CSE4001-Parallel and distributed computing

Name: O G RAGAVI

Reg No: 20BCE1988

Lab Ex: 10

Title: Dijshktra's algorithm implementation using open MP and mpi

Open MP implementation:

Code:

```
#include<omp.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include<time.h>
// Number of vertices in the graph
#define V 9

int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
```

```c
                    #pragma omp critical
                      min = dist[v], min_index = v;

        return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
        printf("Vertex \t\t Distance from Source\n");
        for (int i = 0; i < V; i++)
                printf("%d \t\t\t\t %d\n", i, dist[i]);
}


void dijkstra(int graph[V][V], int src)
{
        int dist[V];

        bool sptSet[V];
```

```
for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;



dist[src] = 0;

#pragma omp parallel for

for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;



        for (int v = 0; v < V; v++)



                if (!sptSet[v] && graph[u][v]
                        && dist[u] != INT_MAX
                        && dist[u] + graph[u][v] < dist[v])
```

```
                        dist[v] = dist[u] + graph[u][v];
    }



    printSolution(dist);
}

// driver's code
int main()
{

    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    // Function call
```

```
    clock_t s,e;
   s=clock();
    dijkstra(graph, 0);
    e=clock();
     printf("\nfor a 9x9 adjacency matrix:");
   printf("\nTime taken %ld \n",(e-s));


    return 0;
}
```

Output:

```
ragavi@RAGAVI:~$ ./a.out
Vertex           Distance from Source
0                                0
1                                4
2                                12
3                                19
4                                21
5                                11
6                                9
7                                8
8                                14

for a 9x9 adjacency matrix:
Time taken 527
```

```
ragavi@RAGAVI:~$ gcc -fopenmp lab10a.c
ragavi@RAGAVI:~$ ./a.out
Vertex           Distance from Source
0                                0
1                                1
2                                1
3                                1

for a 4x4 adjacency matrix:
Time taken 50
ragavi@RAGAVI:~$
```

2.MPI Implementation:

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include "mpi.h"
#define N 5
#define SOURCE 1
#define MAXINT 9999999
void SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm
comm)
{
 int temp[N];
 int i, j;
 int nlocal; /* The number of vertices stored locally */
 int *marker; /* Used to mark the vertices belonging to Vo */
 int firstvtx; /* The index number of the first vertex that is stored locally
*/
 int lastvtx; /* The index number of the last vertex that is stored locally
*/
 int u, udist;
```

```
int lminpair[2], gminpair[2];
int npes, myrank;
MPI_Status status;
MPI_Comm_size(comm, &npes);
MPI_Comm_rank(comm, &myrank);
nlocal = n / npes;
firstvtx = myrank * nlocal;
lastvtx = firstvtx + nlocal - 1;
/* Set the initial distances from source to all the other vertices */
for (j = 0; j < nlocal; j++)
{
lengths[j] = wgt[source * nlocal + j];
}
/* This array is used to indicate if the shortest part to a vertex has been
found
or not. */
/* if marker [v] is one, then the shortest path to v has been found,
otherwise it
has not. */
marker = (int *)malloc(nlocal * sizeof(int));
for (j = 0; j < nlocal; j++)
{
marker[j] = 1;
```

```
}
/* The process that stores the source vertex, marks it as being seen */
if (source >= firstvtx && source <= lastvtx)
{
marker[source - firstvtx] = 0;
}
/* The main loop of Dijkstra's algorithm */
for (i = 1; i < n; i++)
{
/* Step 1: Find the local vertex that is at the smallest distance from
source */
lminpair[0] = MAXINT; /* set it to an architecture dependent large
number
*/
lminpair[1] = -1;
for (j = 0; j < nlocal; j++)
{
if (marker[j] && lengths[j] < lminpair[0])
{
lminpair[0] = lengths[j];
lminpair[1] = firstvtx + j;
}
}
```

```
/* Step 2: Compute the global minimum vertex, and insert it into Vc */
MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC, comm);
udist = gminpair[0];
u = gminpair[1];
/* The process that stores the minimum vertex, marks it as being seen
*/
if (u == lminpair[1])
{
marker[u - firstvtx] = 0;
}
/* Step 3: Update the distances given that u got inserted */
for (j = 0; j < nlocal; j++)
{
if (marker[j] && ((udist + wgt[u * nlocal + j]) < lengths[j]))
{
lengths[j] = udist + wgt[u * nlocal + j];
}
}
}
free(marker);
}
int main(int argc, char *argv[])
```

```
{
 int npes, myrank, nlocal;
 int weight[N][N]; /*adjacency matrix*/
 int distance[N]; /*distance vector*/
 int *localWeight; /*local weight array*/
 int *localDistance; /*local distance vector*/
 int sendbuf[N * N]; /*local weight to distribute*/
 int i, j, k;
 char fn[255];
 double time_start, time_end;
 struct timeval tv;
 struct timezone tz;
 gettimeofday(&tv, &tz);
 time_start = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;
 /* Initialize MPI and get system information */
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &npes);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 nlocal = N / npes; /* Compute the number of elements to be stored
locally. */
 /*allocate local weight and local disatance arrays for each prosess*/
 localWeight = (int *)malloc(nlocal * N * sizeof(int));
```

```
localDistance = (int *)malloc(nlocal * sizeof(int));
/* Open input file, read adjacency matrix and prepare for sendbuf */
// printf("\nThe adjacency matrix: \n");
for (i = 0; i < N; i++)
{
for (j = 0; j < N; j++)
{
weight[i][j] = rand() % 11;
// if (weight[i][j] == 9999999) printf("%4s", "INT");
// else printf("%4d", weight[i][j]);
}
// printf("\n");
}
/*prepare send data */
for (k = 0; k < npes; ++k)
{
for (i = 0; i < N; ++i)
{
for (j = 0; j < nlocal; ++j)
{
sendbuf[k * N * nlocal + i * nlocal + j] = weight[i][k * nlocal + j];
}
```

```
}
}
/*distribute data*/
MPI_Scatter(sendbuf, nlocal * N, MPI_INT, localWeight, nlocal * N,
MPI_INT, SOURCE,
MPI_COMM_WORLD);
/*Implement the single source dijkstra's algorithm*/
SingleSource(N, SOURCE, localWeight, localDistance,
MPI_COMM_WORLD);
/*collect local distance vector at the source process*/
MPI_Gather(localDistance, nlocal, MPI_INT, distance, nlocal, MPI_INT,
SOURCE,
MPI_COMM_WORLD);
if (myrank == SOURCE)
{
printf("Nodes: %d\n", N);
printf("The distance vector is \n");
for (i = 0; i < N; ++i)
{
printf("%d ", distance[i]);
}
printf("\n");
```

```
gettimeofday(&tv, &tz);

time_end = (double)tv.tv_sec + (double)tv.tv_usec / 1000000.00;

printf("time cost is %1f\n", time_end - time_start);

}

free(localWeight);

free(localDistance);

MPI_Finalize();

return 0;

}
```

Output:

```
ragavi@RAGAVI:~$ mpicc -o x x.c
ragavi@RAGAVI:~$ mpirun -np 2 ./x
Nodes: 5
The distance vector is
4 0 6 3 0
time cost is 0.230125
```