Veermata Jijabai Technological Institute, Mumbai 400019

Experiment No.: 06

Aim : Implement Parallel Traveling Salesman Problem using OpenMP.

Name: Kiran K Patil

Enrolment No.: 211070904

Branch: Computer Engineering

Batch: IV

Theory :

Parallel Traveling Salesman Problem using OpenMP

- The program below is an implementation of the Traveling Salesman Problem (TSP) using OpenMP, which is an NP-hard problem in computer science. The program takes the number of cities and distances between them as input and randomly selects a starting city. It then uses the TSP algorithm to find the optimal path that visits each city exactly once and returns to the starting city, and outputs the optimal path and distance.

- The TSP algorithm is implemented using recursive function **tsp()** which uses depth-first search to explore all possible paths. The function takes the current city and the number of visited cities as input, and if all the cities are visited, it checks if the distance from the last city to the starting city is less than the current minimum distance. If it is, it updates the current minimum distance. Otherwise, it returns.

- If all cities are not visited, the function visits each unvisited city in parallel using OpenMP. It sets the current city as visited, updates the current path, and recursively calls the **tsp()** function with the new current city and the updated number of visited cities. After the recursive call, it reverts the changes by unmarking the current city as visited and updating the current distance.

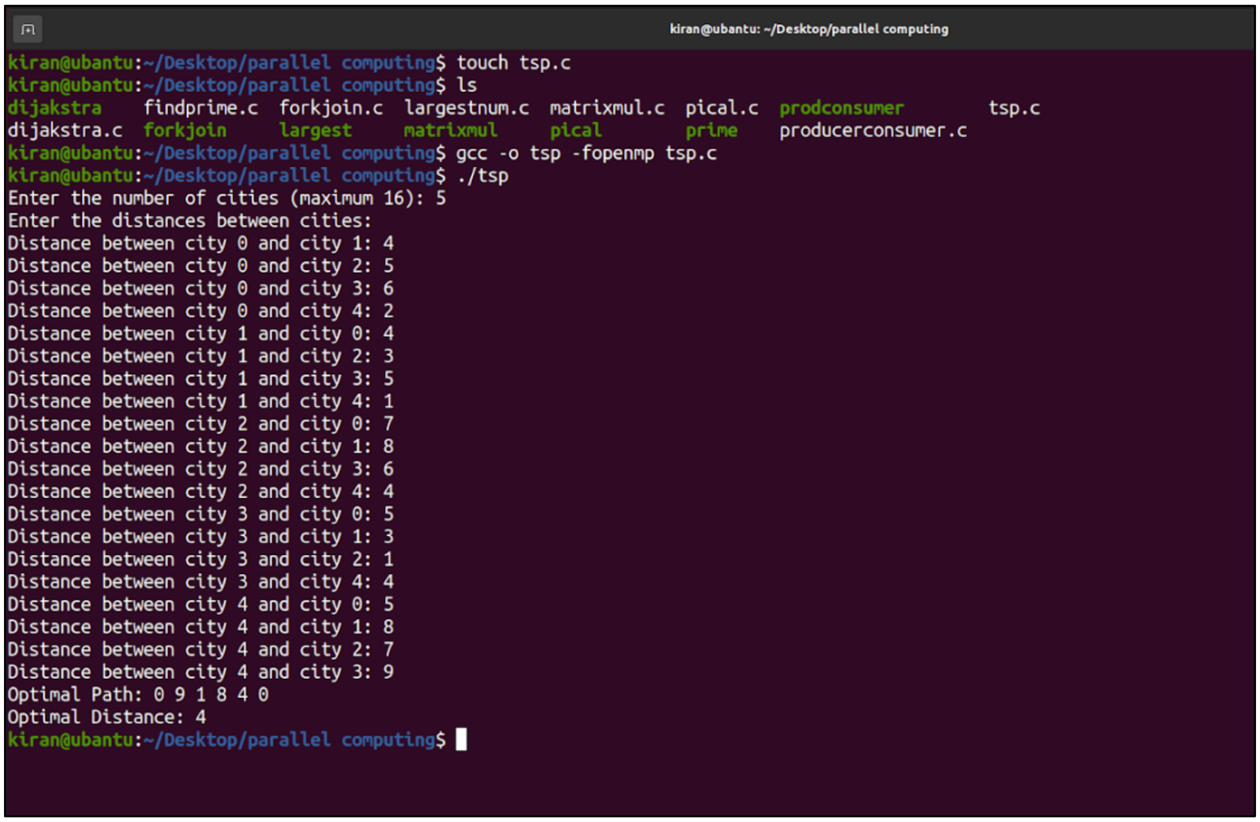
- Finally, the **main()** function initializes the distances between the cities, selects a random starting city, and calls the tsp() function in parallel using OpenMP. After the function returns, it outputs the optimal path and distance.

Program :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include <omp.h>

#define MAX_CITIES 16

int n;                // number of cities
int dist[MAX_CITIES][MAX_CITIES]; // distances between cities
int currDist = INT_MAX; // current minimum distance
int currPath[MAX_CITIES];   // current optimal path
int visited[MAX_CITIES];  // array to keep track of visited citi
es

void tsp(int currCity, int visitedCount)
{
  if (visitedCount == n) {
      if (dist[currCity][0] < INT_MAX && currDist > currPath[vis
itedCount-1] + dist[currCity][0]) {
          currDist = currPath[visitedCount-
1] + dist[currCity][0];
      }
      return;
  }

  #pragma omp parallel
  {
      int localDist = currDist;
      #pragma omp for
      for (int i = 0; i < n; i++) {
          if (!visited[i]) {
              visited[i] = 1;
              currPath[visitedCount] = dist[currCity][i];
              localDist += dist[currCity][i];
              tsp(i, visitedCount + 1);
              localDist -= dist[currCity][i];
              visited[i] = 0;
          }
      }
      #pragma omp critical
      {
          if (localDist < currDist) {
              currDist = localDist;
          }
      }
  }
}

int main()
{
  srand(time(NULL));

  // Initialize distances between cities
  printf("Enter the number of cities (maximum %d): ", MAX_CITIES
);
  scanf("%d", &n);

  printf("Enter the distances between cities:\n");
  for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
          if (i == j) {
              dist[i][j] = 0;
          } else {
              printf("Distance between city %d and city %d: ", i
, j);
              scanf("%d", &dist[i][j]);
              if (dist[i][j] == 0) {
                  dist[i][j] = INT_MAX;
              }
          }
      }
  }

  // Select a random starting city
  int start = rand() % n;

  // Find the optimal path
  #pragma omp parallel
  {
      #pragma omp single
      {
          visited[start] = 1;
          currPath[0] = 0;
          tsp(start, 1);
      }
  }
```

```
  // Output the optimal path and distance
  printf("Optimal Path: ");
  for (int i = 0; i < n; i++) {
      printf("%d ", currPath[i]);
  }
  printf("0\n");
  printf("Optimal Distance: %d\n", currDist);

  return 0;
}
```

Output :



```
kiran@ubantu:~/Desktop/parallel computing$ touch tsp.c
kiran@ubantu:~/Desktop/parallel computing$ ls
dijakstra    findprime.c  forkjoin.c  largestnum.c  matrixmul.c  pical.c  prodconsumer        tsp.c
dijakstra.c  forkjoin     largest     matrixmul     pical        prime    producerconsumer.c
kiran@ubantu:~/Desktop/parallel computing$ gcc -o tsp -fopenmp tsp.c
kiran@ubantu:~/Desktop/parallel computing$ ./tsp
Enter the number of cities (maximum 16): 5
Enter the distances between cities:
Distance between city 0 and city 1: 4
Distance between city 0 and city 2: 5
Distance between city 0 and city 3: 6
Distance between city 0 and city 4: 2
Distance between city 1 and city 0: 4
Distance between city 1 and city 2: 3
Distance between city 1 and city 3: 5
Distance between city 1 and city 4: 1
Distance between city 2 and city 0: 7
Distance between city 2 and city 1: 8
Distance between city 2 and city 3: 6
Distance between city 2 and city 4: 4
Distance between city 3 and city 0: 5
Distance between city 3 and city 1: 3
Distance between city 3 and city 2: 1
Distance between city 3 and city 4: 4
Distance between city 4 and city 0: 5
Distance between city 4 and city 1: 8
Distance between city 4 and city 2: 7
Distance between city 4 and city 3: 9
Optimal Path: 0 9 1 8 4 0
Optimal Distance: 4
kiran@ubantu:~/Desktop/parallel computing$
```

Conclusion:

- The above program is a parallel implementation of the traveling salesman problem using OpenMP. It uses a brute force approach to find the optimal path and distance between cities.

- The program initializes the distances between cities, selects a random starting city, and then finds the optimal path using the tsp() function. The tsp() function recursively visits each city and updates the current minimum distance and optimal path.

- The program uses OpenMP parallelism to speed up the execution of the tsp() function. The program outputs the optimal path and distance between cities. Overall, the program provides an efficient and parallelized solution to the traveling salesman problem.