Aishwarya Ganesh
Ankith C Kowshik

Parallel Processing: Assignment 3

Objective:

The goal of the assignment is to compute a solution to the Travelling Salesman Problem (TSP), using parallel search and MPI.

MPI calls used to solve the given problem are:

MPI Calls	Description
int MPI_Init(int *argc, char ***argv)	used to Initialize the MPI execution environment
Int MPI_Comm_size(MPI_Comm comm, int *size)	Determines the size of the group associated with a communicator
int MPI_Comm_rank(MPI_Comm comm, int *rank)	Determines the rank of the calling process in the communicator
double MPI_Wtime()	Returns an elapsed time on the calling processor.
int MPI_Comm_free(MPI_Comm *comm)	Marks the communicator object for deallocation
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)	Sends personalized data from one process to all other processes in a communicator
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)	Gathers values from a group of processes
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)	Broadcasts a message from the process with rank "root" to all other processes of the communicator
int MPI_Barrier(MPI_Comm comm)	Blocks until all processes in the communicator have reached this routine.
int MPI_Finalize()	Terminates MPI execution environment

Aishwarya Ganesh
Ankith C Kowshik

Formulation:

The **travelling salesman problem** asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" The problem belongs to NP hard category of problems. Our current implementation uses a parallel search algorithm to solve the TSP problem using MPI. The algorithm works by distributing the given input to the various nodes in the topology. Each node runs an identical sequential algorithm on the different branches of the solution space tree. When a branch's cost is worse than the best solution found up to that point, it is cut off and the search for the optimal path on that branch is stopped. Thus, at the end only the optimal solution remains with the rest being cut off at various parts of the execution.

The code requires an input file that contains the list of nodes and their x and y coordinates. Examples for this type of file can be found at http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/
The program was run for the following set of inputs

n, size of the TSP: 101, 666, 1173, 7397, 33810, 85900
p, number of processors: p = 1, 2, 4, 8, 16, 32, 64

Results:

eli 101:

	Processing Time	Communication Time	Total Time
n=1, p=1	0.000129	0.000027	0.001988
n=2, p=1	0.000082	0.002458	0.003365
n=2, p=2	0.000063	0.006458	0.009547
n=2, p=4	0.000054	0.002653	0.006355
n=4, p=4	0.000057	0.005517	0.009952
n=4, p=8	0.000069	0.007883	0.013085
n=4, p=16	0.000108	2.011602	2.013027

gr666:

	Processing Time	Communication Time	Total Time
n=1, p=1	0.004493	0.000114	0.016034
n=2, p=1	0.002354	0.014169	0.033756
n=2, p=2	0.001301	0.025276	0.045039
n=2, p=4	0.000799	0.033977	0.056909
n=4, p=4	0.000659	0.042018	0.059614
n=4, p=8	0.000606	0.050347	0.075852

pcb1173:

Aishwarya Ganesh
Ankith C Kowshik

	Processing Time	Communication Time	Total Time
n=1, p=1	0.01374	0.000212	0.033966
n=2, p=1	0.007092	0.024836	0.062469
n=2, p=2	0.00387	0.031944	0.069388
n=2, p=4	0.002318	0.028108	0.058921
n=4, p=4	0.001521	0.067329	0.104897
n=4, p=8	0.001355	0.062458	0.101076
n=4, p=16	0.001725	23.741331	23.758017

pla7397:

	Processing Time	Communication Time	Total Time
n=1, p=1	0.518396	0.001305	0.549574
n=2, p=1	0.271307	0.009221	0.311025
n=2, p=2	0.311025	0.188242	0.423404
n=2, p=4	0.075302	0.196105	0.365586
n=4, p=4	0.040762	0.346259	0.518391
n=4, p=8	0.025059	0.396906	0.552282
n=4, p=16	0.019467	147.940643	148.054062

pla33810:

	Processing Time	Communication Time	Total Time
n=1, p=1	10.80124	0.008444	10.943254
n=2, p=1	5.41642	0.990614	6.832498
n=2, p=2	2.839455	0.897225	4.187331
n=2, p=4	1.503437	0.919864	2.884327
n=4, p=4	0.763169	1.760659	3.139396
n=4, p=8	0.415361	1.7027	2.735716

pla85900:

	Processing Time	Communication Time	Total Time
n=1, p=1	69.87262	0.026899	70.186012
n=2, p=1	35.190517	2.27734	40.563286
n=2, p=2	18.201206	2.846655	22.274071
n=2, p=4	9.721385	3.277199	13.19631
n=4, p=4	4.917103	4.08457	10.634117
n=4, p=8	2.564602	4.520848	8.662455

Aishwarya Ganesh
Ankith C Kowshik

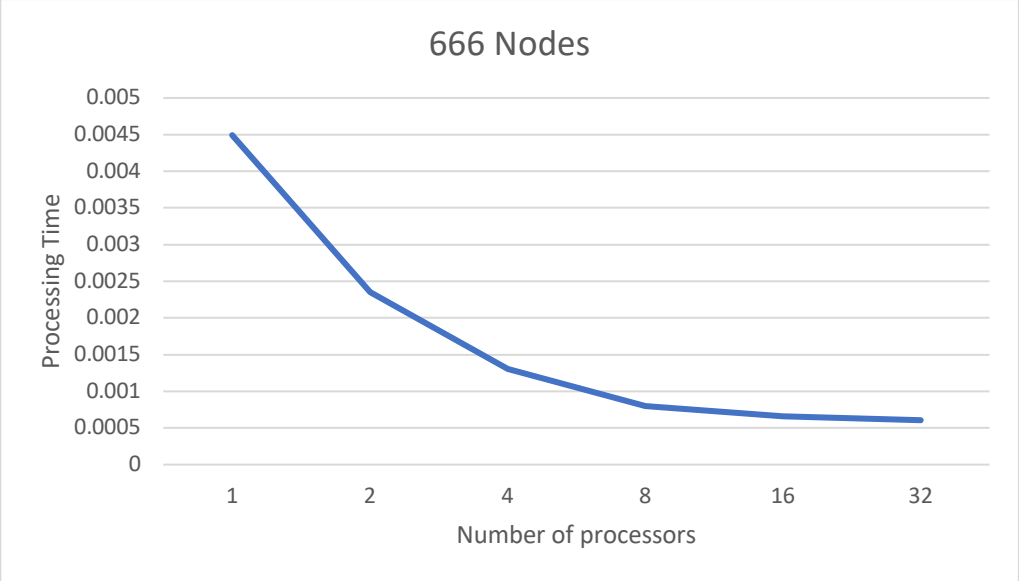

Figure 1

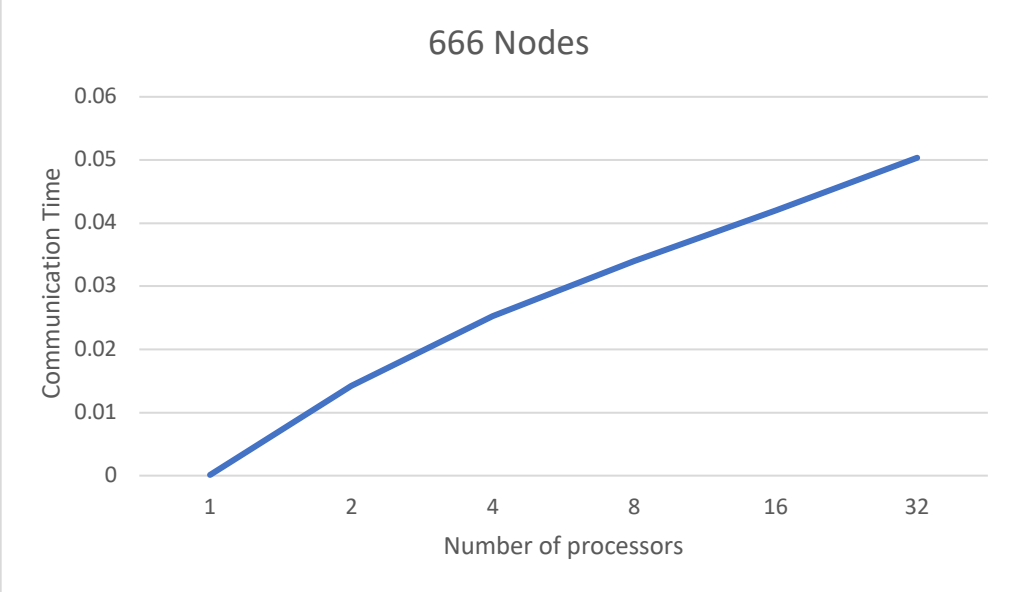

Figure 2

Aishwarya Ganesh
Ankith C Kowshik

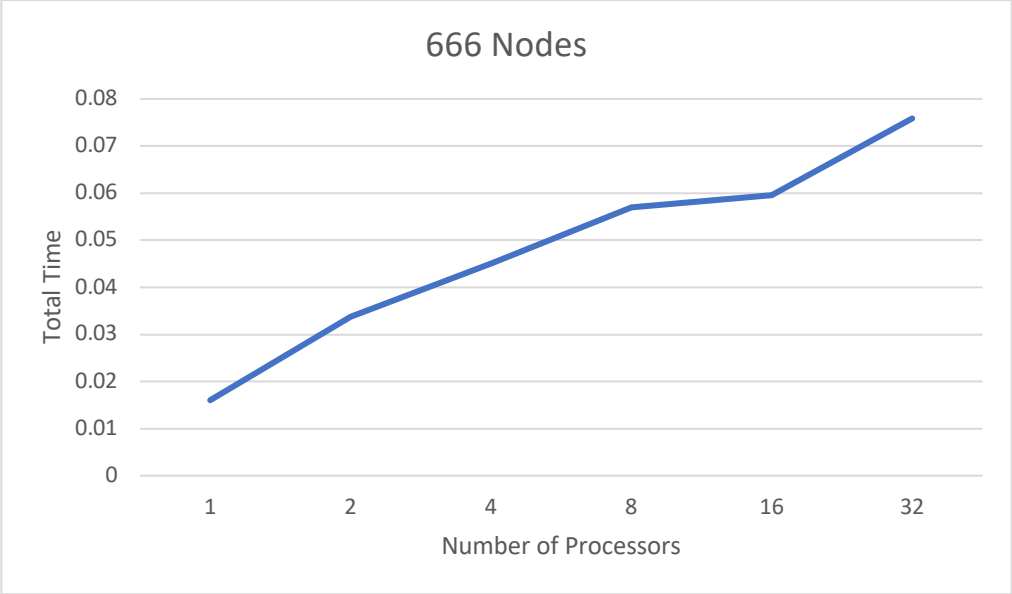

Figure 3

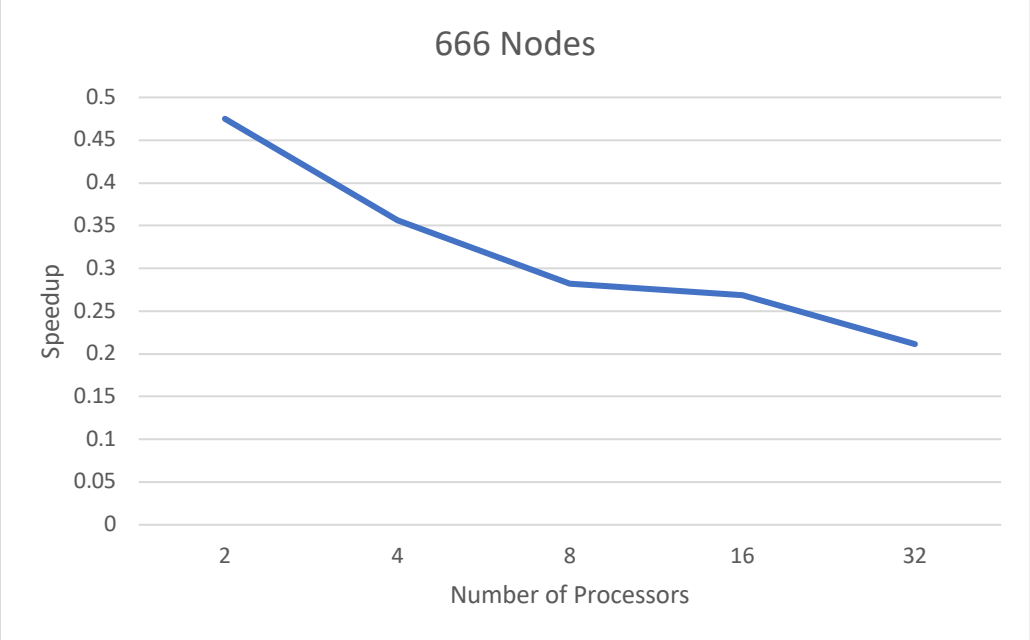

Figure 4

Aishwarya Ganesh
Ankith C Kowshik

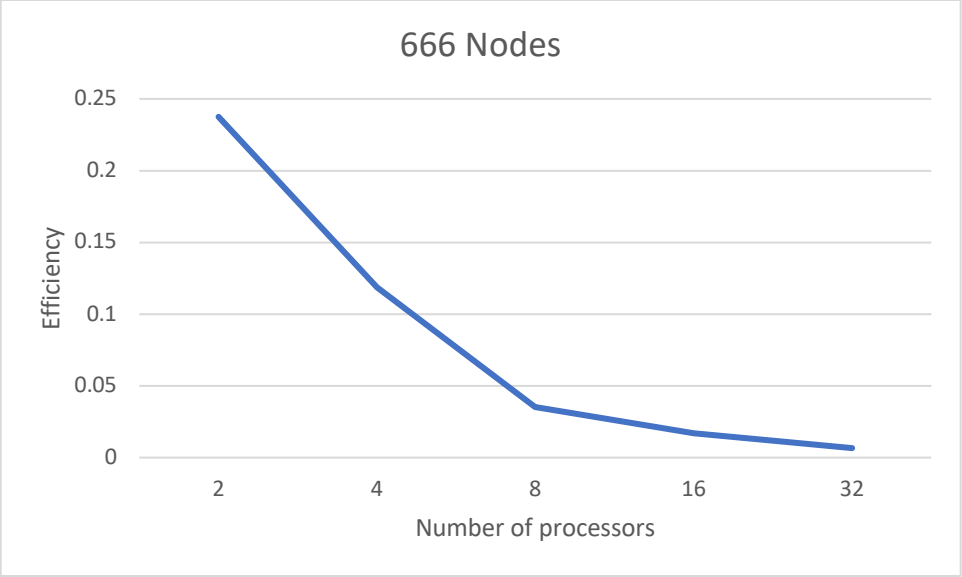

Figure 5

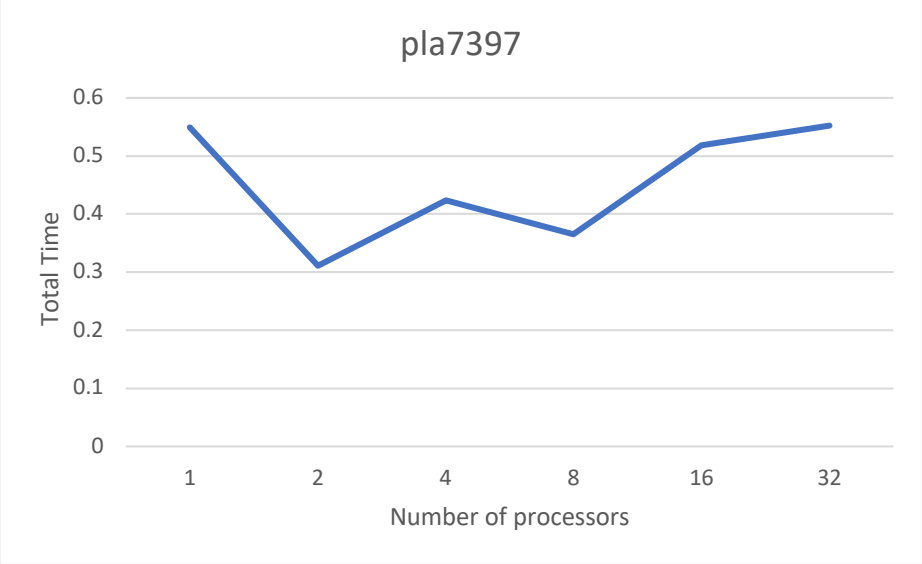

Figure 6

Aishwarya Ganesh
Ankith C Kowshik

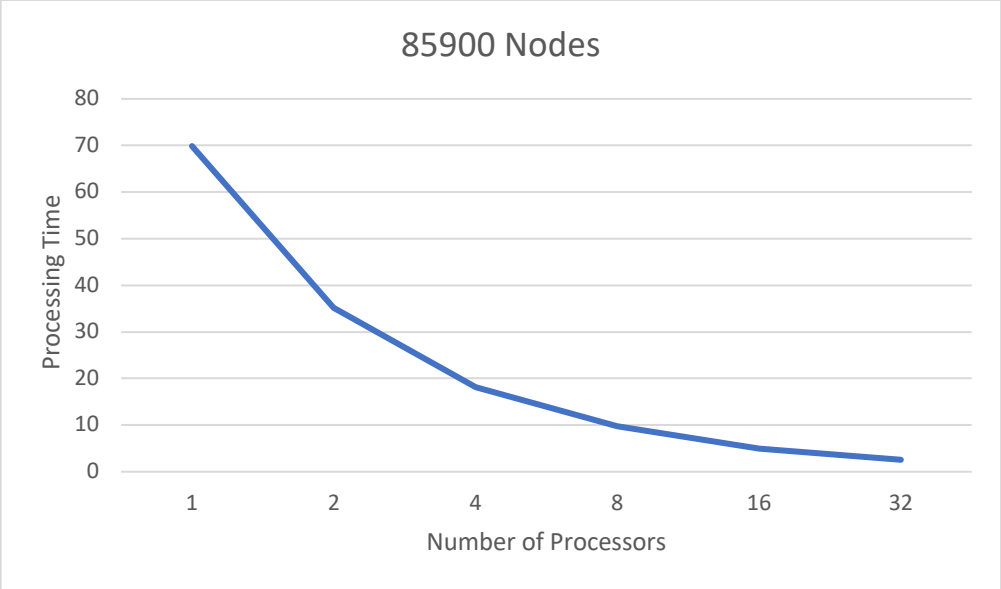

Figure 7

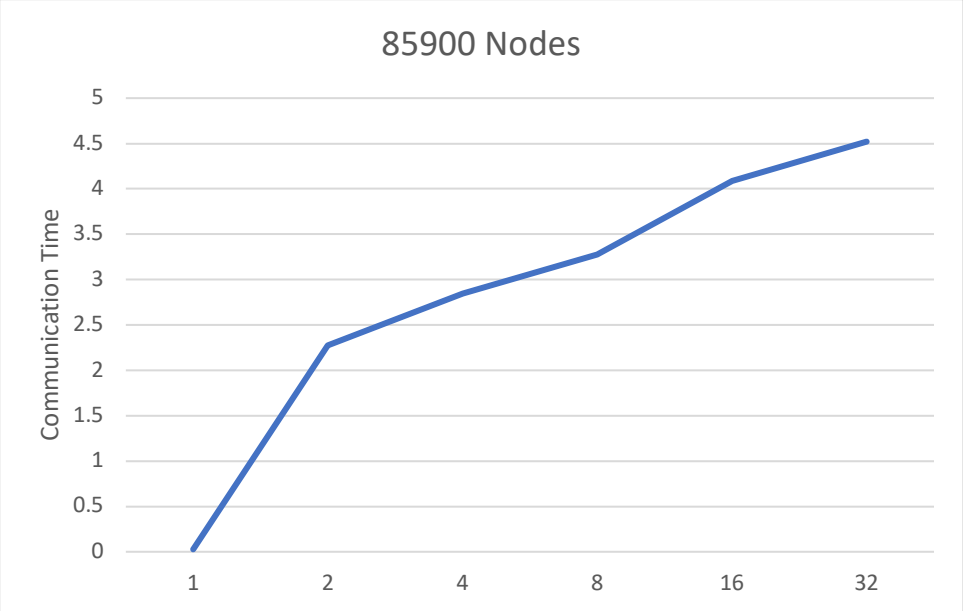

Figure 8

Aishwarya Ganesh
Ankith C Kowshik

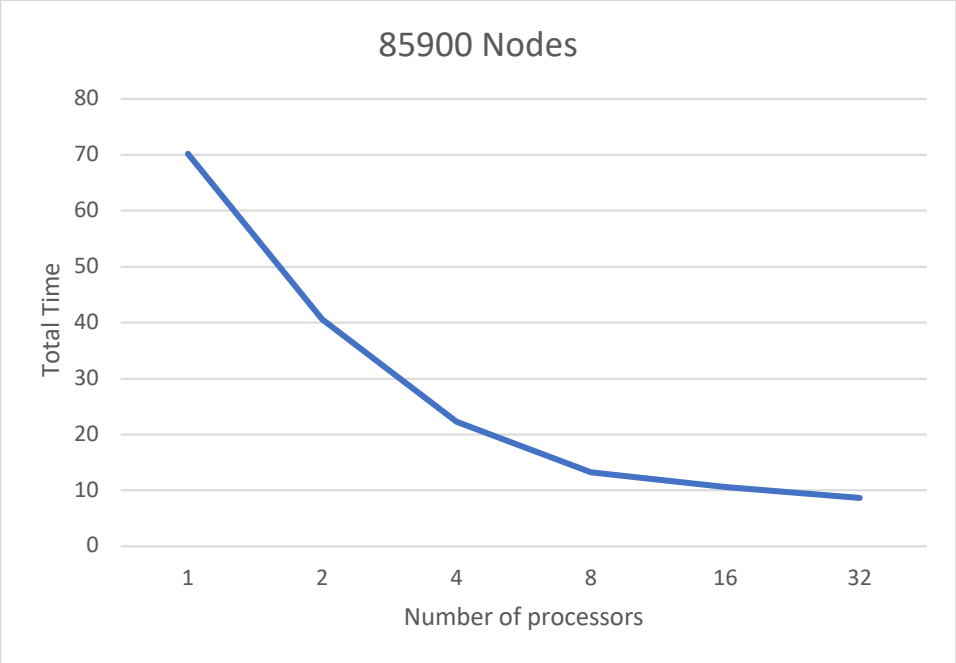

Figure 9

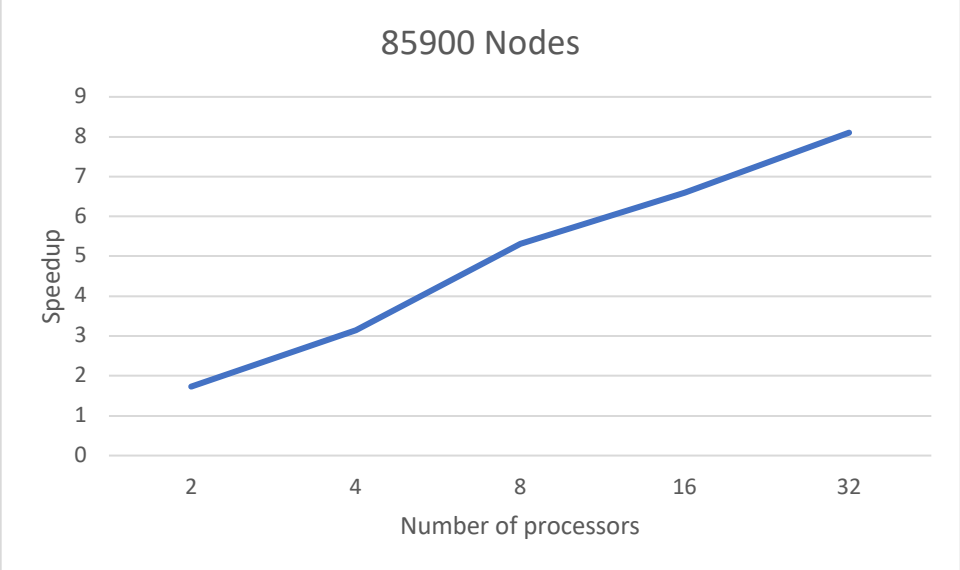

Figure 10

Aishwarya Ganesh
Ankith C Kowshik

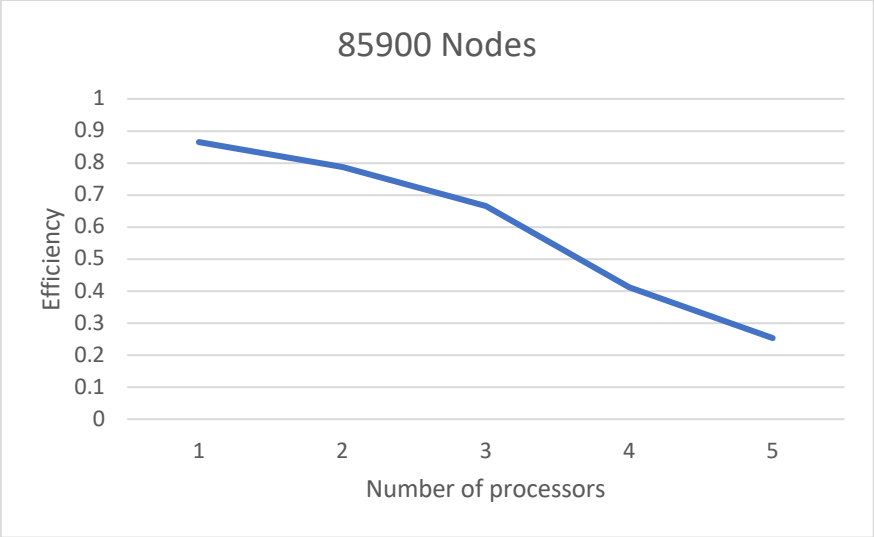

Figure 11

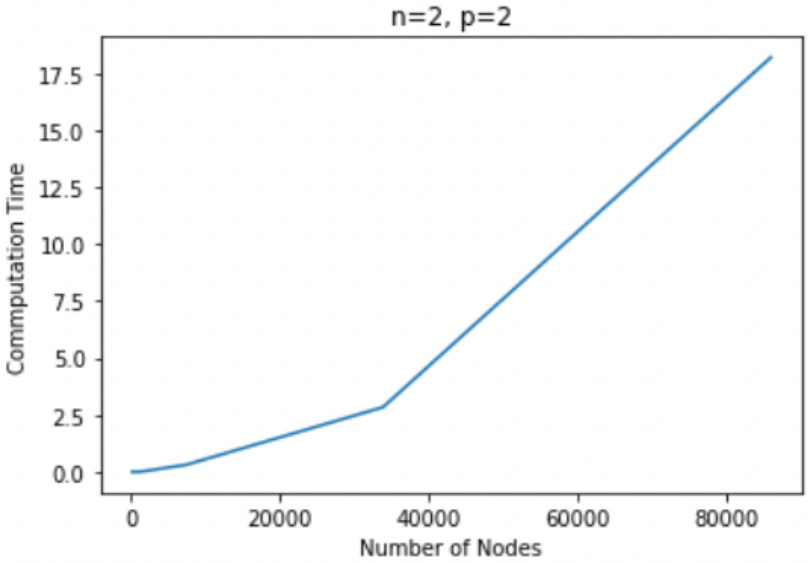

Figure 12

Aishwarya Ganesh
Ankith C Kowshik

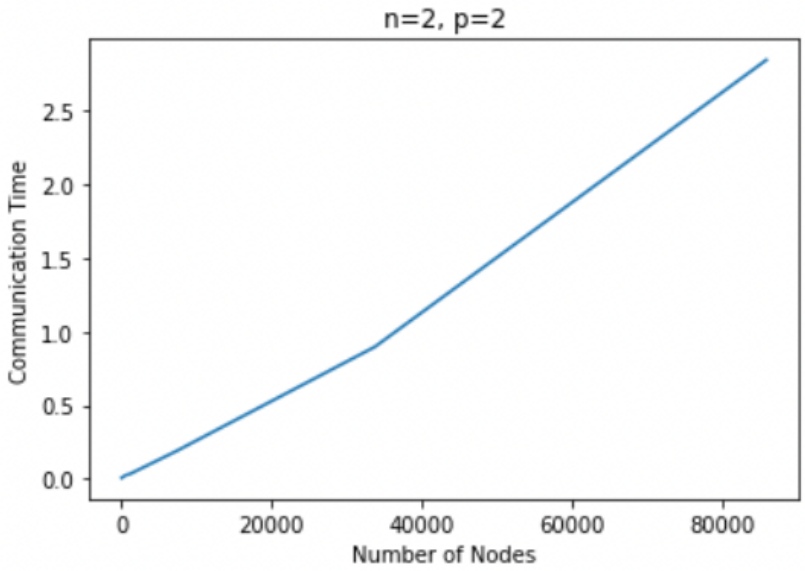

Figure 13

Analysis:

- From figure 1 and figure 7 of our results we can see that there is a linear increase in the computation time with an increase in the number of processors for any given input data size.
- Similarly, there is a linear increase in the communication time with the increase in number of processors. This can be observed from figures 2 and 8.
- From figure 3 we see that for smaller data sets where the number of nodes to be traversed are less, with an increase in the number of processors, the increase in communication time is greater than the decrease in computation time. Thus, the overall time to complete execution increases. This is due to the already small computation time.
- The true efficiency and speedup of the algorithm can be observed when larger datasets are considered. In situations such as this, the decrease in computation time is significant compared to the increase in the communication time as we increase the number of processors. This gives us an almost linear decrease in total execution time with an increase in the number of processors as evident from figure 9.
- From Figure 6, for the current implementation of the algorithm, at around 8000 nodes, the tradeoff between increase in communication time and decrease in speedup balance out. Thus, to achieve good results the input dataset must consist of at least 8000 nodes.
- Similarly, from figures 4 and 10 and we see that for small input sizes there is a linear decrease in speedup and efficiency due to communication overheads, while a linear increase is observed to larger input sizes.

Aishwarya Ganesh
Ankith C Kowshik

Lesson Learnt:

- In this lab assignment we learnt how to solve the Travelling Salesman Problem using a parallel formulation and analyzed the results.
- We understood the factors that affect the time taken to reach a solution for TSP and that for certain input sizes, it is better to use fewer processors as the communication overhead far exceeds the time gained by performing fewer computations.
- The assignment helped us understand the parallel search strategy that can be applied to solve various problems using a parallel formulation
- For large input sizes, there is a linear increase in the speedup achieved when using the parallel search strategy to solve TSP.

```c
#include <float.h>
#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#define TSP_FILE "gr666.tsp"
#define DIST(a,b) sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2))

int n = 666;
float proc_time = 0.0, comm_time = 0.0, total_time = 0.0;

typedef struct
{
   int visited;
   float x, y;
} LOCATION;
LOCATION *locs;

int nearest(int curr, int start, int end)
{
   int i, index = -1;
   float min = FLT_MAX;
   float dist;

   for(i = start; i<= end; ++i) {
      dist = DIST(locs[curr], locs[i]);
      if(dist < min && i != curr && locs[i].visited == 0) {
         min = dist;
         index = i;
      }
   }
   return index;
}

void pp_tsp(int rank, int world_size)
{
   int i, j, index, start_loc, end_loc, next;
   int loc_per_n = n / world_size;
   int *inm;
   int travel_path[n];
   double start, end, dt;
   float min = FLT_MAX;
   float dist;
   float cost = 0.0f;
   inm = (int*)malloc(sizeof(int) * world_size);
   start_loc = loc_per_n * rank;
   end_loc = start_loc + loc_per_n - 1;

  if(rank == world_size - 1)
      end_loc += n % world_size;
   next = 0;
   travel_path[0] = 0;
   for(i = 0; i < n - 1; i++) {
      start = MPI_Wtime();
      MPI_Bcast(&next, 1, MPI_INT, 0, MPI_COMM_WORLD);
      end = MPI_Wtime();
      dt = end - start;
      comm_time += dt;
      start = MPI_Wtime();
      locs[next].visited = 1;
      index = nearest(next, start_loc, end_loc);
      end = MPI_Wtime();
      dt = end - start;
```

```
        proc_time += dt;
        start = MPI_Wtime();
        MPI_Gather(&index, 1, MPI_INT, inm, 1, MPI_INT, 0, MPI_COMM_WORLD);
        end = MPI_Wtime();
        dt = end - start;
        comm_time += dt;
        if(rank == 0)
        {
            start = MPI_Wtime();
            index = inm[0];
            min = FLT_MAX;
            for(j = 0; j < world_size; ++j)
            {
                if(inm[j] < 0)
                    continue;
                dist = DIST(locs[next], locs[inm[j]]);
                if(dist < min)
                {
                    min = dist;
                    index = inm[j];
                }
            }
            next = index;
            travel_path[i + 1] = index;
            end = MPI_Wtime();
            dt = end - start;
            proc_time += dt;
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    if(rank == 0) {
        printf("\nFinal Optimal Travel path is:\n");
        for(i = 0; i < n; ++i)
            printf("%d ", travel_path[i]);
        printf("\n");
    }
    free(inm);
}

void parse_input_file()
{
    FILE *fp;
    float x, y;
    int i, line;
    char buff[1024];

    fp = fopen(TSP_FILE, "r");

    for(i = 0; i < 9; i++)
        fgets(buff, 1024, fp);

    while(fscanf(fp, "%d %f %f", &line, &x, &y) > 0 ) {
        if(line == n) {
            break;
        }
    }

    locs = (LOCATION *) malloc(sizeof(LOCATION) * line);
    rewind(fp);

    for(i = 0; i < 9; i++)
        fgets(buff, 1024, fp);

    while(fscanf(fp, "%d %f %f", &line, &x, &y) > 0  && line <= n) {
        locs[line - 1].x = x;
```

```
        locs[line - 1].y = y;
        locs[line - 1].visited = 0;
    }
    fclose(fp);
}

int main(int argc, char **argv)
{
    int i;
    double t_start = 0, t_end = 0;
    int rank, world_size;

    parse_input_file();
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    t_start = MPI_Wtime();
    pp_tsp(rank, world_size);
    t_end = MPI_Wtime();
    total_time = t_end - t_start;
    if(rank == 0)
        printf("\nFor %d cities with %d procs processing time is %f, communication time is %f"
               " total time taken is %f", n, world_size, proc_time, comm_time, total_time);

    free(locs);
    MPI_Finalize();
    return 0;
}
```