HPC Project

Topic:Travelling Salesman Problem

Authors: Devarsh Sheth(201301423)
Jaimin Khanderia(201301424)

1.Introduction

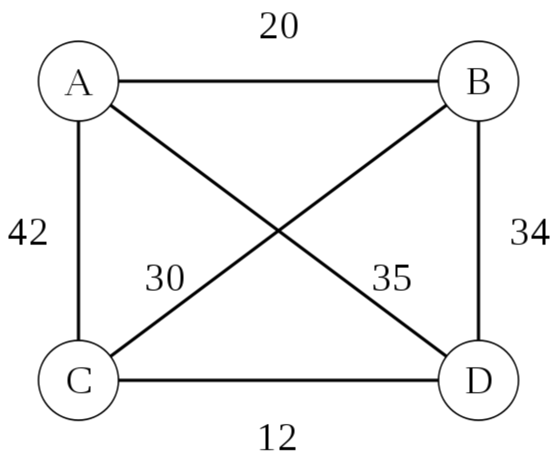
1.1 Problem Description:

The **Travelling Salesman Problem** (often called **TSP**) is a classic algorithmic problem in the field of computer science. It is focused on optimization. In this context *better solution* often means *a solution that is cheaper*. TSP is a mathematical problem. It is most easily expressed as a graph describing the locations of a set of nodes.

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once
We will be considering Symmetric Travelling Salesman Problem in which the the distance between two cities is the same in each opposite direction, forming an undirected graph.

Example:

Input:



Symmetric TSP with four cities.
Source Node: A

Output:
A-→B-→C-→D-→A

1.2 Input:

A complete undirected graph and a source node from which the tour would start.

1.3 Output:

The best tour which travels all the cities exactly once starting from source vertex and return back to the source vertex. Also the cost of the best tour is obtained as an output.

1.4 Applications:

The TSP naturally arises as a subproblem in many transportation and logistics applications, for example the problem of arranging school bus routes to pick up the children in a school district. This bus application is of important historical significance to the TSP.
Recent applications involve the scheduling of service calls at cable firms, the delivery of meals to homebound persons, the scheduling of stacker cranes in warehouses, the routing of trucks for parcel post pickup, and a host of others.More applications can be found out on the following link:
http://www.tsp.gatech.edu/index.html

2. Travelling Salesman Problem Algorithm

2.1 Pseudocode

Serial Algorithm:

```
push(stack, tour)
 while stack is non empty
        tour←pop(stack)
        if citycount (tour) = n
                if besttour(tour)
                        updatebesttour(tour)
        else for b = n − 1 downto 1
                if feasible (tour,b)
                        add(city , tour)
                        push(stack, tour)
                        removelast(tour, city )
```

2.2 Scope of parallelism

There is a lot of scope for parallelism of this algorithm.By looking at the above psuedocode we can see that we can only parallelise the core part of the algorithm which can be parallelized in two different ways. We have discussed the parallelizing strategies in section 3.

2.3 Complexity

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.The complexity of the algorithm is O(n!) where n is the total number of cites or number of vertices in an undirected graph.

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.

When we increase the problem size by increasing the number of nodes in a graph the complexity increases factorially as the number of tours is equal to n! and the cost for overall computation becomes expensive.

3 Parallelization Strategies

There are primarily two strategies for parallelizing the algorithm.Both the strategies will parallelize the TreeSearch. One would use a common stack and each process will do computations for the tour on the top of the stack until the stack is empty. The second one will parallelize the Tree search by initially splitting the stack into parts depending on the number of threads and each process will work on its own stack.

3.1

The first strategy is to parallelize the computations on different tours while using the same stack.

```
#pragma omp parallel private(tour) reduction(min:minCost)
    while(!isEmpty(stack))
        #pragma omp critical
            tour←pop(s)
        if citycount (tour) = n
            if besttour(tour)
                updatebesttour(tour)
        else for b = n − 1 downto 1
            if feasible (tour,b)
            add(city , tour)
            #pragma omp critical
                push(stack, tour)
            removelast(tour, city )
```

3.2

The second strategy splits the main stack and distributes it equally among the given number of threads.So each process has a local stack which is part of the main stack and it does the computaions only on this stack.No sharing of the main stack

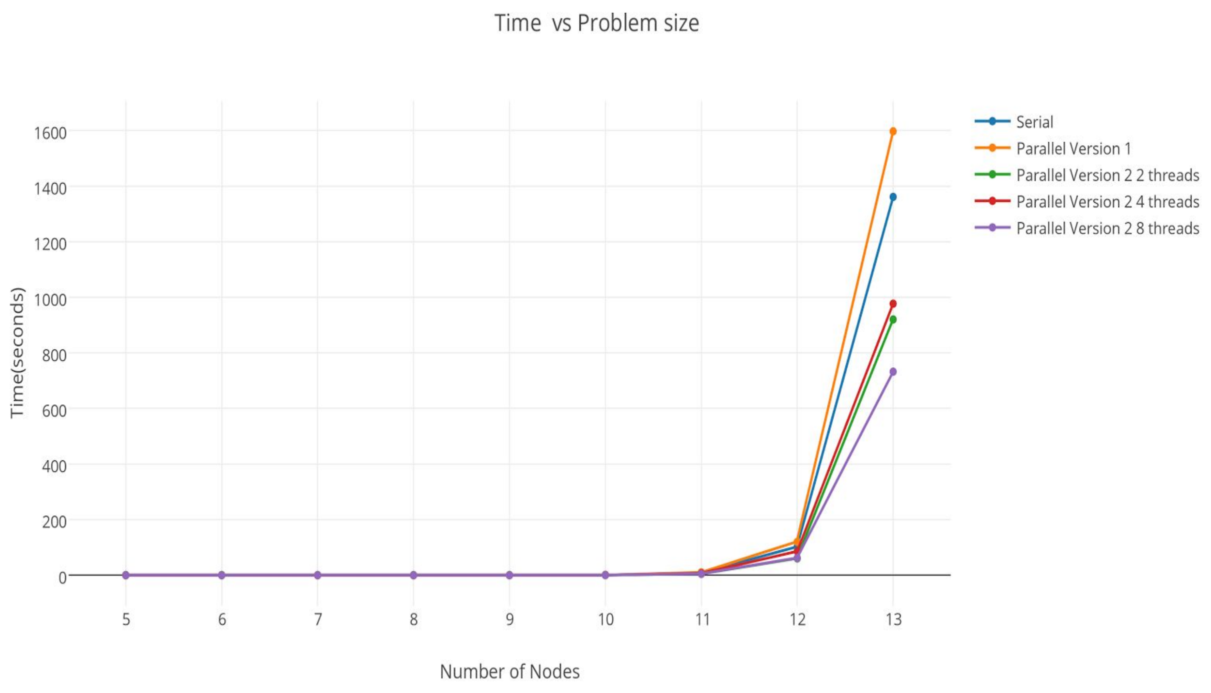
```
#pragma omp parallel private(tour) reduction(min:minCost)
        stack s1=split_stack(main_stack,rank)
        while(!isEmpty(s1))
            tour←pop(s1)
            if citycount (tour) = n
                if besttour(tour)
                    updatebesttour(tour)
            else for b = n − 1 downto 1
                if feasible (tour,b)
                add(city , tour)
                push(s1, tour)
                removelast(tour, city)
```

4 Results and Observations

4.1

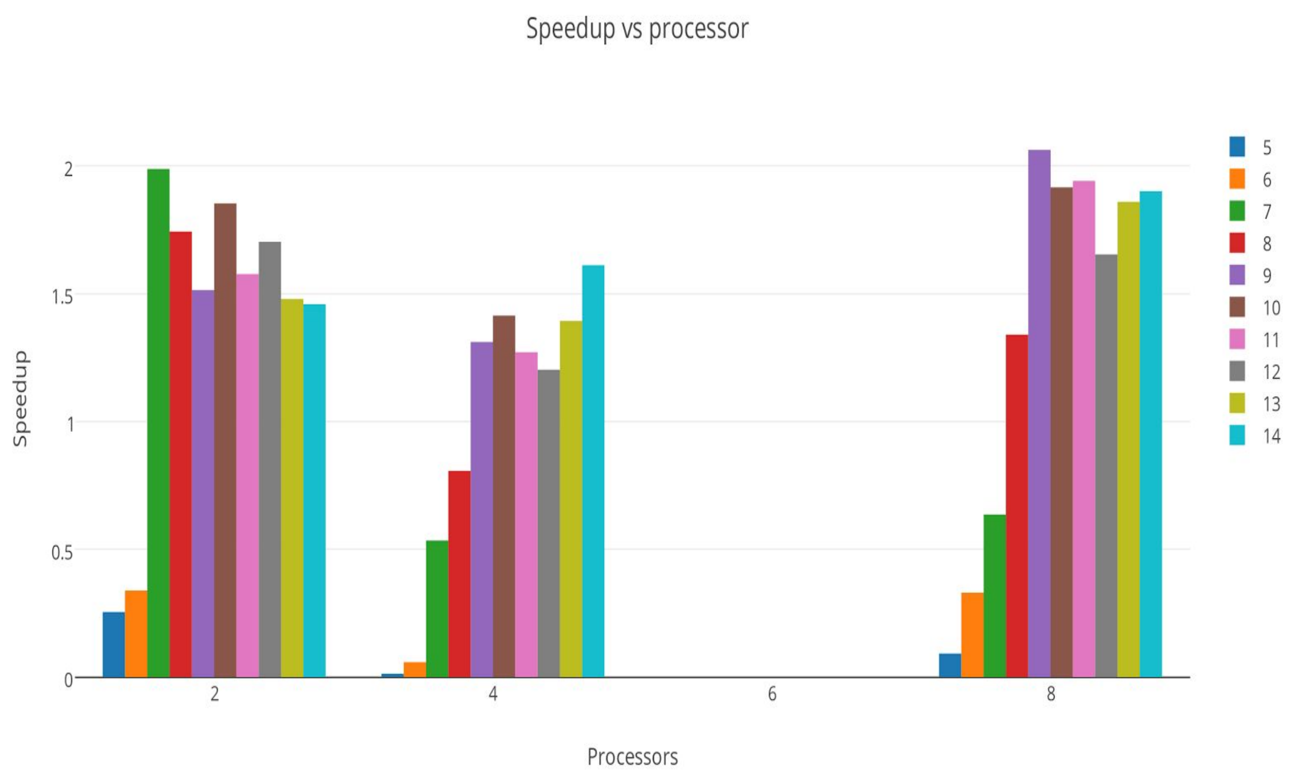
As we increase the problem size the serial part as well as the parallel part both increase but the increase of parallel part is more dominant than the serial part.Hence, we are able to parallelize this code and we are able to get a fair enough speedup.

4.2 Observations and Discussions



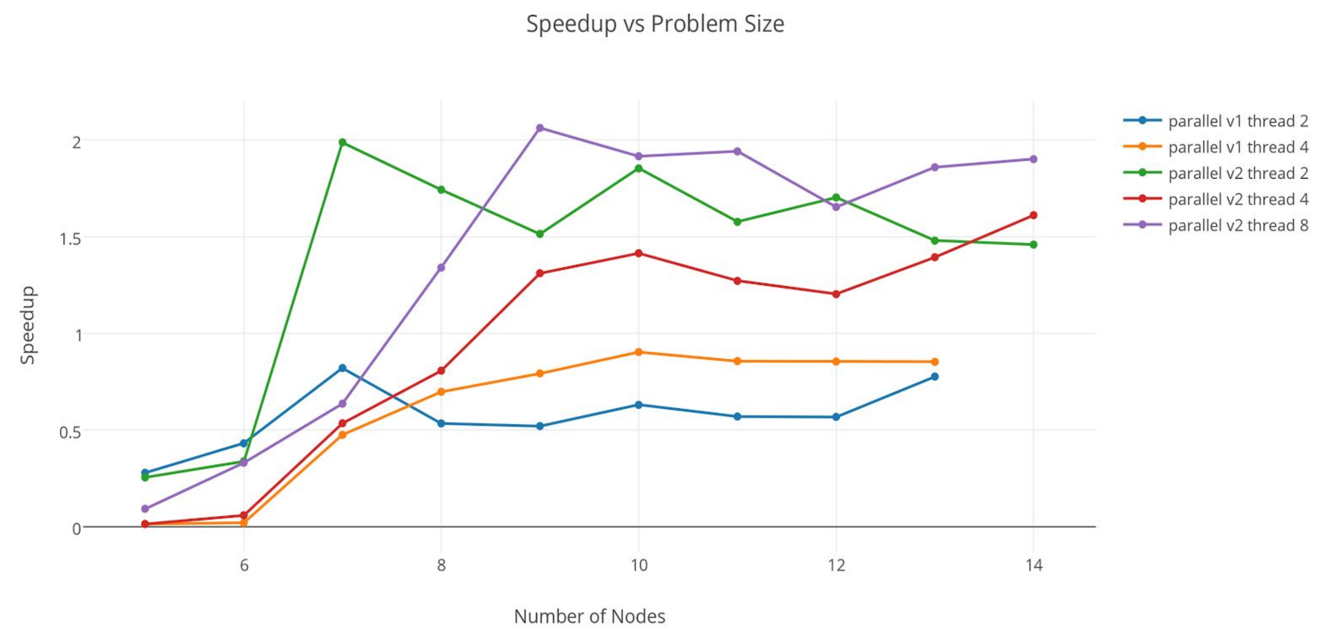
X-axis: Number of nodes
Y-axis: Time in seconds

As we can see from the graph the time taken by the serial code is much more than the parallel code. Also as we can increase the number of threads for parallel version 2 the time taken by the the code decreases.



Parallel Code Version 2
Bars represent the number of nodes in a graph for a processor
X-axis: Number of Processors
Y-axis: Speedup

As we can see from the above figure that as the number of processors increase for a fixed number of nodes(sufficiently large) the speedup also increases.For lesser number of nodes the speedup does not increase with increase in the processors.The parallel overhead generated in splitting the stack might be the reason for this.



X-axis: Number of nodes
Y-axis: Speedup

As we can see from the graph the first version is not that good and does not give any speedup.The parallel overhead generated due to critical section of pushing and popping into the stack might be the reason for this.

For second version,we can see that when the problem size is small the speedup is more in case of less number of threads whereas compared to more number of threads.The parallel overhead generated in case of more number of threads might be a reason for this.But as we increase the problem size the speedup obtained in case of 2 threads gradually decreases.Also,as the problem size increases the speedup also increases for more number of threads because the increase in parallel part is dominant over the serial part.

4.3 Karp-Flatt Metric Analysis :

We calculated the Karp Flatt metric for a fixed problem size(number of nodes=14) and we saw that the metric was slightly increasing when we increased the number of processors.

Number of Processors	Karp Flatt Metric
2	0.3708
4	0.4509
8	0.4586

5 Conclusions and Future Scope

5.1 Conclusion

As it can be seen from results mentioned in section 4 the first parallelization strategy does not give any speedup.This is due to the parallel overhead generated due to the critical section for popping and pushing onto the same stack.

On the other hand the second parallelization strategy gives speedup as the problem size increases.When comparing for 2,4 and 8 threads the code gives more speedup when there are 2 threads and when the problem size is small.This is because the overhead generated in splitting the stack is more in 4 and 8 threads. But as the size increases the parallel overhead becomes negligible with respect to the total time and thus 4 and 8 threads give more speedup.

5.2 Future Scope

The second parallelization strategy splits the stack and distributes it among the processes.Each process computes the minimum tour in it’s allocated subtree.There is no communication between the processes.If at any point of time if the cost for reaching a node from source is more than the cost of global tour then there is no meaning in going further deep in the tree.So it is possible that a process may complete its computation much before the other processes.In this case it will have to wait while the other processes work.

When a process runs out of work, it can busy-wait for more work or notification that program is terminating A process with work can send part of it to an idle process Alternatively, a process without work can request such form other process(es).

A process checks if it has at least two tours If it has received request for work, it splits its stack and sends work to the requesting process If it cannot send, it sends ”no work” reply.

Setup:

1) a process has its own my stack
2) fulfill request(my stack) checks if a request for work is received,if so it splits the stack and sends work
3) send rejects checks for work requests and sends ”no work” reply If no requests, it simply returns

References

- http://www.cas.mcmaster.ca/~nedialk/COURSES/4f03/tsp/tsp.pdf
- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- http://www.geeksforgeeks.org/travelling-salesman-problem-set-1