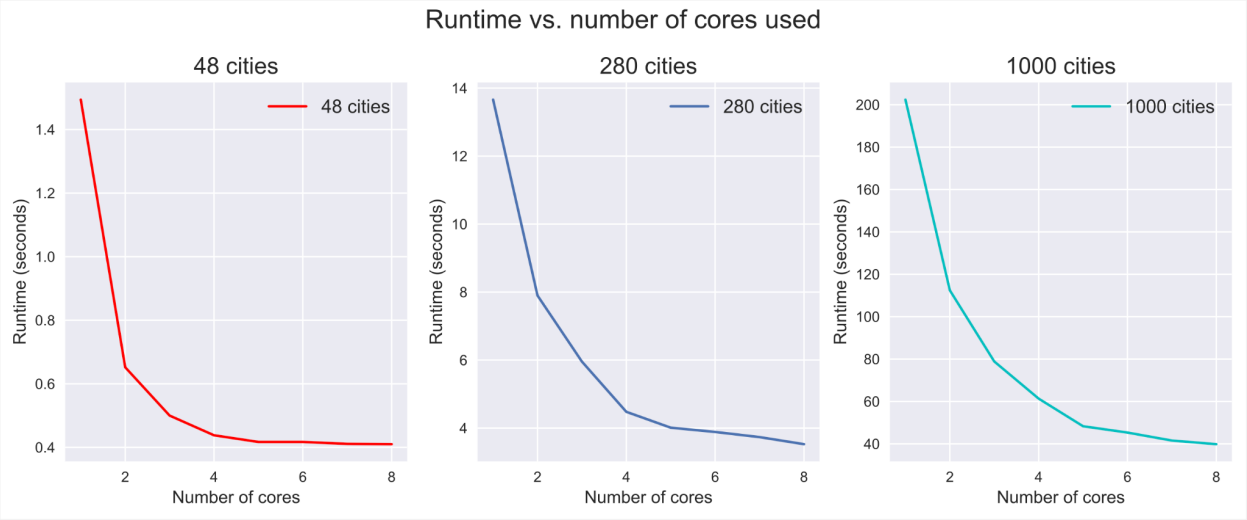

FIGURE 2: Runtime versus number of cores for different maps (different plots used due to y-axis scale)

The same trend is visible across all maps (approaching the function $\frac{1}{x}$). As the number of cities increases the decrease in runtime takes longer to plateau, meaning larger problems may benefit from stronger computing hardware. The resulting threadscope visualization of the *eventLog* file back these results as all cores were used in a balanced manner and each additional core was used:

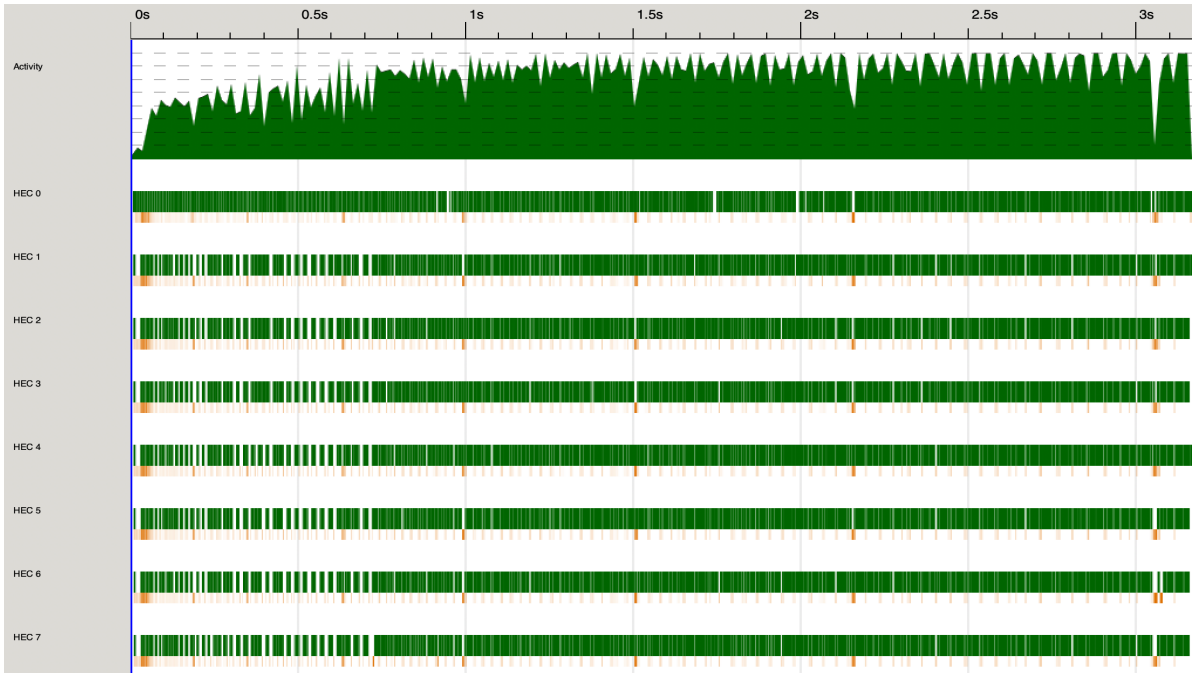

FIGURE 3: Threadscope visualization of core usage. Increased activity later in the runtime is a product of the increased population size.

Speedup Analysis

The best theoretical speedup of a parallelized program is dictated by Amdahl's Law, which states:

$$Speedup = \frac{1}{1-P+\frac{P}{N}}$$

Where P is the percentage of the program which may be parallelized. We compared the results we obtained from our parallelized program for different maps against the theoretical speedup our program could have achieved in the case that 95% of it could be parallelized:

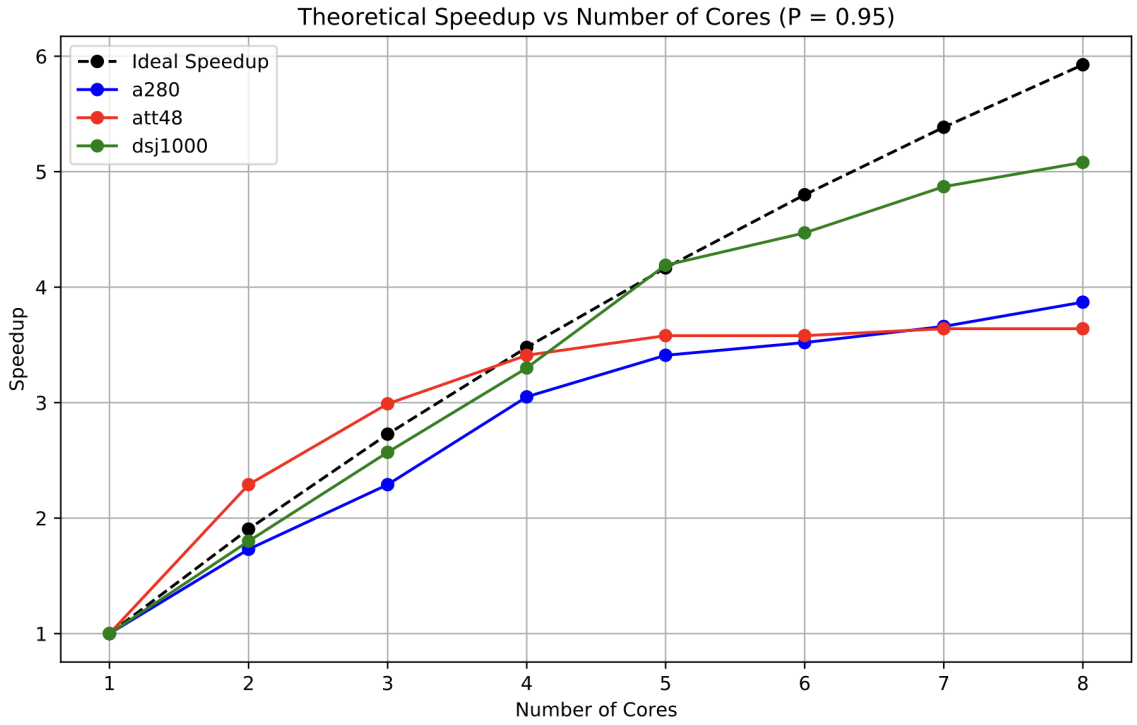

FIGURE 4: Actual speedup versus ideal speedup

The first thing to note is that the att48 map (with 48 cities) surpassed the ideal speedup, which is theoretically. This is likely due to possible hardware considerations and also possibly due to our P estimate being incorrect. It is also important to note that for all maps, the more cores added meant the farther we were from the ideal speedup. Furthermore our program appears to work better for maps of increased sizes, as evidenced by the fact that the larger the map the higher its maximum speedup was.

In order to estimate the percentage of our program which currently run in parallel, we assumed our speedup for 2 cores for the dsj1000 map was ideal and solved for P using Amdahl's Law:

$$S = \frac{1}{1-P+\frac{P}{N}} \Rightarrow 1 = S(1-P) + \frac{SP}{N} \Rightarrow P = \frac{S-1}{S-\frac{S}{N}}$$

Using the formula above and our achieved speedup (of 1.8x), we calculated that our estimate of the percentage of our program which runs in parallel is P = 0.89.

CODE APPENDIX

```
{-
COMPILE THE CODE: stack --resolver lts-21.9 ghc -- -O2 -threaded -rtsopts --make -Wall -O tsp

USAGE: ./tsp [problem_file] +RTS -N8 -ls
[problem_file] format:
 - Each line represents a city
 - Each line takes the form:
   [index] [x-coord] [y-coord]
-}

import System.Environment (getArgs,getProgName)
import System.IO (hPutStrLn,stderr)
import Control.Parallel.Strategies (parList,rpar,using,rseq,parMap)
import System.Exit (exitFailure)
import qualified Data.Map as Map
import System.Random.Shuffle (shuffleM)
import System.Random (randomRIO)
import Data.List (sortBy,maximumBy)
import Data.Function (on)
import Control.Monad (replicateM)


type City = (Int,Float,Float)
type Route = [Int]

-- MUTATES OFFSPRING. TUNE MUTATION FUNCITON HERE
mutate :: Route -> IO Route
mutate tour = do
   let len = length tour
   index1 <- randomRIO (0, len - 1)
   index2 <- randomRIO (0, len - 1)
   let mutatedTour = swapCities index1 index2 tour
   return mutatedTour

-- SWAPS TWO CITIES IN A TOUR
swapCities :: Int -> Int -> Route -> Route
swapCities i j tour = map swap [0 .. ((length tour) - 1)]
  where
    swap k
      | k == i     = (tour !! j)
      | k == j     = (tour !! i)
      | otherwise  = (tour !! k)
```

```
--PERFORMS THE CROSSOVER BETWEEN EACH PAIR OF PARENTS IN A GIVEN LIST
crossover :: [(Route, Route)] -> IO [Route]
crossover parentPairs = mapM (\(parent1, parent2) -> crossoverSingle parent1 parent2) parentPairs
--Ordered Crossover (OX)
crossoverSingle :: Route -> Route -> IO Route
crossoverSingle parent1 parent2 = do
   let len = length parent1
   (start, end) <- do
      indices <- shuffleM [0 .. (len - 1)]
      let start' = head indices
          end' = last indices
      return (min start' end', max start' end')
   let slice = take (end - start) . drop start
      sliceP1 = slice parent1
      remainderP2 = filter (`notElem` sliceP1) parent2
      offspring = sliceP1 ++ remainderP2
   mutationProb <- randomRIO (1, 100) :: IO Int
   finalOffspring <- if mutationProb <= 5       --TUNE MUTATION PROBABILITY HERE
                then mutate offspring
                else return offspring
   return $ finalOffspring ++ [head finalOffspring]

-- CALCULATES THE TOTAL DISTANCE TRAVELED DURING A TOUR
tourLength :: Map.Map (Int, Int) Float -> Route -> Float
tourLength distances tour =
   sum [Map.findWithDefault 0 (city1,city2) distances | (city1,city2) <- zip tour (tail tour)]

-- CALCULATES THE FITNESS OF AN INDIVIDUAL
tourFitness :: Map.Map (Int, Int) Float -> Route -> Float
tourFitness distances tour = 1 / tourLength distances tour --TUNE FITNESS METRIC HERE

-- RANDOMLY GENERATES A ROUTE THAT VISITS EACH CITY ONCE AND RETURNS TO
THE STARTING POINT
newRandomRoute :: Int -> IO Route
newRandomRoute numCities = do
   shuffledIndices <- shuffleM [1..numCities]
   return $ shuffledIndices ++ [head shuffledIndices]

-- RANDOMLY GENERATES AN INITIAL POPULATION
generateInitPop :: Int -> Int -> IO [Route]
generateInitPop numCities populationSize =
   sequence [newRandomRoute numCities | _ <- [1..populationSize]]

-- PARSES INPUT FILE AND STORES CITY INFORMATION
```

```
parseCity :: String -> City
parseCity line = case words line of
   [index,x,y]   -> (read index, read x, read y)
   _             -> error "Invalid problem file format"

-- CALCULATES PAIRWISE EUCLYDEAN DISTANCE FOR THE GIVEN SET
calculateDistances :: [City] -> Map.Map (Int,Int) Float
calculateDistances cities =
   Map.fromList [((i,j), distance (x1,y1) (x2,y2)) | (i,x1,y1) <- cities, (j,x2,y2) <- cities, i /= j]

-- CALCULATES EUCLYDEAN DISTANCE BETWEEN TWO CITIES
distance :: (Float,Float) -> (Float,Float) -> Float
distance (x1,y1) (x2,y2) = sqrt ((x2 - x1)^(2::Integer) + (y2 - y1)^(2::Integer))

-- RANDOMLY PICKS A PARENT FROM A LIST
selectParent :: [Route] -> IO Route
selectParent elites = do
   parentIndex <- randomRIO (0, length elites - 1)
   return (elites !! parentIndex)

-- HELPER FUNCITON TO BREAK A LIST INTO CHUNKS WITH chunkSize ELEMENTS IN EACH
makeChunks :: Int -> [a] -> [[a]]
makeChunks _ [] = []
makeChunks chunkSize lst = chunk : makeChunks chunkSize rest
   where
      (chunk,rest) = splitAt chunkSize lst

-- CREATE A NEW GENERATION FROM THE OLD
generateNextGeneration :: [Route] -> [Route] -> IO [Route]
generateNextGeneration remainingPopulation elites = do
   pairs <- replicateM 50 $ do    -- TUNE NUMBER OF CROSSOVERS PER GENERATION HERE
      parent1 <- selectParent elites
      parent2 <- selectParent elites
      return (parent1, parent2)

   let pairChunks = makeChunks 1 pairs  --TUNE NUMBER OF PAIRS PER CHUNK HERE
      offSpring = map crossover pairChunks `using` parList rseq

   offspring <- sequence (offSpring)
   return $ elites ++ (concat offspring) ++ remainingPopulation

-- THIS FUNCTION PERFORMS THE BULK OF THE ALGORITHM
evolve :: Int -> [Route] -> Map.Map (Int, Int) Float -> IO [Route]
evolve 0 population _ = return population
```

```
evolve gen population distances = do
  let fitnesses = parMap rpar (tourFitness distances) population
      sortedPopulation = map snd $ sortBy (compare `on` fst) $ zip fitnesses population
      elites = take (length population `div` 10) sortedPopulation
      remainingPopulation = drop (length population `div` 10) sortedPopulation
  newGeneration <- generateNextGeneration remainingPopulation elites
  evolve (gen - 1) newGeneration distances

main :: IO ()
main = do
  args <- getArgs
  case args of
    [filename] -> do
      content <- readFile filename
      let cities = map parseCity (lines content)
          distances = calculateDistances cities
          numCities = length cities
            initialPopulation <- generateInitPop numCities numCities  --TUNE INITIAL POPUALTION
SIZE HERE
      let numGenerations = 100  --TUNE NUMBER OF GENERATIONS HERE
      finalPopulation <- evolve numGenerations initialPopulation distances
      let fitnesses = parMap rpar (tourFitness distances) finalPopulation
          bestTour = snd $ maximumBy (compare `on` fst) $ zip fitnesses finalPopulation
          bestFitness = tourFitness distances bestTour
          bestDistance = tourLength distances bestTour
      putStrLn $ "Best Tour: \n" ++ show bestTour ++ "\n"
      putStrLn $ "Fitness: " ++ show bestFitness
      putStrLn $ "Distance: " ++ show bestDistance
    _ -> do
      pn <- getProgName
      hPutStrLn stderr $ "Usage: " ++ pn ++ " <tsp_filename>"
      exitFailure
```