School of Computer Science and Engineering

J Component report

Programme : B.Tech

Course Title : Parallel and Distributed Computing

Course Code : CSE4001

Slot : D2

Title: Parallel Implementation of the Travelling Salesman Problem for real world applications

Faculty: Dr. Kumar R

Team Members:	**G S Jyothssena**	**20BCE1317**
	Satwika Sridhar	**20BCE1348**
	Prathiba Lakshmi Narayan	**20BCE1360**

ABSTRACT

This project aims to understand the Travelling Salesman Problem in detail and in turn find the most optimized algorithm for the same, that is, the algorithm that gives us the least execution time.
The TSP has numerous applications, one of which is the warehousing problem. When a warehouse receives a purchase order for a specific subset of the products kept there, these products in this order must be picked up by a vehicle before being delivered to the consumer. This is related to the TSP. The nodes of the graph are represented by the object's' storage locations. The time required to transport a vehicle from one point to another provides the distance between two nodes. A TSP may be used to discover the shortest route for the vehicle with the quickest pickup time.

The deterministic method to solve the TSP problem involves traversing all possible routes, finding the distances and hence the minimal distance. The total number of possible routes traversing n cities is n! and, therefore, in cases of large values of n it becomes difficult to find the cost of all tours in polynomial time. Parallel processing helps reduce the computation time of the TSP problem. We would also want to find the near-optimal solution of the TSP problem and understand the performance of the parallel system for that particular application. This is done using different parallelization methods like OpenMP and MPI and also using different algorithms like the Brute force algorithm and the Branch and bound algorithm.

INTRODUCTION

To find efficient algorithms to solve the TSP problem and hence use it in real world applications, we try out various serial algorithms along with their parallel implementations to show that parallelizing the work makes the process faster.
We even try different methods to parallelize the process like using OpenMP, MPI and a combination of both as well. This is also tested with different number of threads/processes.

The results obtained using the above methods are then tabulated to see which method gives us most optimization. To see when the parallel overhead gets too high we also test it with different sizes of the matrix.

The TSP algorithms that were used are the Brute Force algorithm and the Branch and Bound approach. The brute force algorithm calculates and compares all possible permutations of routes or paths to determine the shortest unique solution.
The branch and bound algorithm, for the current node in the tree, we compute a bound on the best possible solution that we can get if we down this node. If the bound on the best possible solution itself is worse than the current best, then we ignore the subtree rooted with the node.

We compute the time taken when we implement the above algorithms serially, using OpenMP and MPI which are then plotted and compared.

MOTIVATION

One of the most studied algorithms in computer science is perhaps the TSP. Numerous applications of the TSP problem exist, including computer wiring, task scheduling, minimizing fuel consumption in aircraft, vehicle routing problem, robot learning, etc.
These applications are very relevant today and their current operations can be greatly improved when using concepts of parallelization in them.

The importance of the TSP is that it is representative of a larger class of problems known as combinatorial optimization problems. TSP is known as an intractable problem.
It hasn't actually been proven that there is no tractable solution to TSP, although many of the world's top computer scientists have worked on this problem for the last 40 years, trying to find a solution but without success. Thus TSP problem belongs in the class of such problems known as NP-complete/intractable. Specifically, if one can find an efficient (i.e., polynomial-time) algorithm for the traveling salesman problem, then efficient algorithms could be found for all other NP-complete problems.
NP-complete algorithms are very slow, to the point of being impossible to use. Considering the various applications of TSP, parallelizing the TSP problem would make it more efficient.

IMPLEMENTATION DETAILS

Source code can be found at the following link : CSE4001_Project

Both the approaches were implemented using OpenMP, MPI and a combination of OpenMP and MPI

➢ OPENMP

The parallel for statement is called before the iteration in the main function.
In the function kth_permutation, the path is compared to min_path in the critical section.
OMP_GET_WTIME is used to calculate time taken.

➢ MPI

In main, for loop is split between the number of thread.
MPI_Reduce is used to find the minimum from local minimums.
MPI_Wtime is used to calculate the time difference.

- **BRUTE FORCE APPROACH**

The Brute Force approach, also known as the Naive Approach, calculates and compares all possible permutations of routes or paths to determine the shortest unique solution.

To solve the TSP using the Brute-Force approach, you must calculate the total number of routes and then draw and list all the possible routes. Calculate the distance of each route and then choose the shortest one—this is the optimal solution.

- **ALGORITHM**

Step 1: Define n- number of vertices
dist[n][n]- distance matrix
min_path set to MAX

Step 2: Define function kth_permutations with parameters vector v and integers k,p and s where p is the path and s is the source
If vector v is empty,
Find minimum of min_path and path p and return.
For the kth permutation, find the next vertice from vector v.
Increment path by the distance between previous vertice s and current v[selected].
Call the function kth_permutation again with updated v, s and p.

Step 3: Get current time start.
Iterate from 1 to factorial(n) and call the function kth_permutatiom for each.
Get time end.
Print the minimum path and time taken.

- PARALLEL (OPENMP) AND SERIAL EXECUTION OF BRUTE FORCE APPROACH

```
jyothssena@jyothssena-VirtualBox:~$ g++ parallel1.cpp -fopenmp
jyothssena@jyothssena-VirtualBox:~$ cat tsp_input.txt | ./a.out
244
15454.7
jyothssena@jyothssena-VirtualBox:~$ g++ serial1.cpp
jyothssena@jyothssena-VirtualBox:~$ cat tsp_input.txt | ./a.out
244
19243
jyothssena@jyothssena-VirtualBox:~$
```

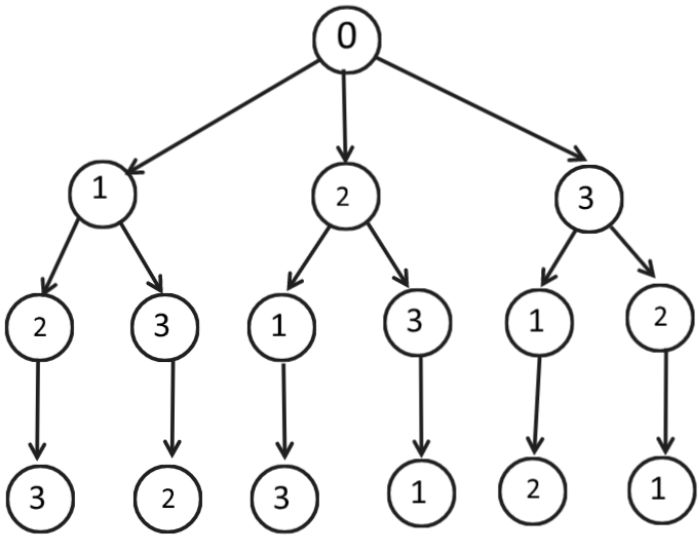
- PARALLEL (MPI) EXECUTION OF BRUTE FORCE APPROACH

```
prathiba@prathiba:~/Documents/pdc_lab/jcomp$ mpic++ parallel2.cpp
prathiba@prathiba:~/Documents/pdc_lab/jcomp$ mpiexec -np 4 ./a.out
244
1459.84
prathiba@prathiba:~/Documents/pdc_lab/jcomp$ mpiexec -np 10 ./a.out
244
544.786
```

- **DYNAMIC BRANCH AND BOUND APPROACH**

In Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best, then we ignore the subtree rooted with the node.

- **GRAPHICAL REPRESENTATION OF THE ALGORITHM**



- SERIAL EXECUTION OF DYNAMIC BRANCH AND BOUND APPROACH

```
jyothssena@jyothssena-VirtualBox:~/pdc_project$ g++ tsp_bb.cpp
jyothssena@jyothssena-VirtualBox:~/pdc_project$ cat tsp_input.txt | ./a.out
Path Taken : 0 3 5 7 2 6 4 1
Minimum cost : 244
Time taken : 184
```

- PARALLEL (OPENMP) EXECUTION OF DYNAMIC BRANCH AND BOUND APPROACH

```
jyothssena@jyothssena-VirtualBox:~/pdc_project$ g++ tsp_bbp.cpp -fopenmp
jyothssena@jyothssena-VirtualBox:~/pdc_project$ cat tsp_input.txt | ./a.out
0 3 5 7 2 6 4 1
Minimum cost : 244
Time taken : 3067.1
```

- PARALLEL (MPI) EXECUTION OF DYNAMIC BRANCH AND BOUND APPROACH

```
jyothssena@jyothssena-VirtualBox:~$ mpic++ tsp_bbmpi.cpp -fopenmp
jyothssena@jyothssena-VirtualBox:~$ mpiexec -np 8 ./a.out
244
77627.9
jyothssena@jyothssena-VirtualBox:~$
```

- **COMBINING OPENMP AND MPI FOR THE BRUTE FORCE APPROACH**

We implemented a parallel algorithm using both OpenMP and MPI constructs, parallelizing it with one method in each level.

```
#include <mpi.h>
#include <stdio.h>
#include <bits/stdc++.h>
#include<sys/time.h>
#include<iostream>
using namespace std;
#define MAX 100000
int global_min=MAX;
int dist[100][50] = {{ 0, 29, 82, 46, 68, 52, 72, 42}, { 29,0 , 55, 46,
42, 43,43, 23},
    { 82, 55, 0, 68, 46, 55,23,43}, {
46,46,68,0,82,15,72,31},{68,42,46,82,0,74,23,52}, {52, 43,
55,15,74,0,61,23},{72,43,23,72,23,61,0,42},{42,23,43,31,52,23,42,0}};
```

```
int factorial(int n) {
  if (n == 0 || n == 1) return 1;
  return n * factorial(n -1 );
}

int find_kth_permutation( vector<char>& v, int k, int p, int s,int
min_path,int rank) {
  if (v.empty()) {
    p+=dist[s][0];
    min_path=min(min_path,p);
    return min_path;
  }
  int n = (int)(v.size());
  int count = factorial(n - 1);
  int selected = (k - 1) / count;
  int x=(int)(v[selected])-48;
  p+=dist[s][x];
  s=x;
  v.erase(v.begin() + selected);

  k = k - (count * selected);
  min_path=find_kth_permutation(v, k,p,s,min_path,rank);
  return min_path;
}


int main(int argc, char* argv[]) {
    int rank,size;
    struct timeval start,end;
    gettimeofday(&start,NULL);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int min_path=MAX;
    int x=factorial(7)/size;
```

```
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 1 + (x*rank); i <=x*(rank+1); ++i) {
                vector<char> v;
                for (char i = 1; i <= 7; ++i) {
                    v.push_back(i + '0');
                }
                min_path=find_kth_permutation(v, i,0,0,min_path,rank);
            }
            #pragma omp barrier
        }
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Reduce(&min_path, &global_min, 1, MPI_INT, MPI_MIN, 0,
MPI_COMM_WORLD);
    if(rank==0){
        cout<<global_min<<endl;
        }
    MPI_Finalize();
    gettimeofday(&end,NULL);
    if(rank==0)

cout<<(end.tv_sec-start.tv_sec)*1000000L+(end.tv_usec-start.tv_usec)<<endl
;
}
```

- PARALLEL (OPENMP & MPI) EXECUTION OF BRUTE FORCE APPROACH

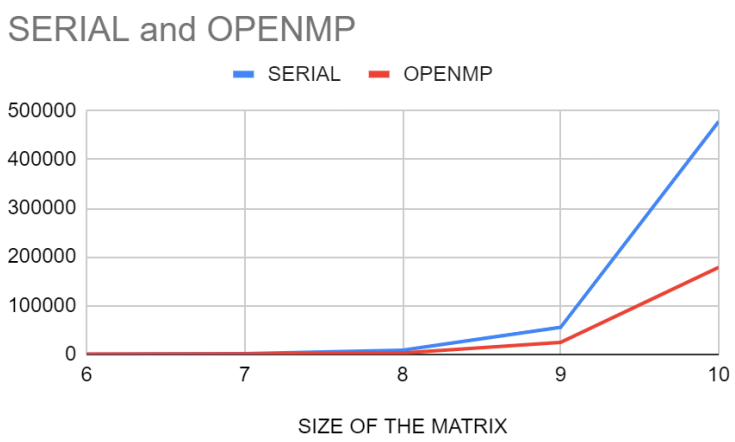
```
jyothssena@jyothssena-VirtualBox:~$ mpic++ tsp_bbc.cpp -fopenmp
jyothssena@jyothssena-VirtualBox:~$ mpiexec -np 8 ./a.out
244
398926
jyothssena@jyothssena-VirtualBox:~$
```

EXPERIMENTAL RESULTS AND DISCUSSION

- **BRUTE FORCE RESULTS (OPENMP)**

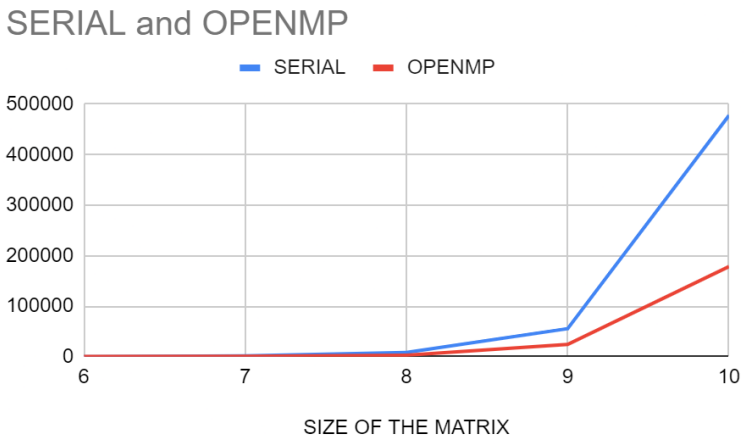
Number of threads = 2

SIZE OF THE MATRIX	SERIAL	OPENMP
6	193	265
7	1638	771
8	8457	5004
9	55451	37554
10	477856	273378



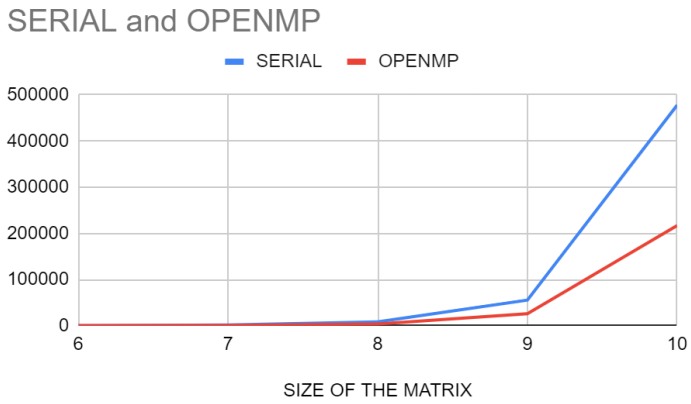
Number of threads = 4

SIZE OF THE MATRIX	SERIAL	OPENMP
6	193	345
7	1638	823
8	8457	2836
9	55451	24452
10	477856	178505



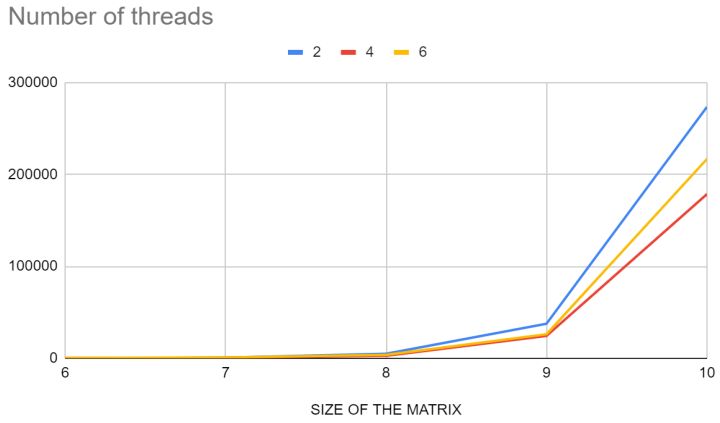
Number of threads = 6

SIZE OF THE MATRIX	SERIAL	OPENMP
6	193	516
7	1638	987
8	8457	3790
9	55451	26174
10	477856	216886



Number of threads

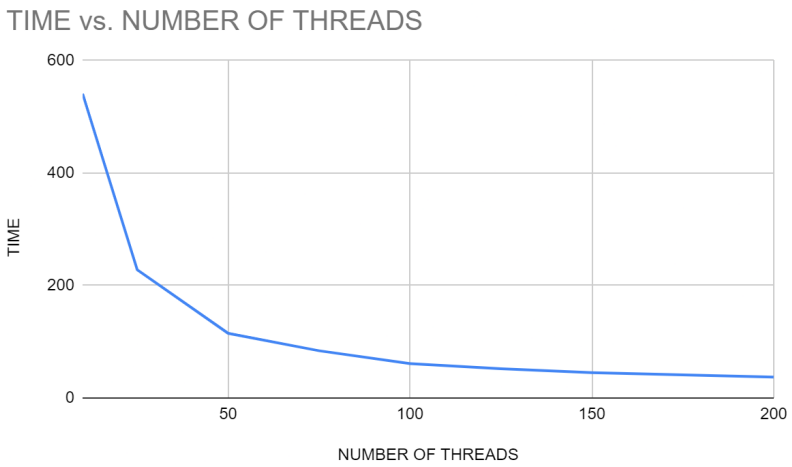
SIZE OF THE MATRIX	2	4	6
6	265	345	516
7	771	823	987
8	5004	2836	3790
9	37554	24452	26174
10	273378	178505	216886



- **BRUTE FORCE RESULTS (MPI)**

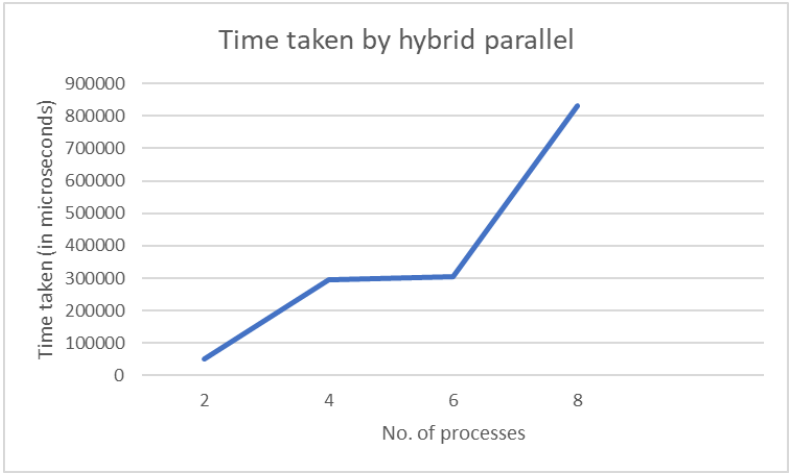
For size of matrix as 8, the serial time taken was 8457 microseconds

NUMBER OF PROCESSES	TIME
10	541
25	228
50	115
75	84
100	61
125	52
150	45
175	41
200	37



- **BRUTE FORCE RESULTS (OPENMP AND MPI)**

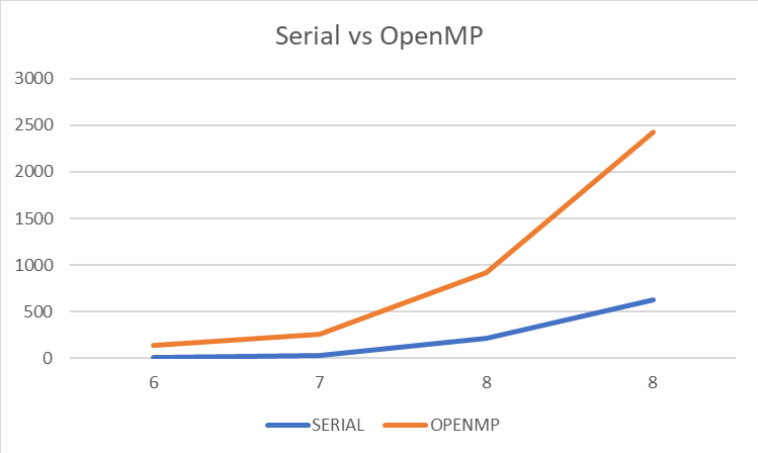
NUMBER OF PROCESSES	TIME
2	52306
4	295177
6	303997
8	832854



- **BRANCH AND BOUND RESULTS (OPENMP)**

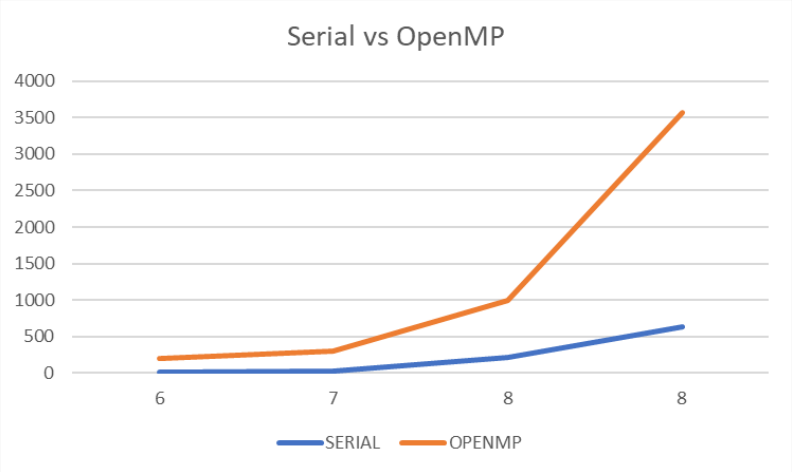
Number of threads = 2

SIZE OF THE MATRIX	SERIAL	OPENMP
6	15	137
7	31	260
8	216	922
9	632	2428



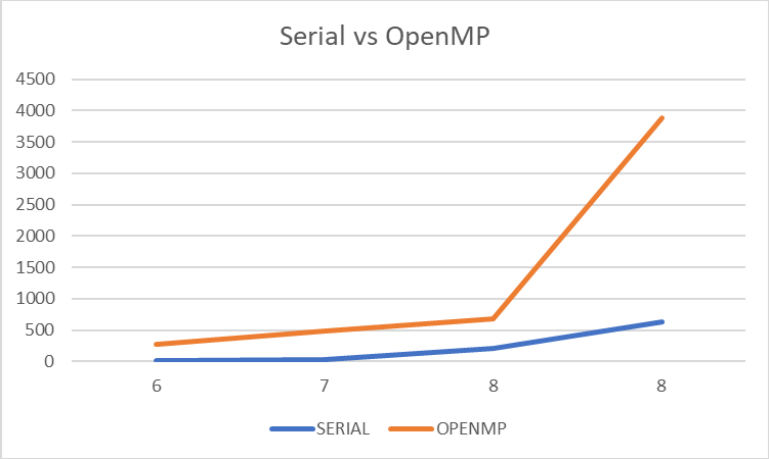
Number of threads = 4

SIZE OF THE MATRIX	SERIAL	OPENMP
6	15	205
7	31	298
8	216	993
9	632	3570



Number of threads = 6

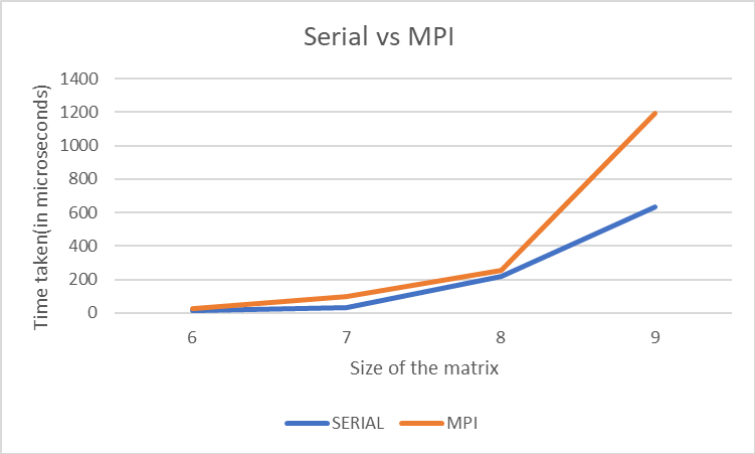
SIZE OF THE MATRIX	SERIAL	OPENMP
6	15	271
7	31	493
8	216	685
9	632	3891



- **BRANCH AND BOUND RESULTS (MPI)**

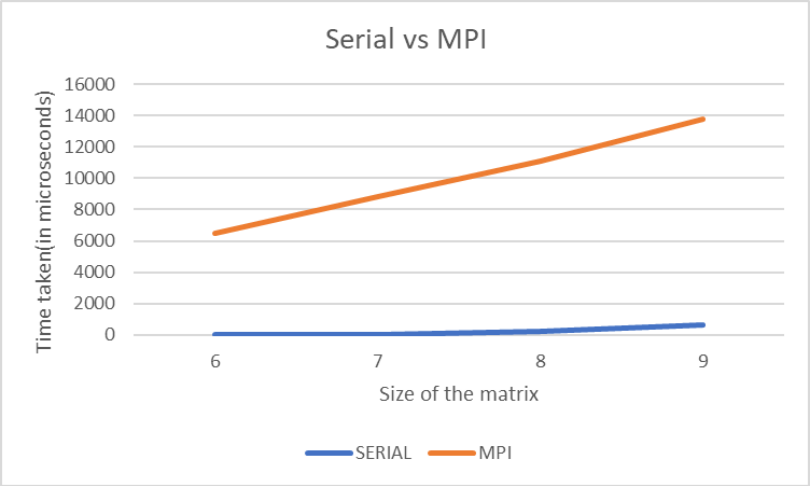
Number of threads = 2

SIZE OF THE MATRIX	SERIAL	OPENMP
6	15	27
7	31	97
8	216	252
9	632	1191



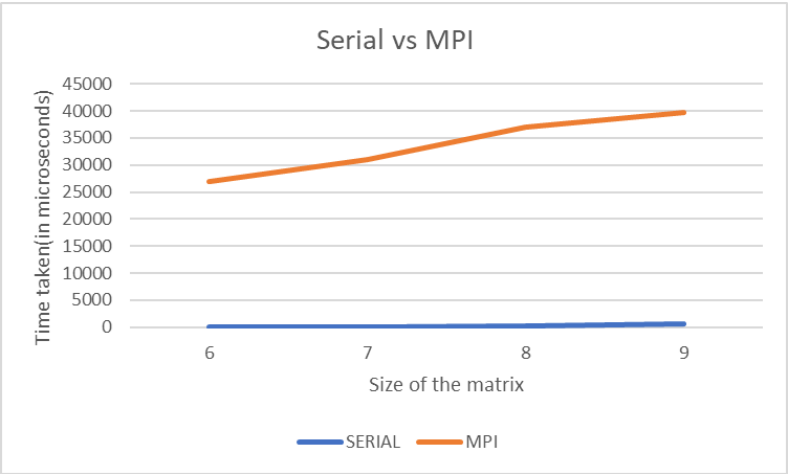
Number of threads = 4

SIZE OF THE MATRIX	SERIAL	OPENMP
6	15	6455
7	31	8839
8	216	11121
9	632	13773



Number of threads = 6

SIZE OF THE MATRIX	SERIAL	OPENMP
6	15	27008
7	31	31094
8	216	36963
9	632	39799



- In the previous tables, we see that the execution times using MPI is much much larger than the serial execution times, similarly for OpenMP as well.
- This is because dynamic programming can not be parallelized as in dynamic programming, the next iteration depends on the last.
- So, when we parallelize it, the overhead increases a lot, hence the execution time increases.

CONCLUSION AND FUTURE WORK

Thus the TSP was studied and more specifically the different approaches to the problem (Brute force and Dynamic programming) was looked at in detail. These approaches were also parallelized using OpenMp, MPI and a combination of the two and the execution times for these were plotted and compared to the execution times of the serial algorithms.

For OpenMP, the number of threads and number of nodes were varied and in MPI, the number of processes and nodes were varied and results tabulated.

Using this study, we can further extend this algorithm and optimize its real world applications like the warehousing problem, task scheduling, computer wiring etc. A TSP may be used to discover the shortest route for the vehicle with the quickest pickup time for the warehousing problem.

It can also be applied in airplane crew scheduling. Each flight leg is unique with its source, destination, arrival and departure time. We also have crew stationed in different cities. We need to allocate crew people to flights such that at the end of their duty (ideally), every crew member is back at their home airport. So, assigning crew to flights can be completely mapped to the traveling salesman problem and can be parallelized.