PARALLEL IMPLEMENTATION OF DJIKSTRA’S ALGORITHM TO INVESTIGATE NATIONAL CITY TRAFFIC

Submitted by

NAME	SECTION	ROLL NUMBER	REG. NUMBER	SEMESTER
Sandesh Hebbar	B	02	160905018	6
Sanket Srivastava	B	31	160905055	6
Sanchit Khetawat	B	32	160905208	6

Department of Computer Science & Engineering

MANIPAL INSTITUTE OF TECHNOLOGY
A Constituent Institute of Manipal University, Manipal

1. ABSTRACT

In the intelligent transportation system, the calculation of the shortest path and the best path is an important link of the vehicle navigation procedure. Due to more and more real-time information to participate in the calculation, the calculation requires high efficiency of the algorithm. One of the many common algorithms in solving the shortest path problem is Dijkstra's algorithm. The algorithm has its advantages on both reducing the number of repeated operations and reading the shortest path and the path length from the startpoint to all the other nodes by the shortest path tree or by the feature matrix. In this paper, we show that the parallel implementation of Djikstra’s algorithm, which is implemented in both the Message Passing Interface and CUDA parallel implementation platforms, provides better efficiency and better speedup when compared to the algorithm’s sequential execution.

2. OBJECTIVES

To analyze the serial implementation of the established Djikstra’s algorithm and develop its equivalent parallel implementation in two parallel programming platforms namely, MPI, Message Passing Interface, and CUDA.

3. INTRODUCTION

Passengers consider multiple factors when it comes to deciding a route they have to take to reach a pre-planned destination. Some factors can be: the time taken to reach the destination or the cost-incurred. Since there can be multiple factors, the weights in a weighted graph are representative of all the factors combined. In our implementation, users choose a source vertex and a destination vertex. The output will be the optimal path that the user should take to reach their destination along with the total cost incurred.

4. LITERATURE REVIEW

In 1959, Dijkstra proposed a graph search algorithm that can be used to solve the single-source shortest path problem for any graph that has a non-negative edge path cost. This graph search algorithm was later modified by Lee in 2006 and was applied to the Vehicle Guidance System. The shortest path algorithm focuses on route length parameter and calculates the shortest route between each pair of nodes. In 2012, Meghanathan reviewed Dijkstra’s algorithm for finding the shortest path in a graph. He concluded that the time complexity of Dijkstra’s algorithm is O (|E|*log |V|). Based on the referenced paper by Arun Kumar Sangaiah, Minghao Han, and Suzi Zhang, who have applied the algorithm to practice in national traffic advisory procedures, we aim to improve on their serial code by parallelizing it.

5. METHODOLOGY

Classified as a Greedy Algorithm, Djikstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a given graph with non-negative edge path costs, and yields an optimal path. The serial algorithm is provided below:

Algorithm 1 Dijkstra's algorithm for shortest paths problem

```
Require: $N$ as size of the graph
1:  function DIJKSTRA($s$)
2:      $OK[s] \leftarrow$ true
3:      for all $i$ do
4:          $D[i] \leftarrow (s, i)$
5:      loop
6:          x $\leftarrow i$ where $D[i]$ is minimum for $OK[i]$ is false
7:          $OK[x] \leftarrow$ true
8:          for all $i$ where $OK[i]$ is false do
9:              if $D[i] > D[x] + (x, i)$ then
10:                 $D[i] \leftarrow D[x] + (x, i)$
11:                 Indicate that we came to $i$ from $x$
```

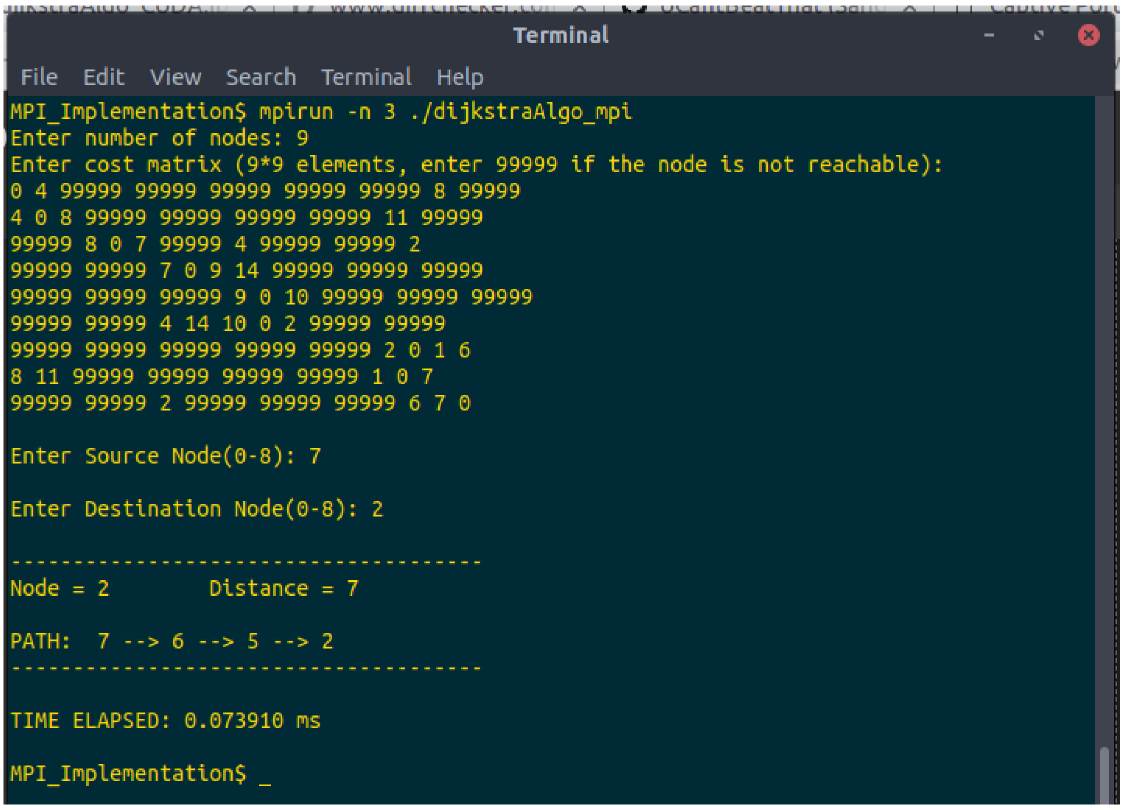
As there is a part where the closest of the nodes is searched which is depicted by the outer for-loop and there is a part where the alternating paths are calculated for each node which is depicted by the inner for-loop wherein the latter can be calculated independently on all the nodes. Hence, we parallelize it.

6. RESULTS

//MPI_Implementation

```
#include<stdio.h>
#include <mpi.h>
#include<stdlib.h>
#include<stdbool.h>
#include <limits.h>
#define MAX_SIZE 20 //max number of nodes

int G[MAX_SIZE][MAX_SIZE];  //adjacency matrix
bool visited[MAX_SIZE]; //nodes done
int D[MAX_SIZE]; //distance
int path[MAX_SIZE]; //we came to this node from
int N; //actual number of nodes
int size, rank;
MPI_Status status;
int flag;
int all_paths[MAX_SIZE];
int readAdjacencyMatrix(){
    printf("Enter number of nodes: ");
    scanf("%d", &N);

        printf("Enter cost matrix (%d*%d elements, enter 99999 if the node is not
reachable):\n", N, N);
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
          scanf("%d", &G[i][j]);
        }
    }

    return N;
}

void dijkstraAlgorithm(int s){
    int i,j;
    int tmp, x;
    int pair[2];
    int tmp_pair[2];

    //initialize all the nodes
    for(i=rank;i<N;i+=size){
        D[i]=G[s][i];
        visited[i]=false;
        path[i]=s;
    }

    //set src as visited node
    visited[s]=true;
    path[s]=-1;

    //compute the distance of the shortest path from the source node
    for(j=1;j<N;j++){
        x=99999;
        tmp=99999;
        for(i=rank;i<N;i+=size){
            if(!visited[i] && D[i]<tmp){ //find the neighbour node with least distance
                x=i;
                tmp=D[i];
            }
        }
        //store that neighbour node and its corresponding distance in an array
        pair[0]=x;
        pair[1]=tmp;
        //compute the global minimum of distances obtained from all the processes
        if(rank!=0){
            MPI_Send(pair,2,MPI_INT,0,rank,MPI_COMM_WORLD);
        }
        else{
            for(i=1;i<size;++i){
                MPI_Recv(tmp_pair,2,MPI_INT,i,i,MPI_COMM_WORLD, &status);
                if(tmp_pair[1]<pair[1]){
                    pair[0]=tmp_pair[0];
                    pair[1]=tmp_pair[1];
                }
            }
        }

        //broadcast the obtained least distance node and distance
        MPI_Bcast(pair,2,MPI_INT,0,MPI_COMM_WORLD);
        x=pair[0];
        D[x]=pair[1];
        visited[x]=true; //mark the node as visited

        //check if any other node can be visited efficiently through the obtained node
        for(i=rank;i<N;i+=size){
            if(!visited[i] && D[i]>D[x]+G[x][i]){
                D[i]=D[x]+G[x][i];
                path[i]=x;
            }
        }
    }

    //to obtain the paths from all the processes
        MPI_Reduce(path, all_paths, N, MPI_INT, MPI_MAX, 0,
MPI_COMM_WORLD);
}


int main(int argc, char** argv){

    double t1, t2;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int src_node, dest_node;
    if(rank==0){
        //read the adjacency matrix along with the actual number of nodes
        N = readAdjacencyMatrix();
        printf("\nEnter Source Node(0-%d): ", N-1);
        scanf("%d", &src_node);
        printf("\nEnter Destination Node(0-%d): ", N-1);
        scanf("%d", &dest_node);
        t1=MPI_Wtime();
    }

    //broadcast required data to all the processes
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(G, MAX_SIZE*MAX_SIZE, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&src_node, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //call the algorithm with the choosen node
    dijkstraAlgorithm(src_node);

    if(rank==0){
        t2=MPI_Wtime();
        //check the results with some output from G[][] and D[]
            printf("\n-------------------------------------\nNode = %d \tDistance = %d \n",
dest_node, D[dest_node]);
        int temp;
        temp = dest_node;
        printf("\nPATH: ");
        int req_path[N];
        int count = 0;
        while(all_paths[temp]!=-1){
           req_path[count++] = all_paths[temp];
            //printf(" <-- %d ", req_path[count-1]);
            temp = all_paths[temp];
        }

        for(int i=count-1;i>=0;i--){
           printf(" %d -->", req_path[i]);
        }
        printf(" %d ", dest_node);

        FILE *fin = fopen("output.txt", "a");
         printf("\n-------------------------------------\n\nTIME ELAPSED: %lf ms\n\n",(t2-
t1)*1000);
        fprintf(fin, "%d\t| %lf\n",size, (t2-t1)*1000);
    }

    MPI_Finalize();
}
```

//MPI_Input/Output



```
MPI_Implementation$ mpirun -n 3 ./dijkstraAlgo_mpi
Enter number of nodes: 9
Enter cost matrix (9*9 elements, enter 99999 if the node is not reachable):
0 4 99999 99999 99999 99999 99999 8 99999
4 0 8 99999 99999 99999 99999 11 99999
99999 8 0 7 99999 4 99999 99999 2
99999 99999 7 0 9 14 99999 99999 99999
99999 99999 99999 9 0 10 99999 99999 99999
99999 99999 4 14 10 0 2 99999 99999
99999 99999 99999 99999 99999 2 0 1 6
8 11 99999 99999 99999 99999 1 0 7
99999 99999 2 99999 99999 99999 6 7 0

Enter Source Node(0-8): 7

Enter Destination Node(0-8): 2

--------------------------------------
Node = 2        Distance = 7

PATH:  7 --> 6 --> 5 --> 2
--------------------------------------

TIME ELAPSED: 0.073910 ms

MPI_Implementation$ _
```

//CUDA Implementation

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define MAX_WEIGHT 1000000
#define TRUE    1
#define FALSE   0

typedef int boolean;

typedef struct
{
        int u;
        int v;
} Edge;
```

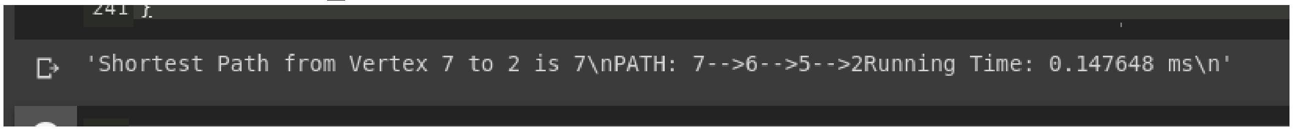
```
typedef struct
{
        int title;
        boolean visited;
}Vertex;

__device__ __host__ int findEdge(Vertex u, Vertex v, Edge *edges, int *weights)
{

        int i;
        for(i = 0; i < E; i++)
        {
                if((edges[i].u == u.title && edges[i].v == v.title) || (edges[i].v == u.title
&& edges[i].u == v.title))
                {
                        return weights[i];
                }
        }

        return MAX_WEIGHT;
}

__global__ void initVertices(Vertex *vertices, Edge *edges, int* weights, int* length,
int* updateLength, Vertex root){

   int i = threadIdx.x;

   if(vertices[i].title != root.title)
                {
                        length[(int)vertices[i].title]  =  findEdge(root,  vertices[i],  edges,
weights);
                        updateLength[vertices[i].title] = length[(int)vertices[i].title];

                }
                else{
                        vertices[i].visited = TRUE;
                }
}

__global__ void findVertex(Vertex *vertices, Edge *edges, int *weights, int *length,
int *updateLength, int* path)
{
        int u = threadIdx.x;

        if(vertices[u].visited == FALSE)
        {

                vertices[u].visited = TRUE;

                int v;
                for(v = 0; v < V; v++)
                {
                        int weight = findEdge(vertices[u], vertices[v], edges, weights);

                        if(weight < MAX_WEIGHT)
                        {

                                if(updateLength[v] > length[u] + weight)
                                {
                              path[v] = u;
                                        updateLength[v] = length[u] + weight;
                                }
                        }
                }
        }
}

__global__ void updatePaths(Vertex *vertices, int *length, int *updateLength)
{
        int u = threadIdx.x;
        if(length[u] > updateLength[u])
        {
                length[u] = updateLength[u];
                vertices[u].visited = FALSE;
        }

        updateLength[u] = length[u];
}

void printShortestPath(int *array, int src, int dest, int *req_path, int count)
{
        printf("Shortest Path from Vertex %d to %d is %d\nPATH: ", src, dest,
array[dest]);
        for(int i=count-1;i>=0;i--)
   {
      printf("%d-->", req_path[i]);
   }
   printf("%d", dest);
}
```

```
int main(void)
{

        Vertex *vertices;
        Edge *edges;


        int *weights;
        int *path;
        int *len, *updateLength;

        int V, E;

        Vertex *d_V;
        Edge *d_E;
        int *d_W;
        int *d_L;
        int *d_C, *d_P;

        printf("Enter number of vertices: ");
        scanf("%d", &V);

        printf("Enter number of edges: ");
        scanf("%d", &E);

        int sizeV = sizeof(Vertex) * V;
        int sizeE = sizeof(Edge) * E;
        int size = V * sizeof(int);

        float runningTime;
        cudaEvent_t timeStart, timeEnd;

        cudaEventCreate(&timeStart);
        cudaEventCreate(&timeEnd);

        vertices = (Vertex *)malloc(sizeV);
        edges = (Edge *)malloc(sizeE);
        weights = (int *)malloc(E* sizeof(int));
        path = (int *)malloc(V*sizeof(int));
        len = (int *)malloc(size);
        updateLength = (int *)malloc(size);

        Edge ed[E];
        int w[E];
        printf("Enter graph data (SRC_VERTEX DEST_VERTEX WEIGHT):");
```

```
for(int i=0;i<E;i++)
{
        int u, v, wt;
        scanf("%d %d %d", &u, &v, &wt);
        ed[i] = {u, v};
        w[i] = wt;
}
//Edge ed[E] = {{0, 1}, {0, 7}, {1, 7}, {1, 2}, {2, 8}, {2, 3}, {2, 5}, {3, 4}, {3,
5},     {4, 5}, {5, 6}, {6, 7}, {6, 8}, {7, 8}};
//int w[E] = {4, 8, 11, 8, 2, 7, 4, 9, 14, 10, 7, 1, 6, 7};

cudaMalloc((void**)&d_V, sizeV);
cudaMalloc((void**)&d_E, sizeE);
cudaMalloc((void**)&d_W, E * sizeof(int));
cudaMalloc((void**)&d_L, size);
cudaMalloc((void**)&d_C, size);
cudaMalloc((void**)&d_P, size);

int src;
printf("Enter source vertex: ");
scanf("%d", &src);
Vertex root = {src, FALSE};
root.visited = TRUE;
len[root.title] = 0;
updateLength[root.title] = 0;

int dest;
printf("Enter destination vertex: ");
scanf("%d", &dest);

int i = 0;
for(i = 0; i < V; i++)
{
        Vertex a = { i , FALSE};
        vertices[i] = a;
path[i] = root.title;
}

for(i = 0; i < E; i++)
{
        edges[i] = ed[i];
        weights[i] = w[i];
}


cudaMemcpy(d_V, vertices, sizeV, cudaMemcpyHostToDevice);
cudaMemcpy(d_E, edges, sizeE, cudaMemcpyHostToDevice);
cudaMemcpy(d_W, weights, E * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_L, len, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_C, updateLength, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_P, path, size, cudaMemcpyHostToDevice);

initVertices<<<1, V>>>(d_V, d_E, d_W, d_L, d_C, root);

cudaMemcpy(len, d_L, size, cudaMemcpyDeviceToHost);
cudaMemcpy(updateLength, d_C, size, cudaMemcpyDeviceToHost);
cudaEventRecord(timeStart, 0);
cudaMemcpy(d_L, len, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_C, updateLength, size, cudaMemcpyHostToDevice);

for(i = 1; i < V; i++)
{
        findVertex<<<1, V>>>(d_V, d_E, d_W, d_L, d_C, d_P);
        updatePaths<<<1,V>>>(d_V, d_L, d_C);
}

cudaEventRecord(timeEnd, 0);
cudaEventSynchronize(timeEnd);
cudaEventElapsedTime(&runningTime, timeStart, timeEnd);
cudaMemcpy(len, d_L, size, cudaMemcpyDeviceToHost);
cudaMemcpy(path, d_P, size, cudaMemcpyDeviceToHost);

int req_path[V];
int temp = dest;
int count = 0;
while(temp!=root.title)
{
        req_path[count++] = path[temp];
   temp = path[temp];
}

printShortestPath(len, root.title, dest, req_path, count);

printf("Running Time: %f ms\n", runningTime);

free(vertices);
free(edges);
free(weights);
free(len);
free(updateLength);
cudaFree(d_V);
cudaFree(d_E);
```

```
        cudaFree(d_W);
        cudaFree(d_L);
        cudaFree(d_C);
        cudaFree(d_P);
        cudaEventDestroy(timeStart);
        cudaEventDestroy(timeEnd);
}
```

//CUDA INPUT_OUTPUT



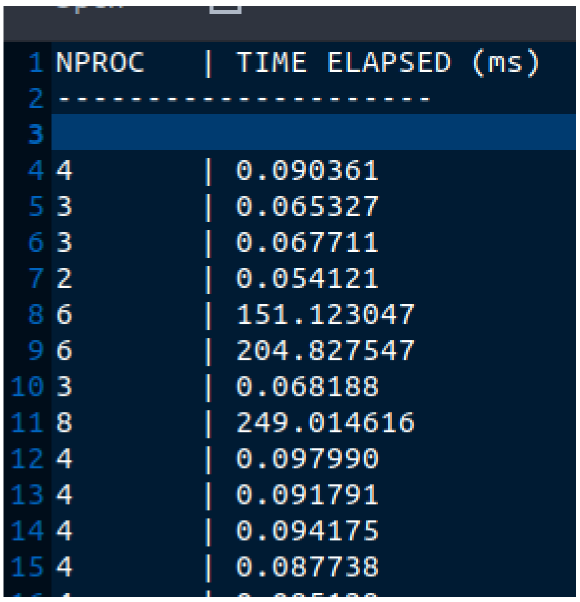
```
'Shortest Path from Vertex 7 to 2 is 7\nPATH: 7-->6-->5-->2Running Time: 0.147648 ms\n'
```

7. LIMITATIONS AND POSSIBLE IMPROVEMENTS

Both the implementations mentioned above perform poorly when compared to their sequential counterparts. In CUDA, we observe a low performance by large number of vertices due to excess memory required. CUDA is not always efficient its performance depends on implementation. In MPI, communication overhead is greater than the parallelism achieved in this case.

8. CONCLUSION

We have succesfully implemented and analysed Parallel Implementation of Dijkstra's Algorithm to find the shortest path among the nodes. As observed, the performance of the MPI implementation was good when the number of processes was under 4. As the number of processes increased more than 4, the time taken increased drastically. We presume that this anomaly is caused due to the hardware limitations. On the other hand, CUDA performed better in the increased number of vertices than MPI since CUDA uses threads rather than processes. However, the serial implementation of the problem is better than either of the parallel implementations.



```
NPROC    | TIME ELAPSED (ms)
---------------------

4        | 0.090361
3        | 0.065327
3        | 0.067711
2        | 0.054121
6        | 151.123047
6        | 204.827547
3        | 0.068188
8        | 249.014616
4        | 0.097990
4        | 0.091791
4        | 0.094175
4        | 0.087738
```

9. REFERENCES

1. Ojekudo, Nathaniel Akpofure, and Nsikan Paul Akpan. “Anapplication of Dijkstra’s Algorithm to Shortest Route Problem.” *Journal of Mathematics*, vol. 13, no. 3, 2017, pp. 20–32., doi:10.9790/5728-1303012032.

2. Sangaiah, Arun Kumar, et al. “An Investigation of Dijkstra and Floyd Algorithms in National City Traffic Advisory Procedures.” *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 2, Feb. 2014, pp. 124–138.

3. X. G. Han et al., "Parallel Dijkstra's Algorithm Based on Multi-Core and MPI", Applied Mechanics and Materials, Vol. 441, pp. 750-753, 2014