National University of Computer and Emerging Sciences

Parallel Distributed Computing

Semester Project Report

Project Statement: Top K Shortest Path Problem with MPI and OpenMP

Project Members:

Saad Ahmed Qureshi 21i-0616

Murtaza Kazmi 21i-0685

Ibrahim Salman 21i-2516

Section: B

Project code for our implementation of the problem:
Serial Execution

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <time.h>
```

Preprocessor directives

```
#define MAX_NODES 265214
#define K 3
```

Costants representing maximum number of nodes n the graph and the number of shortest paths needed to be found

```
typedef struct Edge {
   int vertex;
   int cost;
} Edge;
```

Defines the edge structure in a graph. It contains two values vertex that represents the destination of the vertex of the edge and cost representing weight of the edge.

```
typedef struct AdjListNode {
   Edge edge;
   struct AdjListNode* next;
} AdjListNode;
```

This defines a structure adjlistnode representing a node in the adjacency list and it contains an edge and a pointer to the next node in the list.

```
typedef struct {
   AdjListNode* head;
} AdjList;
```

This represents the adjacency list. It contains a pointer head pointing to the first node in the list

```
typedef struct Graph {
   int V;
   AdjList* array;
} Graph;
```

This defines the structure of a graph. It contains vertices and an array of adjacency lists

```
typedef struct HeapNode {
   int vertex;
   int dist;
} HeapNode;
```

This defines a heapnode representing a node in the min heap that contains the vertex index and distance.

```
typedef struct {
   HeapNode* nodes;
   int size;
   int capacity;
} MinHeap;
```

This is the min heap structure which contains array of Heapnodes and size indicating number of elements in the heap and capacity indicating max capacity of heap.

```
Graph* createGraph(int V) {
   Graph* graph = (Graph*) malloc(sizeof(Graph));
   graph->V = V;
   graph->array = (AdjList*) malloc(V * sizeof(AdjList));
   for (int i = 0; i < V; i++) {
      graph->array[i].head = NULL;
   }
   return graph;
}
```

Graph Creation

```
void addEdge(Graph* graph, int src, int dest, int cost) {
   AdjListNode* newNode = (AdjListNode*) malloc(sizeof(AdjListNode));
   newNode->edge.vertex = dest;
   newNode->edge.cost = cost;
   newNode->next = graph->array[src].head;
   graph->array[src].head = newNode;
}
```

Adding Edges

```
void freeGraph(Graph* graph) {
   for (int i = 0; i < graph->V; i++) {
      AdjListNode* node = graph->array[i].head;
      while (node) {
         AdjListNode* temp = node;
         node = node->next;
         free(temp);
      }
   }
   free(graph->array);
   free(graph);
}
```

Freeing Memory

```
MinHeap* createMinHeap(int capacity) {
   MinHeap* minHeap = (MinHeap*)
malloc(sizeof(MinHeap));
   minHeap->nodes = (HeapNode*)
malloc(capacity * sizeof(HeapNode));
```

Creating a min-heap

```
    minHeap->size = 0;
    minHeap->capacity = capacity;
    return minHeap;
}

void insertMinHeap(MinHeap* minHeap, int v, int dist) {
    if (minHeap->size == minHeap->capacity) {
        return;
    }
    int i = minHeap->size++;
    minHeap->nodes[i].vertex = v;
    minHeap->nodes[i].dist = dist;
    while (i && minHeap->nodes[(i - 1) / 2].dist
 > minHeap->nodes[i].dist) {
        HeapNode tmp = minHeap->nodes[i];
        minHeap->nodes[i] = minHeap->nodes[(i - 1) / 2];
        minHeap->nodes[(i - 1) / 2] = tmp;
        i = (i - 1) / 2;
    }
}
```

Inserting values into min heap

```
HeapNode extractMin(MinHeap* minHeap) {
    if (minHeap->size <= 0) {
        return (HeapNode){-1, INT_MAX};
    }
    HeapNode root = minHeap->nodes[0];
    minHeap->nodes[0] = minHeap->
nodes[--minHeap->size];
    int i = 0;
    while ((2 * i + 1) < minHeap->size) {
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        int smallest = left;
        if (right < minHeap->size && minHeap->
nodes[right].dist < minHeap->nodes[left].dist) {
            smallest = right;
        }
        if (minHeap->nodes[i].dist <= minHeap->
nodes[smallest].dist) break;
        HeapNode tmp = minHeap->nodes[i];
        minHeap->nodes[i] = minHeap->nodes[smallest];
        minHeap->nodes[smallest] = tmp;
        i = smallest;
    }
    return root;
}
void readGraphFromFile(Graph* graph, const char* filename) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Could not open file: %s\n", filename);
        return;
    }
    char line[256];
    while (fgets(line, sizeof(line), file)) {
        if (line[0] == '#') continue;
        int src, dest;
        if (sscanf(line, "%d\t%d", &src, &dest) == 2) {
            addEdge(graph, src, dest, 1);
        }
    }
    fclose(file);}
```

Extracting the minimum element from a min heap.

Reading a graph from a file

```
char* findKShortestPaths(Graph* graph, int src, int dest) {
    int V = graph->V;
    int dis[V][K];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < K; j++) {
            dis[i][j] = INT_MAX;
        }
    }

    MinHeap* minHeap = createMinHeap(V * K);
    insertMinHeap(minHeap, src, 0);
    dis[src][0] = 0;

    while (minHeap->size != 0) {
        HeapNode heapNode = extractMin(minHeap);
        int u = heapNode.vertex;

        for (AdjListNode* crawl = graph->array[u].head;
crawl != NULL; crawl = crawl->next) {
            int v = crawl->edge.vertex;
            int weight = crawl->edge.cost;
            if (dis[v][K-1] > dis[u][0] + weight) {
                dis[v][K-1] = dis[u][0] + weight;
                for (int i = K-1; i > 0 && dis[v][i] < dis[v][i-1]; i--) {
                    int temp = dis[v][i];
                    dis[v][i] = dis[v][i-1];
                    dis[v][i-1] = temp;
                }
                insertMinHeap(minHeap, v, dis[v][K-1]);
            }
        }
    }

    char* paths = (char*)malloc(256 * K * sizeof(char));
    char temp[256];
    paths[0] = '\0';
    printf("Source: %d, Destination: %d\n", src, dest);
    for (int i = 0; i < K; i++) {
        if (dis[dest][i] == INT_MAX) {
```

Finding K shortest paths

```c
            snprintf(temp, sizeof(temp), "Path %d: Infinity\n", i + 1);
        } else {
            snprintf(temp, sizeof(temp), "Path %d: %d\n", i + 1, dis[dest][i]);
        }
        strcat(paths, temp);
    }

    free(minHeap->nodes);
    free(minHeap);

    return paths;
}


int getMaxNodeID(Graph* graph) {
    int max = 0;
    for (int i = 0; i < graph->V; i++) {
        AdjListNode* node = graph->array[i].head;
        while (node) {
            if (node->edge.vertex > max) {
                max = node->edge.vertex;
            }
            node = node->next;
        }
    }
    return max;
}
```

Getting maximum node id in the graph

```c
void generateRandomPairs(int* sources, int*
destinations, int num_pairs, int max_node) {
    srand(time(NULL));
    for (int i = 0; i < num_pairs; i++) {
        sources[i] = rand() % max_node;
        destinations[i] = rand() % max_node;
    }
}
```

Generating random pairs of source and destination nodes

```c
int main() {
   clock_t start_time, end_time;
   double cpu_time_used;

   start_time = clock();

   // Load graph from file
   const char* filename = "Email-EuAll.txt";
   Graph* graph = createGraph(MAX_NODES);
   readGraphFromFile(graph, filename);

   int max_node = getMaxNodeID(graph);

   const int num_pairs = 10;
   int sources[num_pairs];
   int destinations[num_pairs];

   generateRandomPairs(sources, destinations, num_pairs, max_node);

   // 10 pairs
   for (int i = 0; i < num_pairs; i++) {
      char* paths = findKShortestPaths(graph, sources[i], destinations[i]);
      printf("%s", paths);
      free(paths);
   }

   freeGraph(graph);

   end_time = clock();
   cpu_time_used = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

   printf("Execution time: %f seconds\n", cpu_time_used);

   return 0;
}
```

Main function for testing

Project code for our implementation of the kth Shortest Path problem.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <mpi.h>
#include <omp.h>
#include <time.h>
```

Preprocessor directives

```
#define MAX_NODES 265214
#define K 3
```

Costants representing maximum number of nodes n the graph and the number of shortest paths needed to be found

```
typedef struct Edge {
    int vertex;
    int cost;
} Edge;
```

Defines the edge structure in a graph. It contains two values vertex that represents the destination of the vertex of the edge and cost representing weight of the edge.

```
typedef struct AdjListNode {
    Edge edge;
    struct AdjListNode* next;
} AdjListNode;
```

This defines a structure adjlistnode representing a node in the adjacency list and it contains an edge and a pointer to the next node in the list.

```
typedef struct {
    AdjListNode* head;
} AdjList;
```

This represents the adjacency list. It contains a pointer head pointing to the first node in the list

```
typedef struct Graph {
    int V;
    AdjList* array;
} Graph;
```

This defines the structure of a graph. It contains vertices and an array of adjacency lists

```
typedef struct HeapNode {
    int vertex;
    int dist;
} HeapNode;
```

This defines a heapnode representing a node in the min heap that contains the vertex index and distance.

```
typedef struct {
    HeapNode* nodes;
    int size;
    int capacity;
} MinHeap;
```

This is the min heap structure which contains array of Heapnodes and size indicating number of elements in the heap and capacity indicating max capacity of heap.

```c
Graph* createGraph(int V) {
   Graph* graph = (Graph*) malloc(sizeof(Graph));
   graph->V = V;
   graph->array = (AdjList*) malloc(V * sizeof(AdjList));
   for (int i = 0; i < V; i++) {
      graph->array[i].head = NULL;
   }
   return graph;
}
```

Graph Creation

```c
void addEdge(Graph* graph, int src, int dest, int cost) {
   AdjListNode* newNode = (AdjListNode*)
malloc(sizeof(AdjListNode));
   newNode->edge.vertex = dest;
   newNode->edge.cost = cost;
   newNode->next = graph->array[src].head;
   graph->array[src].head = newNode;
}
```

Adding Edges

```c
void freeGraph(Graph* graph) {
   #pragma omp parallel for
   for (int i = 0; i < graph->V; i++) {
      AdjListNode* node = graph->array[i].head;
      while (node) {
         AdjListNode* temp = node;
         node = node->next;
         free(temp);
      }
   }
   free(graph->array);
   free(graph);
}
```

Freeing Memory. Parallize for loop using openMP

```c
MinHeap* createMinHeap(int capacity) {
   MinHeap* minHeap = (MinHeap*)
malloc(sizeof(MinHeap));
   minHeap->nodes = (HeapNode*)
malloc(capacity * sizeof(HeapNode));
```

Creating a min-heap

```
    minHeap->size = 0;
    minHeap->capacity = capacity;
    return minHeap;
}


void insertMinHeap(MinHeap* minHeap, int v, int dist) {
    if (minHeap->size == minHeap->capacity) {
        return;
    }
    int i = minHeap->size++;
    minHeap->nodes[i].vertex = v;
    minHeap->nodes[i].dist = dist;
    while (i && minHeap->nodes[(i - 1) / 2].dist >
minHeap->nodes[i].dist) {
        HeapNode tmp = minHeap->nodes[i];
        minHeap->nodes[i] = minHeap->nodes[(i - 1) / 2];
        minHeap->nodes[(i - 1) / 2] = tmp;
        i = (i - 1) / 2;
    }
}
```

Inserting values into min heap

```
HeapNode extractMin(MinHeap* minHeap) {
   if (minHeap->size <= 0) {
      return (HeapNode){-1, INT_MAX};
   }
   HeapNode root = minHeap->nodes[0];
   minHeap->nodes[0] = minHeap-
>nodes[--minHeap->size];
   int i = 0;
   while ((2 * i + 1) < minHeap->size) {
      int left = 2 * i + 1;
      int right = 2 * i + 2;
      int smallest = left;
      if (right < minHeap->size && minheap
->nodes[right].dist < minHeap->nodes[left].dist) {
         smallest = right;
      }
      if (minHeap->nodes[i].dist <=
minHeap->nodes[smallest].dist) break;
      HeapNode tmp = minHeap->nodes[i];
      minHeap->nodes[i] =
minHeap->nodes[smallest];
      minHeap->nodes[smallest] = tmp;
      i = smallest;
   }
   return root;
}
```

Extracting the minimum element from a min heap.

```
void readGraphFromFile(Graph* graph, const char* filename) {
   FILE* file = fopen(filename, "r");
   if (!file) {
      fprintf(stderr, "Could not open file: %s\n", filename);
      return;
   }
   char line[256];
   while (fgets(line, sizeof(line), file)) {
      if (line[0] == '#') continue;
      int src, dest;
      if (sscanf(line, "%d\t%d", &src, &dest) == 2) {
         addEdge(graph, src, dest, 1);
      }
   }fclose(file);}
```

Reading a graph from a file

```
char* findKShortestPaths(Graph* graph, int src, int dest, int rank) {
   int V = graph->V;
   int dis[V][K];
   #pragma omp parallel for
   for (int i = 0; i < V; i++) {
      #pragma omp parallel for
      for (int j = 0; j < K; j++) {
         dis[i][j] = INT_MAX;
      }
   }
   MinHeap* minHeap = createMinHeap(V * K);
   insertMinHeap(minHeap, src, 0);
   dis[src][0] = 0;
   while (minHeap->size != 0) {
      HeapNode heapNode = extractMin(minHeap);
      int u = heapNode.vertex;
      for (AdjListNode* crawl = graph->
array[u].head; crawl != NULL; crawl = crawl->next) {
         int v = crawl->edge.vertex;
         int weight = crawl->edge.cost;
         if (dis[v][K-1] > dis[u][0] + weight) {
            dis[v][K-1] = dis[u][0] + weight;
            for (int i = K-1; i > 0 && dis[v][i] < dis[v][i-1]; i--) {
               int temp = dis[v][i];
               dis[v][i] = dis[v][i-1];
               dis[v][i-1] = temp;
            }
            insertMinHeap(minHeap, v, dis[v][K-1]);
         }
      }
   }

   char* paths = (char*)malloc(256 * K * sizeof(char));
   char temp[256];
   paths[0] = '\0'; // Initialize as empty string
   printf("Pair %d -: Source: %d, Destination: %d\n", rank, src, dest);
   for (int i = 0; i < K; i++) {

      if (dis[dest][i] == INT_MAX) {
         snprintf(temp, sizeof(temp), "Path %d: Infinity\n", i + 1);
```

Finding K shortest paths. Parallelize the for loops using openMP directives

```
        } else {
            snprintf(temp, sizeof(temp), "Path %d: %d\n", i + 1, dis[dest][i]);
        }
        strcat(paths, temp);
    }

    free(minHeap->nodes);
    free(minHeap);

    return paths;
}

int getMaxNodeID(Graph* graph) {
    int max = 0;
    for (int i = 0; i < graph->V; i++) {
        AdjListNode* node = graph->array[i].head;
        while (node) {
            if (node->edge.vertex > max) {
                max = node->edge.vertex;
            }
            node = node->next;
        }
    }
    return max;
}
```

Getting maximum node id in the graph

```
void generateRandomPairs(int* sources,
 int* destinations, int num_pairs, int max_node) {
    srand(time(NULL));
    for (int i = 0; i < num_pairs; i++) {
        sources[i] = rand() % max_node;
        destinations[i] = rand() % max_node;
    }
}
```

Generating random pairs of source and destination nodes

```c
int main(int argc, char* argv[]) {

    clock_t start_time, end_time;
    double cpu_time_used;

    start_time = clock();

    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Load graph from file
    const char* filename = "graph1.txt";
    Graph* graph = createGraph(MAX_NODES);
    readGraphFromFile(graph, filename);

    // Get the maximum possible node ID
    int max_node = getMaxNodeID(graph);

    // Generate random source and destination nodes
    const int num_pairs = 10;
    int sources[num_pairs];
    int destinations[num_pairs];

    generateRandomPairs(sources, destinations, num_pairs, max_node);

    // Distribute workload among processes
    int pairs_per_process = num_pairs / size;
    int remainder_pairs = num_pairs % size;

    int start_index = rank * pairs_per_process;
    int end_index = start_index + pairs_per_process;

    if (rank == size - 1) {
        end_index += remainder_pairs;
    }
```

Main function for testing. For parallelization MPI commands and functions were used.

```
    // Process pairs assigned to this process
    for (int i = start_index; i < end_index; i++) {

        char* paths = findKShortestPaths(graph, sources[i], destinations[i],rank);
        printf("%s", paths); // Print paths for the current pair
        free(paths); // Free memory allocated for paths
    }

    MPI_Barrier(MPI_COMM_WORLD); // Synchronize all processes
    freeGraph(graph);
    MPI_Finalize();
    end_time = clock();
    cpu_time_used = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Execution time: %f seconds\n", cpu_time_used);

    return 0;
}
```

Challenges Faced During Preprocessing:

1. The doctorwho.csv file had to be converted to a txt file with all the names of the characters converted into unique integer values.
2. The txt files had similar format but there were some differences which had to be taken into account to make sure that the code created similar structured graph for all 3 files.
3. Since some of the files did not have weights assigned to their edges, when creating the edges the weights were set to the value of 1.

Challenges Faced During implementation and testing:

1. It was easy to find the kth shortest paths using a small sample size however the provided txt files contained hundreds of thousands of nodes so initially there were some issues regarding segmentation however those solved by modifying our code to cater to larger data sizes by making the max_nodes macro equal the the largest node count from the 3 files provided. This was 265214 for the EU email file.

2. In the testing phase we had the issue that although it was easy to confirm the results for a smaller graph by dry running the solution by hand, given the large data size of the provided files it was harder to confirm the results for them as the dataset was much larger. The only solution we had was to see a known path and test for only it to confirm that our solution was correct.

Optimizations used to improve performance:

1. MPI processes were used to handle the subset of paths. If 10 pairs of sources and destinations are to be used then 10 processess each can be used for each pair of nodes. Parallelization allowed for quicker execution time.
2. MPI Barrier was used for synchronization.
3. Using OpenMP the for loops were mostly all parallelized.

Results for Doctor Who file:

Serial Execution:

1st Execution time	2nd Execution time	3rd Execution time	Average Execution time
1.8 seconds	2.4 seconds	2.23 seconds	2.14 seconds

Parallel Execution:

1st Execution time	2nd Execution time	3rd Execution time	Average Execution time
0.04 seconds	0.04 seconds	0.04 seconds	0.04 seconds

Results for Enron email file:

Serial Execution:

1st Execution time	2nd Execution time	3rd Execution time	Average Execution time
0.6 seconds	0.6 seconds	0.8 seconds	0.66 seconds

Parallel Execution:

1st Execution time	2nd Execution time	3rd Execution time	Average Execution time
2.5 seconds	2.6 seconds	2.4 seconds	2.5 seconds

Results for EU email file:

Serial Execution:

1st Execution time	2nd Execution time	3rd Execution time	Average Execution time
0.78 seconds	0.82 seconds	0.94 seconds	0.84 seconds

Parallel Execution:

1st Execution time	2nd Execution time	3rd Execution time	Average Execution time
2.8 seconds	3.1 seconds	2.9 seconds	2.9 seconds

Experimental Setup:

We tested the code and ran the code in Kali Linux virtual machine environment using VMWare software.

Overall Analysis:

For our datasets the serial code performs better in all instances compared to the parallel code. This is because the overheads increase and process creation takes more time.

Our insights:

There is not much difference in the execution times in the serial and parallel versions when the dataset size gets larger. However for smaller dataset sizes the serial code executes many times faster than parallel version.