Chris Jimenez
Dr. Mohamed Zahran
Parallel Computing
April 10, 2014

Programming Assignment 2 Report

Introduction

For this programming assignment, a modified version of the traveling salesman problem(TSP) was solved using OpenMP.

In this modified version of TSP, the salesman does not need to return to their destination after visiting all cities. For example, lets say there are 3 cities(called 0, 1, 2 respectively) and the source city is city 0. The path may be 0->1->2 or 0->2->1. Let the number of cities be n. Generally, there are (n - 1)! possible paths. So in the example above, there are 2 paths and if n = 5, there are 4! or 24 possible paths. With that said, the goal of this program is to find the most optimal path, where the optimal path is defined to be the least costly path given the distances between the cities.

Algorithm

The algorithm used was implemented specifically to take advantage of OpenMP's ability to separate the work amongst the threads using a for loop in the main method.

The OpenMP directive used to separate the work is the following:

```
#pragma omp parallel for
```

Which is called before the main for loop. The for loop iterates through all the cities except for city 0. In each iteration of the loop, a path is created from city 0 to the corresponding city in the iteration. And this acts as a driver for a recursive procedure called compute_path, which computes the path all the way up until all cities are visited. The Path struct and the main algorithm used in the program is shown below:

```
struct Path{
    int n; //number of cities
    int cost; // cost of path
    int city_path[CITY_PATH_MAX];
    int visited[CITY_PATH_MAX];
};
```

```
global num_cities = the number of cities
global best_path = current best path

procedure Driver()
      -Create a best path starting with city 0, for comparison later
      for i = 1:num_cities do
            -Create a Path p starting at city 0
            -Add city i to path p(so now path is 0->i)
            -Initial call to compute_path(p)
      endfor
      -Print best path
end_Driver

procedure compute_path(p)
      if p->n + 1 == num_cities then
            -Add remaining unvisited city(which will make p->n = num_cities)
            if check_best_path(p) == TRUE then
                  -Replace current best path with p
            endif
            -Remove last 2 cities form path p
      else
            for i = 0:num_cities do
                  -Add city i to path p
                  -Call compute_path(p)
            endfor
            -Remove last city from path p
      endif
end_compute_path
```

In the Driver() method, the work for the for loop is separated amongst the threads. Since each iteration of the for loop is not dependent on one another, this implementation works well with OpenMP.

Input/Output

The input of the program is a .txt file with an n by n adjacency matrix which has the distances between each city. An example of an adjacency matrix for n = 3 is shown below.

```
0 2 2
2 0 8
2 8 0
```

For this particular program, the number of cities must be specified at the command line after the input file is specified. Therefore, to compile and run the program for, say, a problem with 3 cities, you must type the following at the command line:

```
gcc -g -Wall -fopenmp -o tsm tsm.c
./tsm cities3.txt 3
```

The program was tested using 3, 5, and 10 cities. The output of the program is the the optimal path with the corresponding cost of the path. The output will look the like following:

```
Path: 0 1 2
Cost: 10
```

Data

Serial Time(sec)

# of cities	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
3	0.003	0.003	0.003	0.003	0.003	0.003
5	0.004	0.003	0.003	0.003	0.003	0.0032
10	0.093	0.094	0.093	0.093	0.093	0.0932

Parallel Time(sec)

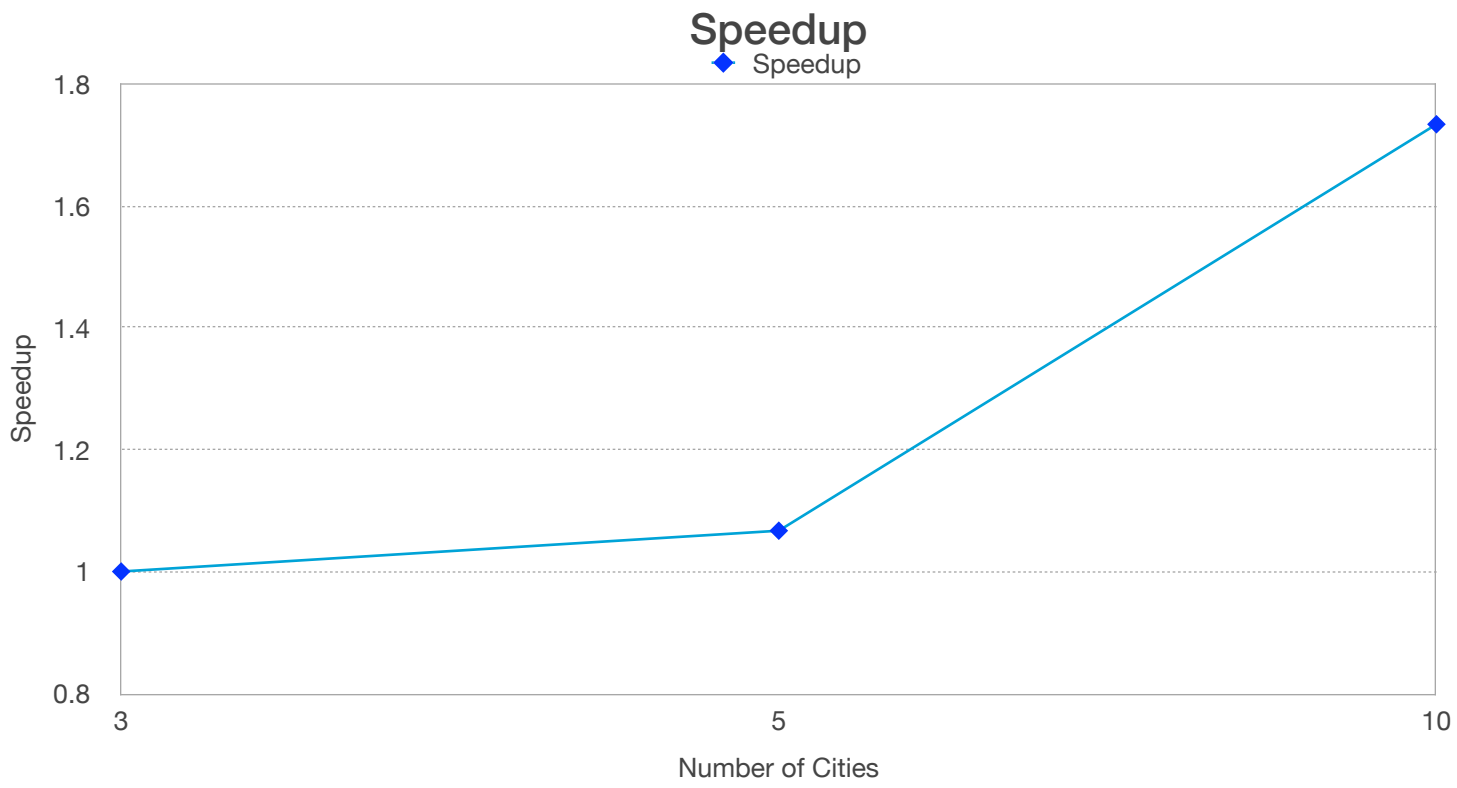
# of cities	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
3	0.003	0.003	0.003	0.003	0.003	0.003
5	0.003	0.003	0.003	0.003	0.003	0.003
10	0.053	0.053	0.054	0.053	0.055	0.0536

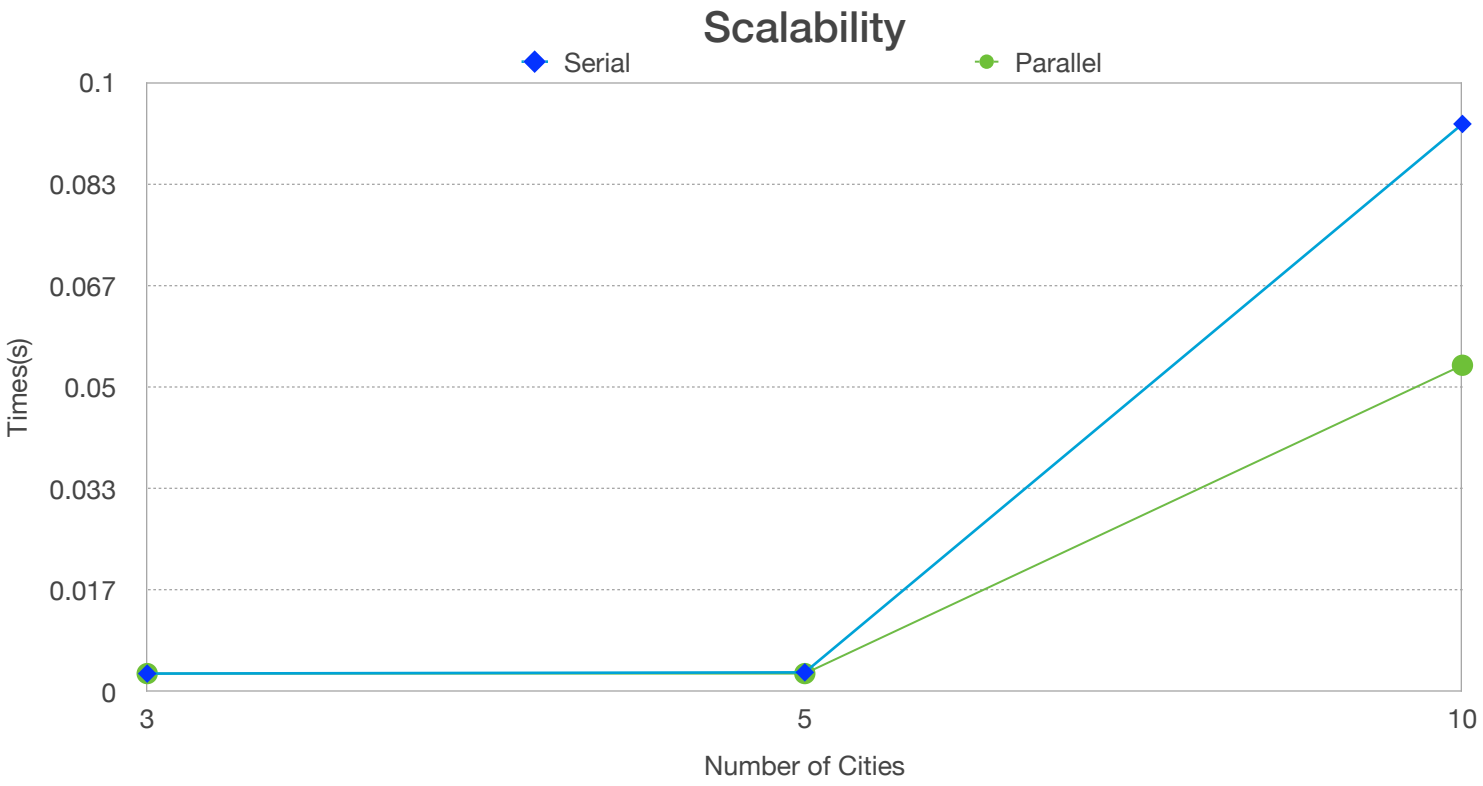
The speedup was measured using the following formula:

```
time_seq_x_cities / time_parallel_x_cities
```

Speedup

# of cities	Serial(avg)	Parallel	Speedup
3	0.003	0.003	1
5	0.0032	0.003	1.06666666666667
10	0.0932	0.0536	1.73880597014925


Speedup
Speedup
1.8
1.6
1.4
1.2
1
0.8
Speedup
3
5
10
Number of Cities


Scalability
Serial
Parallel
0.1
0.083
0.067
0.05
0.033
0.017
0
Times(s)
3
5
10
Number of Cities

Conclusion

The first graph showed that as the number of cities increased, the speed up increased as well. This indicates that the speed up is not that noticeable when the number of cities is a small number, in this case 3 or 5. But when the number of cities reaches 10, the speedup is very noticeable. The serial time for 10 cities, as shown above in the tables, is 0.0932 seconds. The parallel time for 10 cities is 0.0536 seconds, nearly 0.04 seconds faster! This means that parallelizing the program with OpenMP did make the program run faster. And based on the trend shown on the graph, the speedup would be dramatically higher if the number of cities were greater than 10.

The second graph showed that as the number of cities increased, the time it took for the program to run for both the serial version and parallel version increased as well. This is expected since the number of possible paths dramatically increases as the number of cities increases. For instance, when the number of cities n = 3, there are only 2 possible paths, but when n =10, there are 362,880 possible paths!