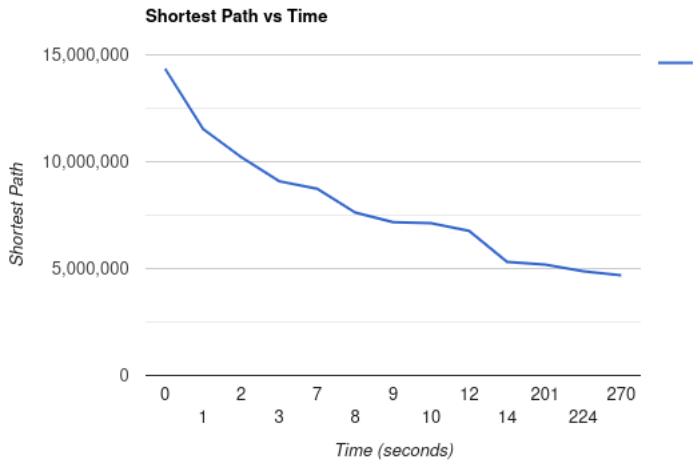
hw8 - Traveling Salesperson

Ian Macfarlane, A02243880

March 22, 2021

You can run my code using:
mpicc hw8.c -o hw8 -lm
mpiexec -np 2 ./hw8
My implementation should run for any number of processors. Though if you only give it 1 processor it won't start running the algorithm because there are no slaves.
If you want to run my sequential version just replace hw8 in the above commands with sequential.
I implemented my traveling salesperson algorithm using a parallel genetic algorithm. I initially implemented the genetic algorithm in sequence, which is in sequential.c This sequential version is actually a lot faster than my parallel implementation. I'm pretty sure that this is mostly because I am using c and all my array operations are super slow. For my parallel implementation, as processors increase my genetic algorithm increases its global population. So that each process no matter how many processes is always dealing with the same size sub population.




```
#include <mpi.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <time.h>

int distance[100][100];// returns distance between city a and b

int pathDistance(int path[100]) {

  int dist = 0;
  for (int i = 0; i < 99; i++) {
```

```
    dist += distance[path[i]][path[i+1]];
  }
  dist += distance[path[0]][path[99]];

  return dist;
}

int main(int argc, char ** argv)
{
  int rank, size, data[1], flag;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Request request;
  MPI_Status status;

  srand(time(NULL));
  time_t startTime = time(NULL);

  // read in city locations from cities.txt
  int locationX[100];// returns x location of city a
  int locationY[100];// returns y location of city a
  FILE *fp;
  fp = fopen("cities.txt", "r");
  char buff[6];
  for (int i = 0; i < 200; i++) {
    fscanf(fp, "%s", buff);
    if (i%2 == 0) {
      sscanf(buff, "%d", &locationX[i/2]);
    }
    else {
      sscanf(buff, "%d", &locationY[(i-1)/2]);
    }
  }

  // fill 2d array with distance between each pair of cities
  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
      // calculate distance between two cities using distance formula
      distance[i][j] = sqrt(pow(locationX[i]-locationX[j], 2) + pow(locationY[i]-locationY
    }
  }

  if (rank == 0) {

    // initialize global population with random paths
    int popSize = 1000 * (size - 1);
    int population[popSize][100];// array of paths
    for (int i = 0; i < popSize; i++) {
      // fill path in population
      for (int j = 0; j < 100; j++) {
        population[i][j] = j;
      }
```

```
  // randomize path
  for (int j = 0; j < popSize; j++) {
    int a = rand()%100;
    int b = rand()%100;
    population[i][a] = b;
    population[i][b] = a;
  }
}

int shortest = INT_MAX;
int iterations = 0;
while (iterations < 100000) {

  int subPop[1000][100];
  int slave = 1;

  // distribute sub populations
  for (int i = 0; i <= popSize; i++) {

    if (i != 0 && i % 1000 == 0) {

      MPI_Isend(&subPop, 1000*100, MPI_INT, slave, 0, MPI_COMM_WORLD, &request);
      slave++;
    }

    if (i != popSize) {
      for (int j = 0; j < 100; j++) {
        subPop[i%1000][j] = population[i][j];
      }
    }
  }

  // repopulate global population
  int subs = 0;
  while (subs < size-1) {
    flag = 0;
    MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &flag, &status);
    if (flag != 0) {
      MPI_Recv(&subPop, 1000*100, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATU

      // append subPop to population
      for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 100; j++) {
          population[i+1000*subs][j] = subPop[i][j];
        }
      }

      subs++;
    }
  }

  // calculate shortest path in population
  // calculate fitness for each sub population
  int fitness[popSize];// contains path distance for each path in population
  int order[popSize];// returns location of path in population
```

```
    for (int i = 0; i < popSize; i++) {
      fitness[i] = pathDistance(population[i]);
      order[i] = i;
    }

    // perform selection by sorting fitness from lowest to greatest, each even path is p
    int sorted = 0;
    while (sorted == 0) {
      sorted = 1;
      for (int i = 0; i < popSize-1; i++) {
        if (fitness[i] > fitness[i+1]) {
          sorted = 0;
          int temp = fitness[i];
          fitness[i] = fitness[i+1];
          fitness[i+1] = temp;

          temp = order[i];
          order[i] = order[i+1];
          order[i+1] = temp;
        }
      }
    }

    if (shortest > fitness[0]) {
      shortest = fitness[0];
      printf("Time: %ld, Iteration: %d, Shortest Path: %d\n", time(NULL)-startTime, itera
      //printf("%ld %d\n", time(NULL)-startTime, shortest);
    }

    // perform random mutation
    for (int i = 0; i < popSize; i++) {
      // randomly swap two cities
      int a = rand()%100;
      int b = rand()%100;
      population[i][a] = b;
      population[i][b] = a;
    }

    iterations++;
  }

  // tell slaves to stop looping
  for (int i = 1; i < size; i++) {
    MPI_Isend(&data, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &request);
  }

  printf("Shortest Path: %d\n", shortest);
}
else {
  int population[1000][100];
  int loop = 1;
  while (loop == 1) {

    // recieve subpopulation from master
    flag = 0;
```

```
MPI_Iprobe(0, 0, MPI_COMM_WORLD, &flag, &status);
if (flag != 0) {
  MPI_Recv(&population, 1000*100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

  // calculate fitness for each sub population
  int fitness[1000];// contains path distance for each path in population
  int order[1000];// returns location of path in population
  for (int i = 0; i < 1000; i++) {
    fitness[i] = pathDistance(population[i]);
    order[i] = i;
  }

  // perform selection by sorting fitness from lowest to greatest, each even path is
  int sorted = 0;
  while (sorted == 0) {
    sorted = 1;
    for (int i = 0; i < 999; i++) {
      if (fitness[i] > fitness[i+1]) {
        sorted = 0;
        int temp = fitness[i];
        fitness[i] = fitness[i+1];
        fitness[i+1] = temp;

        temp = order[i];
        order[i] = order[i+1];
        order[i+1] = temp;
      }
    }
  }

  // perform crossover
  for (int i = 0; i < 500; i+=2) {
    // select random crossover point
    int point = rand()%99;

    for (int j = 0; j < 100; j++) {
      if (j <= point) {
        population[order[i+500]][j] = population[order[i]][j];
        population[order[i+501]][j] = population[order[i+1]][j];
      }
      else {
        population[order[i+500]][j] = population[order[i+1]][j];
        population[order[i+501]][j] = population[order[i]][j];
      }
    }
  }

  // send subpopulation to master
  MPI_Isend(&population, 1000*100, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
}

// check to see if algorithm is finished
flag = 0;
MPI_Iprobe(0, 1, MPI_COMM_WORLD, &flag, &status);
if (flag != 0) {
```

```
            loop = 0;
          }
        }
      }

      MPI_Finalize();
      return 0;
}
```