Assignment 8

Stuart Hopkins, A02080107

March 2020

1 Traveling Salesperson

I started creating the Traveling Salesperson by making a serial program. After creating a serial program using Partial Crossover methods I made it parallel. To make the program parallel, I send the best path info to the process with a rank one greater. If this path is the best, it will eventually get through to all processes. Here is my code:

```
#include <iostream>
#include <mpi.h>
#include <unistd.h>
#include <bits/stdc++.h>
#include <stdlib.h>
#include <time.h>
#include <chrono>
#include <vector>

#define MCW MPI_COMM_WORLD

using namespace std;
using namespace std::chrono;

void print(string message, vector<int> arr);
void print(string message, int x);
void printProgress(int rank, int gen, int lowest);
void printBreak();

vector<int> swap(vector<int> vect, int left_idx, int right_idx);

vector<int> fitness(const int DATA_X[], const int DATA_Y[], vector<vector<int>> vect);
vector<vector<int>> sort_by_fitness(vector<vector<int>> vect, vector<int> fit_vect);

vector<int> mutate(vector<int> vect);

int findIndex(vector<int> vect, int value);
vector<int> crossover(vector<int> top, vector<int> bottom, int start, int end);
vector<vector<int>> generateNewVect(vector<vector<int>> curr_vect);
```

```
vector<vector<int>> initialize_vect(int SIZE);


int main(int argc, char **argv) {

  int rank, size;
  int data;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MCW, &rank);
  MPI_Comm_size(MCW, &size);

  const int SIZE = 100;
  const int ITERS = 1000;
  const int DATA_X[SIZE] = { 179140, 78270, 577860, 628150, 954030, 837880, 410640,
                             287850, 270030, 559020, 353930, 515920, 648080, 594550,
                             386690, 93070, 93620, 426870, 437000, 789810, 749130,
                             481030, 670720, 273890, 138430, 85480, 775340, 862980,
                             155180, 274070, 333340, 822150, 158880, 815560, 678240,
                             394470, 631300, 528320, 666940, 298650, 243750, 220500,
                             338920, 313110, 856380, 549250, 537400, 502110, 498840,
                             482310, 416930, 418400, 374170, 412370, 301090, 235690,
                             475940, 268540, 130500, 81660, 64520, 264690, 90230,
                             38370, 15430, 138890, 264580, 86690, 209190, 425890,
                             312480, 373360, 442850, 505100, 542610, 566730, 615970,
                             612120, 634410, 879480, 868760, 807670, 943060, 827280,
                             896040, 920900, 746380, 734300, 730780, 870570, 607060,
                             926580, 812660, 701420, 688600, 743800, 819700, 683690,
                             732680, 685760
                           };
  const int DATA_Y[SIZE] = { 750703, 737081, 689926, 597095, 510314, 811285, 846947,
                             600161, 494359, 199445, 542989, 497472, 470280, 968799,
                             907669, 395385, 313966, 39662, 139949, 488001, 575522,
                             286118, 392925, 892877, 562658, 465869, 220065, 312238,
                             263662, 74689, 456245, 399803, 612518, 707417, 709341,
                             679221, 846813, 824193, 845130, 816352, 745443, 654221,
                             381007, 201386, 564703, 565255, 604425, 435463, 590729,
                             571034, 765126, 638700, 695851, 570904, 737412, 782470,
                             439645, 609753, 712663, 732470, 711936, 529248, 612484,
                             610277, 579032, 482432, 421188, 394738, 347661, 376154,
                             177450, 142350, 106198, 189757, 224170, 262940, 237922,
                             303181, 320152, 239867, 286928, 334613, 368070, 387076,
                             413699, 454842, 440559, 452247, 471211, 549620, 453077,
                             669624, 614479, 559132, 580646, 669521, 857004, 682649,
                             857362, 866857
                           };
  // Creation of fstream class object
  srand(time(NULL) >> rank);

  vector<vector<int>> vect = initialize_vect(SIZE);
  vector<int> fitness_vect = fitness(DATA_X, DATA_Y, vect);
  int prev_fitness = fitness_vect.front();
  printProgress(rank, -1, prev_fitness);

  for(int a = 0; a < ITERS; a++) {
    vect = generateNewVect(vect);

    fitness_vect = fitness(DATA_X, DATA_Y, vect);
    vect = sort_by_fitness(vect, fitness_vect);

    if(fitness_vect.front() < prev_fitness) {
      prev_fitness = fitness_vect.front();
      printProgress(rank, a, prev_fitness);
    }

    if(!(a % 5)) {
      int tmp_arr[SIZE];
      for(int i = 0; i < SIZE; i++) {
        tmp_arr[i] = vect[0][i];
      }
      MPI_Send(tmp_arr, SIZE, MPI_INT, (rank + 1) % size, 0, MCW);
      MPI_Recv(tmp_arr, SIZE, MPI_INT, MPI_ANY_SOURCE, 0, MCW, MPI_STATUS_IGNORE);
      for(int i = 0; i < SIZE; i++) {
        vect[0][i] = tmp_arr[i];
      }
    }
  }

  MPI_Finalize();
  return 0;
}


vector<int> swap(vector<int> vect, int left_idx, int right_idx) {
  int tmp = vect[left_idx];
  vect[left_idx] = vect[right_idx];
  vect[right_idx] = tmp;

  return vect;
}


vector<int> mutate(vector<int> vect, int times_to_mutate) {
  // 8 seems like a probable number of good mutations
  for(int i = 0; i < times_to_mutate; i++) {
    int left_idx = rand() % vect.size();
    int right_idx = rand() % vect.size();
    vect = swap(vect, left_idx, right_idx);
  }

  return vect;
}


vector<int> fitness(const int DATA_X[], const int DATA_Y[], vector<vector<int>> vect) {
  vector<int> fitness_vect;
  fitness_vect.resize(vect.size());
  int distance;

  for(int i = 0; i < vect.size(); i++) {
    distance = 0;
    for(int j = 0; j < vect.size() - 1; j++) {
      int x1 = DATA_X[vect[i][j]];
      int y1 = DATA_Y[vect[i][j]];

      int x2 = DATA_X[vect[i][j+1]];
      int y2 = DATA_Y[vect[i][j+1]];

      int dist_x = max(x1, x2) - min(x1, x2);
      int dist_y = max(y1, y2) - min(y1, y2);

      distance += dist_x + dist_y;
    }
    int x1 = DATA_X[vect[i].front()];
    int y1 = DATA_Y[vect[i].front()];

    int x2 = DATA_X[vect[i].back()];
    int y2 = DATA_Y[vect[i].back()];

    int dist_x = max(x1, x2) - min(x1, x2);
    int dist_y = max(y1, y2) - min(y1, y2);

    distance += dist_x + dist_y;
    fitness_vect[i] = distance;
  }

  return fitness_vect;
}


vector<vector<int>> sort_by_fitness(vector<vector<int>> vect, vector<int> fit_vect) {
  int tmp;
  vector<int> tmp_vect;
  for(int i = 0; i < fit_vect.size(); i++) {
    for(int j = 0; j < fit_vect.size(); j++) {
      if(fit_vect[i] < fit_vect [j]) {
        tmp = fit_vect[i];
        tmp_vect = vect[i];

        fit_vect[i] = fit_vect[j];
        vect[i] = vect[j];
        fit_vect[j] = tmp;
        vect[j] = tmp_vect;
      }
    }
  }

  return vect;
}


int findIndex(vector<int> vect, int value) {
  for(int i = 0; i < vect.size(); i++) {
    if(vect[i] == value) {
      return i;
    }
  }
  return -1;
}


vector<int> crossover(vector<int> top, vector<int> bottom, int start, int end) {
  for(int i = start; i < end; i++) {
    int idx = findIndex(bottom, top[i]);
    bottom = swap(bottom, i, idx);
  }
  return bottom;
}


vector<vector<int>> generateNewVect(vector<vector<int>> curr_vect) {
  vector<vector<int>> new_vect;
  vector<vector<int>> selected_vect;
  const int SIZE = curr_vect.size();

  // Always take the best
  selected_vect.push_back(curr_vect.front());

  for(int i = 0; i < sqrt(SIZE) - 1; i++) {
    selected_vect.push_back(curr_vect[rand() % SIZE]);
  }

  new_vect.push_back(curr_vect.front());

  // Fill some with mutations
  for(int i = 0; i < sqrt(SIZE); i++) {
    new_vect.push_back(
      mutate(selected_vect[rand() % selected_vect.size()], SIZE / 2)
    );
  }

  // Fill the rest with crossovers
  while(new_vect.size() < SIZE) {
    int rnd1 = rand() % SIZE;
    int rnd2 = rand() % SIZE;

    new_vect.push_back(
      crossover(
        selected_vect[rand() % selected_vect.size()],
        selected_vect[rand() % selected_vect.size()],
        min(rnd1, rnd2),
        max(rnd1, rnd2)
      )
    );
  }
  return new_vect;
}


vector<vector<int>> initialize_vect(int SIZE) {
  vector<vector<int>> vect;
  for(int i = 0; i < SIZE; i++) {
    vector<int> vect2;
    for(int j = 0; j < SIZE; j++) {
      vect2.push_back(j);
    }
    vect2 = mutate(vect2, SIZE / 2);

    vect.push_back(vect2);
  }
  return vect;
}


void print(string message, vector<int> vect) {
  cout << message << ": " << endl;

  cout << vect[0];
  for(int i = 1; i < vect.size(); i++) {
    cout << ", " << vect[i];
  }
  cout << endl;
}

void print(string message, int x) {
  cout << message << ": " << x << endl;
}

void printBreak() {
  cout << "\n--------------------------------\n";
}

void printProgress(int rank, int gen, int lowest) {
  cout << "Process: " << rank << " | Generation: " << gen << " | Most Viable: " << lowest << endl;
}
```

Then the command line arguments:

```
$ mpic++ main.cpp
$ mpirun -np 8 -oversubscribe a.out

Process: 2 | Generation: -1 | Most Viable: 53781132
Process: 1 | Generation: -1 | Most Viable: 49631082
Process: 7 | Generation: -1 | Most Viable: 52006812
Process: 5 | Generation: -1 | Most Viable: 53827262
Process: 0 | Generation: -1 | Most Viable: 51768700
Process: 4 | Generation: -1 | Most Viable: 55265002
Process: 6 | Generation: -1 | Most Viable: 47479694
Process: 3 | Generation: -1 | Most Viable: 49972954
Process: 1 | Generation: 1 | Most Viable: 44977576
Process: 0 | Generation: 1 | Most Viable: 47028552
Process: 5 | Generation: 1 | Most Viable: 47457108
Process: 7 | Generation: 1 | Most Viable: 46030000
Process: 5 | Generation: 2 | Most Viable: 46413266
Process: 0 | Generation: 2 | Most Viable: 42847790
Process: 6 | Generation: 2 | Most Viable: 45204400
Process: 0 | Generation: 3 | Most Viable: 41500348
Process: 5 | Generation: 4 | Most Viable: 46263572
Process: 7 | Generation: 3 | Most Viable: 44788622
Process: 1 | Generation: 5 | Most Viable: 44378018
Process: 6 | Generation: 3 | Most Viable: 42157578
Process: 0 | Generation: 4 | Most Viable: 41122602
Process: 4 | Generation: 1 | Most Viable: 45411860
Process: 6 | Generation: 4 | Most Viable: 41389628
Process: 2 | Generation: 1 | Most Viable: 44220994
Process: 0 | Generation: 5 | Most Viable: 40797374
Process: 6 | Generation: 5 | Most Viable: 40351448
```

```
Process: 1 | Generation: 6 | Most Viable: 40336378
Process: 4 | Generation: 2 | Most Viable: 45048578
Process: 4 | Generation: 3 | Most Viable: 44216152
Process: 7 | Generation: 6 | Most Viable: 39925028
Process: 0 | Generation: 7 | Most Viable: 40416976
Process: 3 | Generation: 1 | Most Viable: 45185916
Process: 2 | Generation: 5 | Most Viable: 44210838
Process: 0 | Generation: 8 | Most Viable: 39055416
Process: 1 | Generation: 10 | Most Viable: 40302000
Process: 5 | Generation: 6 | Most Viable: 44216152
Process: 5 | Generation: 7 | Most Viable: 43901976
Process: 1 | Generation: 11 | Most Viable: 39055416
Process: 3 | Generation: 3 | Most Viable: 44991792
Process: 0 | Generation: 12 | Most Viable: 38930706
Process: 0 | Generation: 13 | Most Viable: 38313926
Process: 1 | Generation: 15 | Most Viable: 39001808
Process: 3 | Generation: 5 | Most Viable: 44252358
Process: 3 | Generation: 6 | Most Viable: 44210838
Process: 2 | Generation: 6 | Most Viable: 44185440
Process: 1 | Generation: 16 | Most Viable: 38313926
Process: 2 | Generation: 7 | Most Viable: 43880392
Process: 6 | Generation: 14 | Most Viable: 39932940
Process: 6 | Generation: 15 | Most Viable: 38372976
Process: 1 | Generation: 17 | Most Viable: 38136458
Process: 4 | Generation: 8 | Most Viable: 44199818
Process: 7 | Generation: 16 | Most Viable: 38181556
Process: 4 | Generation: 10 | Most Viable: 44032834
Process: 7 | Generation: 17 | Most Viable: 37445302
Process: 0 | Generation: 20 | Most Viable: 38290360
Process: 5 | Generation: 13 | Most Viable: 43772472
Process: 2 | Generation: 11 | Most Viable: 40302000
Process: 5 | Generation: 14 | Most Viable: 43145224
Process: 2 | Generation: 12 | Most Viable: 39917886
Process: 0 | Generation: 21 | Most Viable: 37359552
Process: 3 | Generation: 11 | Most Viable: 43880392
Process: 2 | Generation: 13 | Most Viable: 39358360
Process: 3 | Generation: 13 | Most Viable: 41851646
Process: 4 | Generation: 14 | Most Viable: 43933536
Process: 1 | Generation: 21 | Most Viable: 38019762
Process: 3 | Generation: 16 | Most Viable: 39358360
Process: 2 | Generation: 16 | Most Viable: 39001808
Process: 4 | Generation: 16 | Most Viable: 41631364
Process: 3 | Generation: 18 | Most Viable: 38808186
Process: 2 | Generation: 18 | Most Viable: 38615628
Process: 5 | Generation: 21 | Most Viable: 41152824
Process: 5 | Generation: 25 | Most Viable: 40615892
Process: 5 | Generation: 26 | Most Viable: 38213766
Process: 5 | Generation: 31 | Most Viable: 35822766
```

```
Process: 5 | Generation: 36 | Most Viable: 35255718
Process: 5 | Generation: 38 | Most Viable: 34865758
Process: 5 | Generation: 41 | Most Viable: 34860466
Process: 5 | Generation: 46 | Most Viable: 34167498
Process: 5 | Generation: 71 | Most Viable: 33129838
Process: 5 | Generation: 81 | Most Viable: 32482088
Process: 5 | Generation: 86 | Most Viable: 32317112
Process: 5 | Generation: 92 | Most Viable: 32268016
Process: 5 | Generation: 95 | Most Viable: 31604820
Process: 5 | Generation: 116 | Most Viable: 30299732
Process: 5 | Generation: 121 | Most Viable: 29904826
Process: 5 | Generation: 130 | Most Viable: 29599470
Process: 3 | Generation: 21 | Most Viable: 37370746
Process: 3 | Generation: 25 | Most Viable: 36846662
Process: 3 | Generation: 35 | Most Viable: 36184992
Process: 3 | Generation: 36 | Most Viable: 36129226
Process: 3 | Generation: 61 | Most Viable: 33129838
Process: 3 | Generation: 71 | Most Viable: 32482088
Process: 3 | Generation: 79 | Most Viable: 32317112
Process: 3 | Generation: 101 | Most Viable: 32030090
Process: 3 | Generation: 106 | Most Viable: 30299732
Process: 3 | Generation: 112 | Most Viable: 30295864
Process: 3 | Generation: 113 | Most Viable: 30166964
Process: 4 | Generation: 21 | Most Viable: 38808186
Process: 4 | Generation: 22 | Most Viable: 38651282
Process: 4 | Generation: 25 | Most Viable: 38213766
Process: 4 | Generation: 26 | Most Viable: 36846662
Process: 4 | Generation: 30 | Most Viable: 36784454
Process: 4 | Generation: 34 | Most Viable: 36744794
Process: 4 | Generation: 35 | Most Viable: 35255718
Process: 4 | Generation: 39 | Most Viable: 34860466
Process: 4 | Generation: 42 | Most Viable: 34823388
Process: 4 | Generation: 43 | Most Viable: 34401018
```

You can see that this outputs correctly. It should be not that because each process self reports some information comes out of order. When viewed using the generation stamps it reads correctly. Each process will eventually report the same lowest value because they do not report from a central source.