CSS 534
Program 1: Parallelizing Travelling Salesman Problem with OpenMP

Summary: Parallelization Strategies and Performance Impact

This assignment was a great way to learn about the benefits of effective parallelization and its substantial impact on program performance when compared to sequential execution. The program's inherently calculation-intensive nature results in a significant time cost during sequential execution, clocking in at **20,072,555** units. However, the strategic implementation of parallelization techniques significantly reduces the execution time to **8,109,933** units, highlighting the remarkable performance improvement of **2.47 times**, achieved via parallelization.

The project also helped in understanding the importance of carefully selecting functions and calculations for parallelization. Excessive parallelization can potentially have adverse effects on performance, emphasising the need to identify the right balance between parallelization and sequential execution for optimal results.

Parallelization Strategies

For optimal performance of the program, parallelization is employed within ***evaluate( )*** and ***crossover( )*** functions with OpenMP to utilise multi-core processors and improve the performance of the genetic algorithm. While trying to parallelize the ***mutate()*** function, it was evident that the parallelization techniques cannot be randomly used with each and every function. The details of parallelization techniques can be found below:

Evaluation Parallelization

The ***evaluate()*** function is used to assess the fitness level of an extensive set of 50,000 trips through 150 iterations. This evaluation includes computing the distances between each city within an itinerary, with calculations commencing from the 0,0 position. Given the substantial computational demands and memory usage of this process, this function is an ideal candidate for parallelization.

The ***evaluate()*** function has been parallelized using the ***#pragma omp parallel for reduction(+:totalDistance)*** directive. This directive helps distribute the computational workload across multiple threads. The ***reduction(+:totalDistance)*** clause ensures the prevention of data races in multi-threaded scenarios, thus ensuring that the distance is correctly calculated.

```
59  // Function to evaluate fitness of trips
60  void evaluate(Trip trip[CHROMOSOMES], int coordinates[CITIES][2]) {
61      // Cache for storing distances of all cities
62      float distanceCache[CITIES][CITIES];
63
64      for (int i = 0; i < CITIES; ++i) {
65          for (int j = 0; j < CITIES; ++j) {
66              distanceCache[i][j] = calculateDistance(i, j, coordinates);
67          }
68      }
69
70      float totalDistance=0.0;
71      //Use parallelization - using reduction for totalDistance
72      #pragma omp parallel for reduction(+:totalDistance)
73      for (int i = 0; i < CHROMOSOMES; ++i) {
74          totalDistance = 0.0;
75          char* itinerary = trip[i].itinerary;
```

Crossover Parallelization

The ***crossover()*** function is used in generating offspring, creating new and potentially more optimal itinerary between cities. Given the number of possible trips and the required computational intensity, this function substantially influences the overall program performance and is another ideal candidate for parallelization.

To enhance the efficiency of the ***crossover()*** function, parallelization has been applied using the ***#pragma omp parallel for*** directive. The ***#pragma omp parallel for*** directive is implemented above the two nested loops responsible for generating new itineraries. These loops play a significant role in creating the right offspring by selecting cities with shorter distances from pairs of parents. I haven't used parallelization on qsort function as qsort function is a system function and it is already optimised.

To avoid data races or undesired outcomes, the **currentCity** variable is declared as private to each thread so that each thread can process it independently. Notably, other variables such as parent1Index, parent2Index, parent1NextIndex and parent2NextIndex when declared as private, were impacting the program performance, and hence, I avoided including them. This confirms the importance of being careful with private variables during parallelization otherwise, the performance could be severely hampered.

```
// Function for crossover of parents to generate offspring
void crossover( Trip parents[TOP_X], Trip offsprings[TOP_X], int coordinates[CITIES][2]) {
    //Caching distances
    float distanceCache[CITIES][CITIES];
    for (int i = 0; i < CITIES; ++i) {
        for (int j = 0; j < CITIES; ++j) {
            distanceCache[i][j] = calculateDistance(i, j, coordinates);
        }
    }
    // OMP Parallelization - putting currentCity in private
    char currentCity;
    #pragma omp parallel for private( currentCity )
    for(int i=0;i<TOP_X;i+=2)
    {
```

Outcome of Parallelization Techniques implemented

The parallelization strategies with 4 threads implemented in the code have significantly enhanced the performance of the program when compared with 1 thread implementation, reducing the execution time from **20,072,555** units to **8,109,933** units.

Ratio : **2.47 times**

Comparing with Professor's code time: The performance improvement with four threads in my program is equal to **23543059** (professor's code thread1 time) / **8,109,933** (my code's thread 4 time) = **2.9 times**

Execution Output:

```
[shristi@cssmpi1h prog1]$ ./Tsp 4
# threads = 4
generation: 0
generation: 0 shortest distance = 1265.72        itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 963.146        itinerary = V1X8OT4J37RL9FZUH0EYCWSPMBQ26IANKGD5
generation: 2 shortest distance = 845.916        itinerary = VI1YHZ2S5UF4K7J6OTNXG89R3BM0ECWDALQP
generation: 3 shortest distance = 707.209        itinerary = V1JDB0EHZYUIXG6F9N4TKL3R7A8O2WC5SMPQ
generation: 4 shortest distance = 651.203        itinerary = V1HUZYE0DBMSCW5PQR3L7KFAN4T9XG8OI6J2
generation: 5 shortest distance = 536.724        itinerary = V1IO6JLAN4TXG89FK7R3DBMSWC5HZYUE02PQ
generation: 7 shortest distance = 522.879        itinerary = V1YZHUE02WSC5MPQBDR379F4XGKALJ6OI8TN
generation: 8 shortest distance = 486.543        itinerary = V1YZHUE02WSC5MPBQDJ6IO84TNXKGF9AL7R3
generation: 11 shortest distance = 476.341       itinerary = V1YHZUE20J6OI4NT8GXKF9AL7R3DQBPMSWC5
generation: 12 shortest distance = 459.577       itinerary = V1YZH5CWSMP20EUJ6IO84NTX9FKGAL7R3DBQ
generation: 17 shortest distance = 453.709       itinerary = V1YZHUE20J6OI84NTXGKF9AL7R3DBQPMSWC5
generation: 20
generation: 32 shortest distance = 449.552       itinerary = V1YZHUE20J6OI84NTXGK9FAL7R3DBQPMSWC5
generation: 40
generation: 59 shortest distance = 447.638       itinerary = V1YZHUE20J6OI84TNXGK9FAL7R3DBQPMSWC5
generation: 60
generation: 80
generation: 100
generation: 120
generation: 140
elapsed time = 8109933
[shristi@cssmpi1h prog1]$ ./Tsp 1
# threads = 1
generation: 0
generation: 0 shortest distance = 1265.72        itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 963.146        itinerary = V1X8OT4J37RL9FZUH0EYCWSPMBQ26IANKGD5
generation: 2 shortest distance = 845.916        itinerary = VI1YHZ2S5UF4K7J6OTNXG89R3BM0ECWDALQP
generation: 3 shortest distance = 707.209        itinerary = V1JDB0EHZYUIXG6F9N4TKL3R7A8O2WC5SMPQ
generation: 4 shortest distance = 651.203        itinerary = V1HUZYE0DBMSCW5PQR3L7KFAN4T9XG8OI6J2
generation: 5 shortest distance = 536.724        itinerary = V1IO6JLAN4TXG89FK7R3DBMSWC5HZYUE02PQ
generation: 7 shortest distance = 522.879        itinerary = V1YZHUE02WSC5MPQBDR379F4XGKALJ6OI8TN
generation: 8 shortest distance = 486.543        itinerary = V1YZHUE02WSC5MPBQDJ6IO84TNXKGF9AL7R3
generation: 11 shortest distance = 476.341       itinerary = V1YHZUE20J6OI4NT8GXKF9AL7R3DQBPMSWC5
generation: 12 shortest distance = 459.577       itinerary = V1YZH5CWSMP20EUJ6IO84NTX9FKGAL7R3DBQ
generation: 17 shortest distance = 453.709       itinerary = V1YZHUE20J6OI84NTXGKF9AL7R3DBQPMSWC5
generation: 20
generation: 32 shortest distance = 449.552       itinerary = V1YZHUE20J6OI84NTXGK9FAL7R3DBQPMSWC5
generation: 40
generation: 59 shortest distance = 447.638       itinerary = V1YZHUE20J6OI84TNXGK9FAL7R3DBQPMSWC5
generation: 60
generation: 80
generation: 100
generation: 120
generation: 140
elapsed time = 20072555
[shristi@cssmpi1h prog1]$
```

```
[shristi@cssmpi1h prog1]$ ./Tsp 2
# threads = 2
generation: 0
generation: 0 shortest distance = 1265.72        itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 963.146        itinerary = V1X8OT4J37RL9FZUH0EYCWSPMBQ26IANKGD5
generation: 2 shortest distance = 845.916        itinerary = VI1YHZ2S5UF4K7J6OTNXG89R3BM0ECWDALQP
generation: 3 shortest distance = 707.209        itinerary = V1JDB0EHZYUIXG6F9N4TKL3R7A8O2WC5SMPQ
generation: 4 shortest distance = 651.203        itinerary = V1HUZYE0DBMSCW5PQR3L7KFAN4T9XG8OI6J2
generation: 5 shortest distance = 536.724        itinerary = V1IO6JLAN4TXG89FK7R3DBMSWC5HZYUE02PQ
generation: 7 shortest distance = 522.879        itinerary = V1YZHUE02WSC5MPQBDR379F4XGKALJ6OI8TN
generation: 8 shortest distance = 486.543        itinerary = V1YZHUE02WSC5MPBQDJ6IO84TNXKGF9AL7R3
generation: 11 shortest distance = 476.341       itinerary = V1YHZUE20J6OI4NT8GXKF9AL7R3DQBPMSWC5
generation: 12 shortest distance = 459.577       itinerary = V1YZH5CWSMP20EUJ6IO84NTX9FKGAL7R3DBQ
generation: 17 shortest distance = 453.709       itinerary = V1YZHUE20J6OI84NTXGKF9AL7R3DBQPMSWC5
generation: 20
generation: 32 shortest distance = 449.552       itinerary = V1YZHUE20J6OI84NTXGK9FAL7R3DBQPMSWC5
generation: 40
generation: 59 shortest distance = 447.638       itinerary = V1YZHUE20J6OI84TNXGK9FAL7R3DBQPMSWC5
generation: 60
generation: 80
generation: 100
generation: 120
generation: 140
elapsed time = 11518187
[shristi@cssmpi1h prog1]$ ./Tsp 3
# threads = 3
generation: 0
generation: 0 shortest distance = 1265.72        itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 963.146        itinerary = V1X8OT4J37RL9FZUH0EYCWSPMBQ26IANKGD5
generation: 2 shortest distance = 845.916        itinerary = VI1YHZ2S5UF4K7J6OTNXG89R3BM0ECWDALQP
generation: 3 shortest distance = 707.209        itinerary = V1JDB0EHZYUIXG6F9N4TKL3R7A8O2WC5SMPQ
generation: 4 shortest distance = 651.203        itinerary = V1HUZYE0DBMSCW5PQR3L7KFAN4T9XG8OI6J2
generation: 5 shortest distance = 536.724        itinerary = V1IO6JLAN4TXG89FK7R3DBMSWC5HZYUE02PQ
generation: 7 shortest distance = 522.879        itinerary = V1YZHUE02WSC5MPQBDR379F4XGKALJ6OI8TN
generation: 8 shortest distance = 486.543        itinerary = V1YZHUE02WSC5MPBQDJ6IO84TNXKGF9AL7R3
generation: 11 shortest distance = 476.341       itinerary = V1YHZUE20J6OI4NT8GXKF9AL7R3DQBPMSWC5
generation: 12 shortest distance = 459.577       itinerary = V1YZH5CWSMP20EUJ6IO84NTX9FKGAL7R3DBQ
generation: 17 shortest distance = 453.709       itinerary = V1YZHUE20J6OI84NTXGKF9AL7R3DBQPMSWC5
generation: 20
generation: 32 shortest distance = 449.552       itinerary = V1YZHUE20J6OI84NTXGK9FAL7R3DBQPMSWC5
generation: 40
generation: 59 shortest distance = 447.638       itinerary = V1YZHUE20J6OI84TNXGK9FAL7R3DBQPMSWC5
generation: 60
generation: 80
generation: 100
generation: 120
generation: 140
elapsed time = 10366272
[shristi@cssmpi1h prog1]$
```

Shortest Distance: **447.638**

Performance improvement with 4 threads: 20072555 / 8109933 = **2.47 times**

Source Code - Changes are in EvalXOverMutate.cpp and Trip.h included in zip folder.

Parallelization, Limitations, and Potential Performance Improvement

Parallelization

Effective parallelization strategies can significantly enhance program performance, as evident through this assignment, wherein, parallelization helps improve the program performance by a factor of **2.47**. This highlights the importance of using parallelization techniques appropriately to maximise performance. Within the program, the parallelization efforts are primarily focussed on the ***evaluate()*** and ***crossover()*** functions to leverage multi-core processors for execution performance improvement.

Limitations

***mutate()* Function**

Parallelization also showed its limitation during the implementation of ***mutate()*** function. The ***mutate()*** function does not seem to involve computationally intensive or time-consuming operations. Since the mutation process for each offspring seems relatively quick, parallelizing it should not yield significant performance improvements. To confirm my initial impression, I still tried parallelization, but saw a reduction in performance which led me to deduce that parallelization in this function only hurts the overall program performance. I believe the following factors could have contributed to the impact on performance when parallelizing the ***mutate()*** function:

1. Parallelizing 'mutate()' introduces data races as multiple threads might modify the same offspring, risking erroneous results.
2. The performance gains from additional parallelizations decrease as more functions are parallelized.

For illustration, when we use ***#pragma omp parallel for*** directive and execute the code, it increases the execution time

```
generation: 140
elapsed time = 9609016
[shristi@cssmpi1h prog1]$
```

Mutation Rate in Trip.h file

Mutation is being used to introduce genetic diversity into the population, allowing the algorithm to explore a broader range of optimal itinerary and avoiding premature convergence to suboptimal solutions. While mutation is essential here, the pre-defined mutation rate of 50% seemed very high, with a high likelihood of causing disruptions to good structures within offsprings or trips in this case.This disruption was hindering the genetic algorithm's ability to converge toward better solutions. Experimenting with different mutation rates, I could see that a mutation rate of 25% improved the performance of my program most significantly.

Potential Performance Improvement:

To further enhance program performance, I would consider the following:

1. Fine-tuning parallelization: Focus on evaluating each computationally intensive function in the program with substantial impacts on performance for parallelization.
2. Avoiding over-parallelization to manage overhead and negative impact on the performance.
3. Already implemented caching on distance calculation - Caching the indexes of all the cities should also help improve the performance further.

Lab Sessions 1 : Output

```
[shristi@cssmpi1h ~]$ cd /home/NETID/shristi/prog1
[shristi@cssmpi1h prog1]$ cd lab1
[shristi@cssmpi1h lab1]$ ./pi_integral
Etner the number of iterations used to estimate pi: 1000000
elapsed time for pi = 23662
# of trials = 1000000, estimate of pi is  3.1415926535897643, Error is 0.0000000000000289
[shristi@cssmpi1h lab1]$ ./pi_integral_omp 3
Etner the number of iterations used to estimate pi: 1000000
elapsed time for pi = 5665
# of trials = 1000000, estimate of pi is  3.1415926535898753, Error is 0.0000000000000822
[shristi@cssmpi1h lab1]$ ./pi_monte
Enter the number of iterations used to estimate pi: 1000000
elapsed time for pi = 87242
# of trials = 1000000, estimate of pi is  3.1400229999999998, Error is 0.0015696535897933
[shristi@cssmpi1h lab1]$ ./pi_monte_omp 3
Enter the number of iterations used to estimate pi: 1000000
elapsed time for pi = 1593007
# of trials = 1000000, estimate of pi is  3.1405550000000000, Error is 0.0010376535897931
[shristi@cssmpi1h lab1]$
```

Source code : Only created pi_integral_omp and pi_monte_omp attached in zip folder.

Which version demonstrated performance improvement with OpenMP? Answer is pi_integral_omp

Pi_integral_omp with 3 threads:

When we use ***#pragma omp parallel for*** directive inside pi_monte_omp.cpp, it improves the performance by **4.17** times. Each thread calculates a partial sum and the reduction clause will aggregate the partial sums into a single total sum.

```
// Parallelization using OpenMP: each thread calculates a partial sum of the integral. Uses private copy of x
// The reduction clause aggregates partial sums into single sum variable
  #pragma omp parallel for private( x ) 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;
```

Pi_monte_omp with 3 threads:

When we use ***#pragma omp parallel for*** directive inside pi_monte_omp.cpp ,it drastically increases elapsed time by almost **18** times. So ***#pragma omp parallel for*** directive should be avoided inside the function, if the function is used for generating random numbers.

```
  int quad;

// This OpenMP directive for parallelization will increase the elapsed time due to rand()
// thread-local copies of private variables. Reduction operation aggregates the private pi values.
#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;
```