Programming 1Assignment Report

Student Name: Dazhi Li

Student NETID: dazhili

Student ID#: 2400330

# 1. Documentation

This part briefly describes how I implement my genetic algorithms (evaluate, crossover, mutate) and parallel strategies (OpenMP).

## Evaluate function

Evaluate function is the easiest part for completing the GA algorithm. My evaluation function firstly calculates the fitness, which is the distance of an itinerary, by the coordinates data of cities. Then I used C++ standard library method sort() to finish sorting those trips. As a result, I got a sorted list of trips which will be used in next steps.

Initially, I wrote a quick sorting method by myself. It did work in sorting those trips by ascending. However, when I try to improve the running time of this program, I found that the execution time of my handmade sorting function takes 80% of the total running time. As quick sort method could not be parallelized by OMP, I must give it up and leaving it in my source code.

## Crossover function

I wrote three crossover functions and evaluate them. These methods came from online research and my personal thinking.

### Fully Greedy; shortest distance = 518 (Abandoned)

I call this method fully greedy just because I try to find next city in a child[i]’s itinerary only by shortest distance. I came up with this method from the greedy algorithm. Child[i+1] is still the complement of child[i] as usual.

This method turns out a result that, the fitness decreasing speed is quite fast. However, this could lead to a local optimal result, and I must increase the mutation rate to get a better result. After several attempts on parameters tuning, I finally got a best result as around 560. So, this method was totally abandoned. But the code was left for further research.

### Swapping genes sequential; shortest distance = 447 (Abandoned)

This method is implemented by my after searching related materials on solving TSP problems. As we all know if parent[i] and parent[i+1] swap a sequential of genes, the biggest problem is that the switched part of genes repeats with the unmodified part. I found a method that could solve this problem. For an example, parent[i] = ABCD and parent[i+1] = DABC and we are going to switch the middle two genes which is BC from parent[i] and AB from parent[i+1]. Directly switching those two parts lead to A repeats in parent[i] and C repeats in parent[i+1]. My method is swapping one by one. On parent[i], we are going to swap B with A. But we are not really swapping these two. I will look up the index of letter A in parent[i], which is 0. Then I will swap letters in parent[i] only. By swapping letters A and B in parent[i] we will get a result of child[i] = BACD without any letters repeat/conflicts. That is the same way on how we should manage child[i+1] and going on letters. Shown in Fig.1

A diagram of a diagram

Description automatically generatedFinally, I got a shortest distance of 447. However, this method was abandoned for two reasons. Firstly, it does not come from our lectures or assignment instructions. Secondly, it utilized rand() method to randomly select genes sequential for swapping, which will make it hard for parallelization.

Fig.1 Swapping genes

### Standard Method; shortest distance = 447 (Chosen)

This method came from our homework assignment instructions and lecture discussions. Child[i] uses the first city of parent[i]. The next city should be chosen from cities in parent[i] and parent[i+1]. Parent[i] give a city which is right after the last determined city in child[i]. And so does parent[i+1]. If the last determined city in child[i] is the last city in parent[i]/parent[i+1], we choose the first city in parents. Then we compare those two answers and choose the shortest one to the last determined city in child[i]. If both appeared in child[i], we randomly choose a city which is has not been visited yet. Child[i+1] is just the complement of child[i].

## Mutation function

### Simply reversed two (Abandoned)

This method was initially implemented by its simplicity. I generated two random numbers which indicated the index of cities in the itinerary should be swapped. But this method showed poor ability on jumping out of the global optimum even with a max mutation rate of 100. As a result, this method was abandoned very quickly.

### Reverse gene sequential (Chosen)

A couple of rectangular objects with letters and numbers

Description automatically generated with medium confidenceThis method randomly selects a sequential of genes. Then it reverse it from the begin to the end of this sequential. As shown in fig.2

Fig.2 Reversed Mutation

## Parallelization strategies

My parallelization strategies is divided into three methods. These methods takes up most of the running time and could be parallelized by OMP.

### Evaluation

Evaluation is also the easiest part to achieve parallelism. Due to its computation does not require data from previous iteration. I implement OMP parallel at the loop of computing the fitness of each trip. The sorting method is not implemented with parallelism because it is not doable.

### Crossover

Crossover part looks difficult because of its complicated computation logic on generating children. However, this function could be parallelized from the outer loop of calculating each child[i] and child[i+1]. Loops inside the loop do not require multi-threads because they are running sequentially. I have noticed that calculating child[i] is totally independent to other child[i], which means they do not rely on the results of previous iteration. That’s the reason that why I put my multi-core processing at the outer loop.

### Mutation

Mutation is also a good part for multi-core processing. The mutation in each offsprings is totally independent. However, there is a rand() function to generate random sequential for mutation. We cannot put the rand() function into the multi-core processing because every rand() result relies on the previous one.

I originally created a for loop to generate random numbers and store them into lists. This loop is sequentially operated. Then I used the OMP parallel in the next for loop which is doing the mutation operations. The mutation for loop can read data from the pre-generated random lists to achieve the same results of using rand() function in the mutations loops as well as achieving multi-core processing.

# 2.Source Code

I have only modified the EvalOverMutate\_template.cpp, which is the source code here.

For better viewing, I deleted methods that was abandoned in my attempts. Other EvalOverMuatate methods could be found in my zip source code.

#include <iostream>  // cout

#include <stdlib.h>  // rand

#include <math.h>    // sqrt, pow

#include <omp.h>     // OpenMP

#include <string.h>  // memset

#include <algorithm>

#include "Timer.h"

#include "Trip.h"

#define CHROMOSOMES    50000 // 50000 different trips

#define CITIES         36    // 36 cities = ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

#define TOP\_X          25000 // top optimal 25%

#define MUTATE\_RATE    45    // optimal 50%

using namespace std;

/\*

 \* Evaluates each trip (or chromosome) and sort them out

 \*/

bool tripCompare(Trip A, Trip B) {

    return A.fitness < B.fitness;

}

void evaluate( Trip trip[CHROMOSOMES], int coordinates[CITIES][2] ) {

    // calculating the distance of each trip

    int i, lastCityXCord, lastCityYCord;

    float totalDistance;

#pragma omp parallel for private(i, totalDistance, lastCityXCord, lastCityYCord)

    for (int i=0; i < CHROMOSOMES; i ++) {

        float totalDistance = 0;

        int lastCityXCord = 0;

        int lastCityYCord = 0;

        for (int city=0; city < CITIES; city++) {

            int cityIndex = ( trip[i].itinerary[city] >= 'A' ) ? trip[i].itinerary[city] - 'A' : trip[i].itinerary[city] - '0' + 26;

            totalDistance += sqrt(pow(coordinates[cityIndex][0] - lastCityXCord, 2) + pow(coordinates[cityIndex][1] - lastCityYCord, 2));

            lastCityXCord = coordinates[cityIndex][0];

            lastCityYCord = coordinates[cityIndex][1];

        }

        // assign fitness

        trip[i].fitness = totalDistance;

    }

    // sorting

    std::sort(trip, trip + CHROMOSOMES, tripCompare);

    // Debugging quick sort

    if ( DEBUG ) {

        for (int i=0; i < CHROMOSOMES; i++) {

            cout << trip[i].fitness << endl;

        }

    }

}

/\*

 \* Generates new TOP\_X offsprings from TOP\_X parents.

 \* Noe that the i-th and (i+1)-th offsprings are created from the i-th and (i+1)-th parents

 \*/

void crossover( Trip parents[TOP\_X], Trip offsprings[TOP\_X], int coordinates[CITIES][2] ) {

    float distanceMap[CITIES][CITIES];

    int i;

#pragma omp parallel for private(i) shared(distanceMap)

    for (int i=0; i<CITIES; i++) {

        distanceMap[i][i] = -1;

        for (int j=i+1; j<CITIES; j++) {

            float dist = sqrt(pow(coordinates[i][0] - coordinates[j][0], 2) + pow(coordinates[i][1] - coordinates[j][1], 2));

            distanceMap[i][j] = dist;

            distanceMap[j][i] = dist;

        }

    }

    // generating child[i] and child[i+1]

#pragma omp parallel for

    for (int i=0; i<TOP\_X-1; i+=2) {

        offsprings[i].itinerary[0] = parents[i].itinerary[0];

        int travelHistory[CITIES] = {0};

        for (int j=1; j<CITIES; j++) {

            char previousCity = offsprings[i].itinerary[j-1];

            int previousCityIndex = ( offsprings[i].itinerary[j-1] >= 'A' ) ? offsprings[i].itinerary[j-1] - 'A' : offsprings[i].itinerary[j-1] - '0' + 26;

            travelHistory[previousCityIndex] = 1;

            char nextCityA, nextCityB;

            // getiing the near city of the leaving city

            for(int k=0; k< CITIES-1; k++) {

                if (parents[i].itinerary[k] == previousCity) {

                    nextCityA = parents[i].itinerary[k+1];

                }

                if (parents[i+1].itinerary[k] == previousCity) {

                    nextCityB =parents[i+1].itinerary[k+1];

                }

            }

            // If the leaving city is the last one, choose the first city as the leaving city

            if (parents[i].itinerary[CITIES-1] == previousCity) {

                nextCityA = parents[i].itinerary[0];

            }

            if (parents[i+1].itinerary[CITIES-1] == previousCity) {

                nextCityB = parents[i+1].itinerary[0];

            }

            // comparing distance and usage map to decide next city

            int cityAIdx = (nextCityA >= 'A') ? nextCityA - 'A' : nextCityA - '0' +26;

            int cityBIdx = (nextCityB >= 'A') ? nextCityB - 'A' : nextCityB - '0' +26;

            bool isAReachable = !travelHistory[cityAIdx];

            bool isBReachable = !travelHistory[cityBIdx];

            char nextCity;

            if (isAReachable && isBReachable) {

                if (distanceMap[previousCityIndex][cityAIdx] <= distanceMap[previousCityIndex][cityBIdx]) {

                    nextCity = nextCityA;

                } else {

                    nextCity = nextCityB;

                }

            } else if (isAReachable || isBReachable) {

                if (isAReachable) {

                    nextCity = nextCityA;

                } else {

                    nextCity = nextCityB;

                }

            } else if (!isAReachable && !isBReachable) {

                for(int k=0; k< CITIES; k++) {

                    if (travelHistory[k] == 0) {

                        nextCity = (k < 26) ? 'A' + k : k - 26 + '0';

                        break;

                    }

                }

            }

            offsprings[i].itinerary[j] = nextCity;

        }

        //generate child[i+1]

        for (int j=0; j<CITIES-1; j+=2) {

            offsprings[i+1].itinerary[j] = offsprings[i].itinerary[j+1];

            offsprings[i+1].itinerary[j+1] = offsprings[i].itinerary[j];

        }

    }

}

/\*

 \* Mutate a pair of genes in each offspring.

 \*/

void mutate( Trip offsprings[TOP\_X] ) {

    int rateList[TOP\_X], idxStartList[TOP\_X], idxEndList[TOP\_X];

    for (int i=0; i<TOP\_X;i++) {

        rateList[i] = rand() % 100;

        idxStartList[i] = rand()%(CITIES-1); //[0,34]

        idxEndList[i] = idxStartList[i] + 1 + rand()%(CITIES -1 - idxStartList[i]); //[idxStart+1,35]

    }

    int i, idxStart, idxEnd;

#pragma omp parallel for private(i, idxStart, idxEnd) shared(rateList, idxStartList, idxEndList, offsprings)

    for (int i=0; i<TOP\_X; i++) {

        if (rateList[i] < MUTATE\_RATE) {

            idxStart = idxStartList[i];

            idxEnd = idxEndList[i];

            while (idxStart < idxEnd)

            {

                char tmp = offsprings[i].itinerary[idxStart];

                offsprings[i].itinerary[idxStart] = offsprings[i].itinerary[idxEnd];

                offsprings[i].itinerary[idxEnd] = tmp;

                idxStart++;

                idxEnd--;

            }

        }

    }

}

# 3.Execution Result

Shortest distance = 447.638 shown in Fig.3

A screenshot of a computer screen

Description automatically generatedParallelism performance improvement = 19,874,612/9,861,817 = 2.015

Fig.3 shortest distance

Shown in Fig.4

# A screenshot of a computer screen Description automatically generated4.Discussions

Fig.4 Performance comparing result

## 4.1 How crossover methods influence the shortest results

As I have mentioned above, I implement total 3 different kinds of crossover algorithms.

In computational speed(1 core):

Swapping gene sequential (9,341 ,843)>

fully greedy(13,602,849)>

Standard method (19,874,612)

In best shortest distance on different mutation rate

Standard method(447.638)=

Swapping gene sequential (447.638)>

Fully greedy(518.99)

## 4.2 How number of cores influence the computational speed

These data are based on program execution results from cssmpi8h

Fortunately, I have met the situation that professor mentioned on the lecture that 3 cores computation speed is less than 2 cores. I think the explanation of how virtual cores are assigned and communication exactly matches the result here.

# 5.LAB1 Details

## 5.1 Between pi\_monte and pi\_integral, which performs better on OMP and why

The answer is pi\_integral performs better with OMP. Based on my execution result observation, pi\_integral decreased the running time while pi\_monte does not.

Running time for each program (1,000,000 iterations), OMP version is set to 3 cores:

pi\_monte(88,810) < pi\_monte\_omp(1,287,740)

pi\_integral(17,962) > pi\_integral\_omp(7,423)

The reasons for why this is happening is that pi\_monte used rand() method. And in the loop, which is used for multi-core processing, the rand() method is also called. This will result in no performance improvement on OMP because of the fake random in C++. However, pi\_integral does not use rand() function at all. This will make those loops being executed at the same time by those 3 threads.

## 5.2 Source Code for pi\_monte\_omp.cpp and pi\_integral\_omp.cpp

Pi\_monte\_omp.cpp

// Program to compute Pi using Monte Carlo methods

// Original code availabe from Dartmath through Internet

// Code modified to compute all four quadrants

#include "Timer.h"    // for performance measurement

#include <omp.h>

#include <iostream>

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <string.h>

#define SEED 35791246

using namespace std;

int main(int argc, char\* argv[]) {

  int niter=0;

  double PI25DT = 3.141592653589793238462643; // the actual PI

  double x, y;   // x-y coordinates in each quadrant

  int count = 0; // # of points in each quadrant of unit circle

  double radius;

  double pi = 0.0;

  cout << "Enter the number of iterations used to estimate pi: ";

  cin >> niter;

  Timer timer;

  timer.start( );

  int quad;

  omp\_set\_num\_threads(3);

  srand( SEED );  // initialize random numbers

#pragma omp parallel for private(quad, x, y, radius, count) reduction(+:pi)

  for ( int quad = 0; quad < 4; quad++ ) {

    // for each quadrant

    count = 0;

    for ( int i = 0; i < niter; i++ ) {

      x = ( double )rand( ) / RAND\_MAX;

      y = ( double )rand( ) / RAND\_MAX;

      radius =  x \* x + y \* y;

      if ( radius <= 1 ) count++;

    }

    pi += ( double )count / niter;

  }

  cout << "elapsed time for pi = " << timer.lap( ) << endl;

  printf( "# of trials = %d, estimate of pi is  %.16f, Error is %.16f\n",

    niter, pi, fabs( pi - PI25DT ) );

  return 0;

}

Pi\_integral\_omp.cpp

// Program to compute Pi using Integration

#include <omp.h>

#include <iostream>

#include <stdio.h>

#include <math.h>

#include "Timer.h"   // for performance measurement

using namespace std;

int main( int argc, char \*argv[] )

{

  int niter = 0;

  double PI25DT = 3.141592653589793238462643; // the actual PI

  double x = 0.0;

  cout << "Etner the number of iterations used to estimate pi: ";

  cin >> niter;

  Timer timer;

  timer.start( );

  double h   = 1.0 / (double) niter;

  double sum = 0.0;

  int i;

  omp\_set\_num\_threads(3);

#pragma omp parallel for private(x, i) shared(h)  reduction(+:sum)

  for ( int i = 1; i <= niter; i++ ) {

    // compute integral from 0.0 to 1.0

    x = h \* ( ( double )i - 0.5 );

    sum += ( 4.0 / ( 1.0 + x \* x ) );

  }

  double pi = h \* sum;

  cout << "elapsed time for pi = " << timer.lap( ) << endl;

  printf("# of trials = %d, estimate of pi is  %.16f, Error is %.16f\n",

   niter, pi, fabs( pi - PI25DT ) );

  return 0;

}

## 5.3 Execution Result

As shown in Fig.5

A computer screen with white text

Description automatically generated

Fig.5 Lab1 execution result

# 6.Summary

In this report I briefly documented the algorithm and parallelism implementation methods. And I discussed some elements (algorithms and number of cores) that could influence the results. At last, I discussed about the lab1 and how the results is formed.