Parallelizing Dijkstra algorithm

Project Report in Parallel Systems

Francesco Moschella
fma20001@student.mdh.se

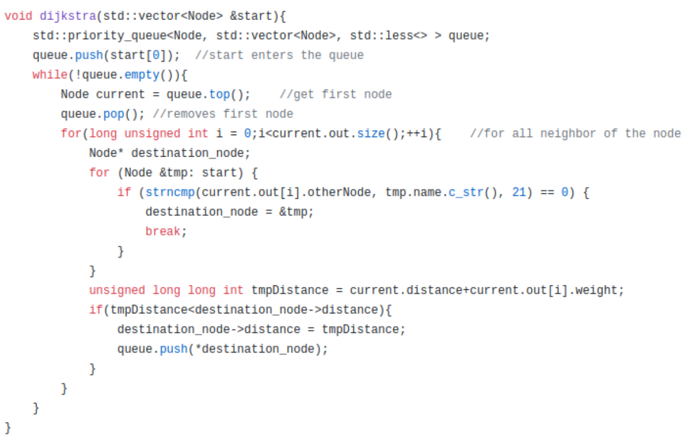
Rea Ballkoci
rbi19003@student.mdh.se

1. Introduction

Given a graph, let $G = (V, E)$ be a directed graph, where $|V| = n, |E| = m$, let s be a distinguished vertex of the graph and w be the non-negative value to the weight of each edge, which represents the distance between the two vertexes.

In graph theory, the purpose of Dijkstra's algorithm is to find the shortest path between the given source vertex s and all other vertices in the graph. You can apply this algorithm in different fields, for example: road networks or everything that is convertible in a graph.

Based on the algorithm, we build a routing table (Table 1) for node s, as described in the example below (Figure 1), where the highlighted cells show the minimum weighted distance. The sequential version of Dijkstra is shown in Figure 2.



```
void dijkstra(std::vector<Node> &start){
    std::priority_queue<Node, std::vector<Node>, std::less<> > queue;
    queue.push(start[0]);  //start enters the queue
    while(!queue.empty()){
        Node current = queue.top();    //get first node
        queue.pop(); //removes first node
        for(long unsigned int i = 0;i<current.out.size();++i){    //for all neighbor of the node
            Node* destination_node;
            for (Node &tmp: start) {
                if (strncmp(current.out[i].otherNode, tmp.name.c_str(), 21) == 0) {
                    destination_node = &tmp;
                    break;
                }
            }
            unsigned long long int tmpDistance = current.distance+current.out[i].weight;
            if(tmpDistance<destination_node->distance){
                destination_node->distance = tmpDistance;
                queue.push(*destination_node);
            }
        }
    }
}
```

Figure 2: Dijkstra algorithm

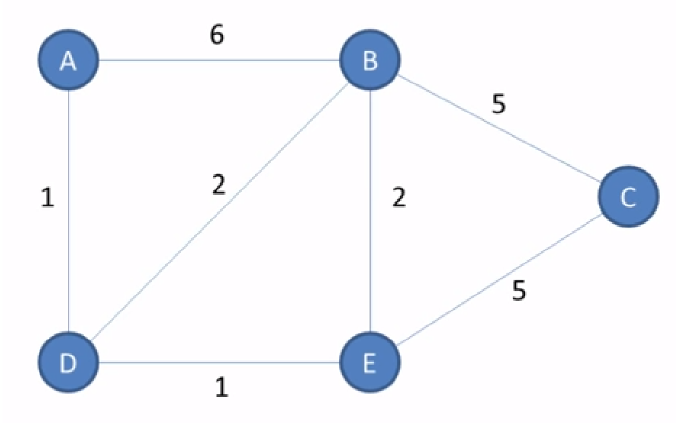
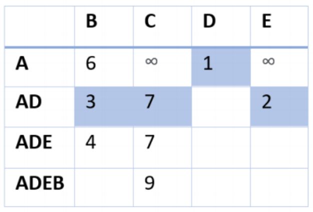


Figure 1: Graph example



	B	C	D	E
A	6	∞	1	∞
AD	3	7		2
ADE	4	7		
ADEB		9		

Table 1: Routing table example

2. Parallel Approach

The sequential implementation of the Dijkstra algorithm has some disadvantages. For instance, if the graph increases in size, the sequential program may require much more time, which is a problem for time-sensitive applications.

As a solution to address this problem, we propose a parallelization of the algorithm, using a *farm/worker* architecture with the help of Boost MPI and *multi-threading* with OpenMP. Workers and Emitter use the message passing paradigm to communicate and broadcast node-related information. The emitter cycle is done in parallel using multiprocessing.

Each processor $p[0, k]$ will receive from the Emitter, one by one, a subgroup of edges of size $\delta = |E|/k = m/k$, some integer values needed to calculate the distance, and calculates the distance from the starting node. If the calculated distance is lower than the one stored in the collector, the processor p will send the new result to the collector. This process will continue until every vertex is scanned and every edge is checked.

A. Theoretical performance

The theoretical performance of the emitter + performance of worker as calculated by us would be

O (1+(num_edges*num_nodes)+2*recv_time+(5*send_time) +irecv_time + num_nodes) + O (1)

where in the Emitter *irecv_time* is 1000ms in worst case scenario and *3*recv_time* in other cases, and in the Worker, time complexity is O (1) because it only performs algebraic operations and if statements. If we consider the communication time to be 0 (send_time=0, recv_time=0) then it would change to:

O(1+(num_edges*num_nodes)+1000+num_nodes) + O (1)

All constants can be removed which leads us to:

O(num_edges*num_nodes+num_edges)
As we already said in section 1 num_edges = —E— = m and num_nodes = —V— = n we would get *O (m*n+n)* and since $m * n >> n$ then we can say that our complexity is *O(m*n)*.

Comparing with the sequential algorithms that has a time complexity of *O(n*n)* where n=$|V|$ the number of nodes or vertices in the graph. We can see that the parallel version would perform faster if $m < n$ which is impossible considering that every vertex should have at least one edge.

3. Implementation

This implementation is not optimal as the parallel algorithm does not always find the best path. We know that the sequential version always finds the optimal way. We tested the output of the two codes, sequential and parallel, looking if they are equal, and, as shown by the keyword "False", the two results are not the same, meaning that the parallel does not always find the optimal route. We can say that this happens due to parallelization and read/write concurrency. As an example, we can take two different vertices, that have an edge that goes to the same vertex, both may do the check at the same moment, and in both cases, it may evaluate to true. Since workers are not atomic, the worker who assigned the distance as last will have its value stored in the final graph.

The approach explained in Section 2 is implemented as depicted in Figure 3 and the results are shown in Figure 4.

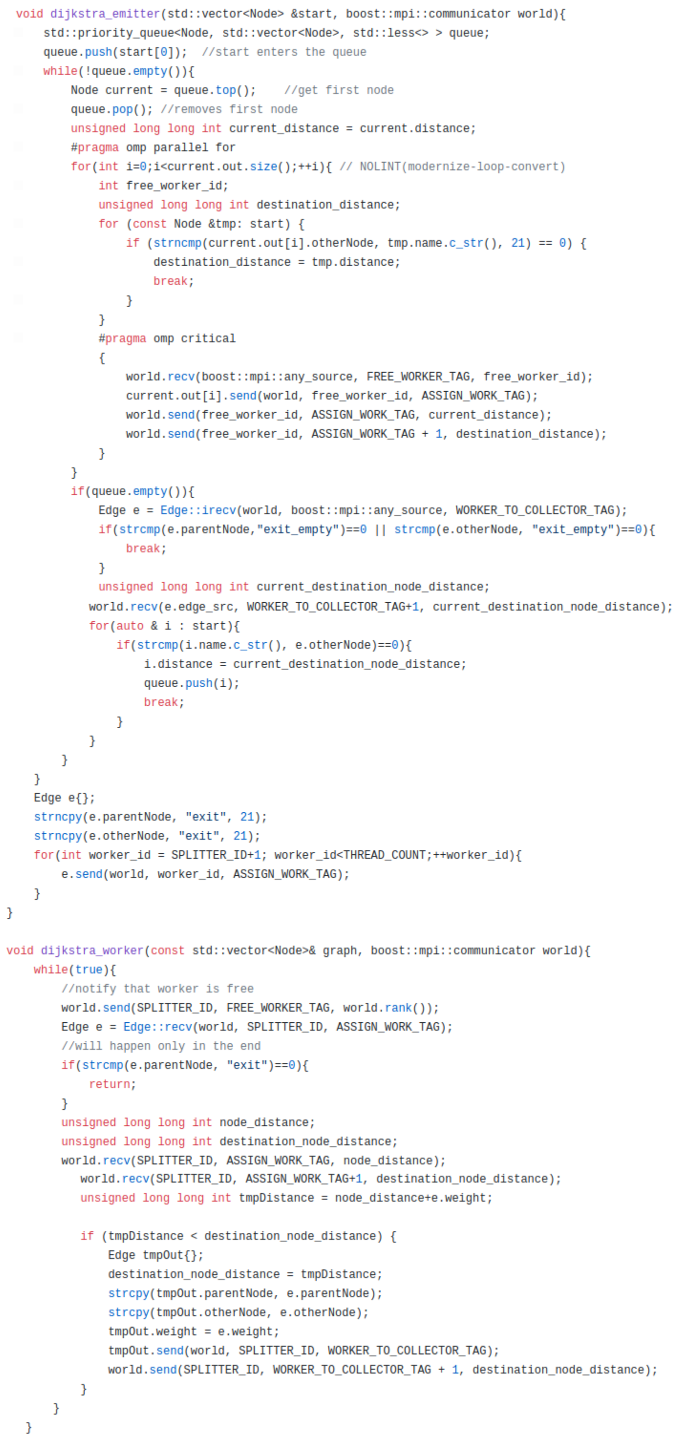
4. Experiments

We conducted some tests on four processors, with two threads per processor, for the sequential and parallel implementation. We tested the code using graphs of different sizes *V*.

The sequential version is faster than the parallel one. For example, it takes only 1 ms to evaluate a graph with 60 nodes. The parallel version is relatively slow. Because of that, we stopped the evaluation at 150 nodes. We believe that the results are still quite descriptive.

It might look like the parallel version does not follow any rising function, as the time needed is lower for 90 nodes rather than 80 nodes. That is due to the uncontrollable randomization factor used during the graph creation that prevents a well-connected graph.

Table 2 represents the results.

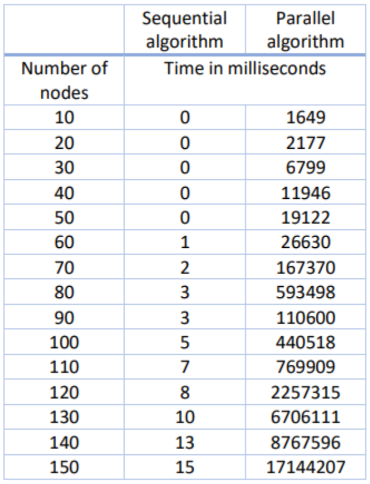


```
void dijkstra_emitter(std::vector<Node> &start, boost::mpi::communicator world){
    std::priority_queue<Node, std::vector<Node>, std::less<> > queue;
    queue.push(start[0]);  //start enters the queue
    while(!queue.empty()){
        Node current = queue.top();    //get first node
        queue.pop(); //removes first node
        unsigned long long int current_distance = current.distance;
        #pragma omp parallel for
        for(int i=0;i<current.out.size();++i){ // NOLINT(modernize-loop-convert)
            int free_worker_id;
            unsigned long long int destination_distance;
            for (const Node &tmp: start) {
                if (strncmp(current.out[i].otherNode, tmp.name.c_str(), 21) == 0) {
                    destination_distance = tmp.distance;
                    break;
                }
            }
            #pragma omp critical
            {
                world.recv(boost::mpi::any_source, FREE_WORKER_TAG, free_worker_id);
                current.out[i].send(world, free_worker_id, ASSIGN_WORK_TAG);
                world.send(free_worker_id, ASSIGN_WORK_TAG, current_distance);
                world.send(free_worker_id, ASSIGN_WORK_TAG + 1, destination_distance);
            }
        }
        if(queue.empty()){
            Edge e = Edge::irecv(world, boost::mpi::any_source, WORKER_TO_COLLECTOR_TAG);
            if(strcmp(e.parentNode,"exit_empty")==0 || strcmp(e.otherNode, "exit_empty")==0){
                break;
            }
            unsigned long long int current_destination_node_distance;
           world.recv(e.edge_src, WORKER_TO_COLLECTOR_TAG+1, current_destination_node_distance);
           for(auto & i : start){
               if(strcmp(i.name.c_str(), e.otherNode)==0){
                   i.distance = current_destination_node_distance;
                   queue.push(i);
                   break;
               }
           }
        }
    }
    Edge e{};
    strncpy(e.parentNode, "exit", 21);
    strncpy(e.otherNode, "exit", 21);
    for(int worker_id = SPLITTER_ID+1; worker_id<THREAD_COUNT;++worker_id){
        e.send(world, worker_id, ASSIGN_WORK_TAG);
    }
}

void dijkstra_worker(const std::vector<Node>& graph, boost::mpi::communicator world){
    while(true){
        //notify that worker is free
        world.send(SPLITTER_ID, FREE_WORKER_TAG, world.rank());
        Edge e = Edge::recv(world, SPLITTER_ID, ASSIGN_WORK_TAG);
        //will happen only in the end
        if(strcmp(e.parentNode, "exit")==0){
            return;
        }
        unsigned long long int node_distance;
        unsigned long long int destination_node_distance;
        world.recv(SPLITTER_ID, ASSIGN_WORK_TAG, node_distance);
           world.recv(SPLITTER_ID, ASSIGN_WORK_TAG+1, destination_node_distance);
           unsigned long long int tmpDistance = node_distance+e.weight;

           if (tmpDistance < destination_node_distance) {
               Edge tmpOut{};
               destination_node_distance = tmpDistance;
               strcpy(tmpOut.parentNode, e.parentNode);
               strcpy(tmpOut.otherNode, e.otherNode);
               tmpOut.weight = e.weight;
               tmpOut.send(world, SPLITTER_ID, WORKER_TO_COLLECTOR_TAG);
               world.send(SPLITTER_ID, WORKER_TO_COLLECTOR_TAG + 1, destination_node_distance);
           }
       }
   }
```

Figure 3: Parallelization of Dijkstra algorithm

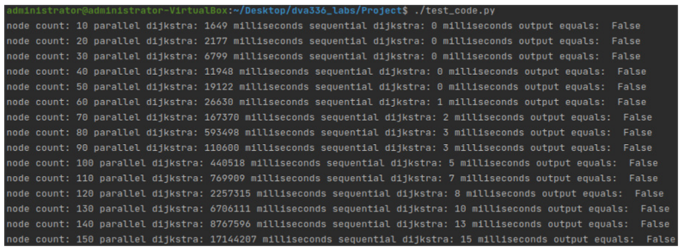
5. References

1) Github repository of the project, [Online]. Available: https://github.com/HarlockOfficial/dva336_labs/tree/main/Project, Accessed 2021-01-21
2) Djikstra algorithm page on Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, Accessed 2021-01-21
3) Boost Mpi library documentation, [Online]. Available: https://www.boost.org/doc/libs/1_75_0/doc/html/mpi.html, Accessed 2021-01-21
4) OpenMP API, [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, Accessed 2021-01-21



	Sequential algorithm	Parallel algorithm
Number of nodes	Time in milliseconds	
10	0	1649
20	0	2177
30	0	6799
40	0	11946
50	0	19122
60	1	26630
70	2	167370
80	3	593498
90	3	110600
100	5	440518
110	7	769909
120	8	2257315
130	10	6706111
140	13	8767596
150	15	17144207

Table 2: Results of the sequential and parallel algorithm



```
administrator@administrator-VirtualBox:~/Desktop/dva336_labs/Project$ ./test_code.py
node count: 10 parallel dijkstra: 1649 milliseconds sequential dijkstra: 0 milliseconds output equals:  False
node count: 20 parallel dijkstra: 2177 milliseconds sequential dijkstra: 0 milliseconds output equals:  False
node count: 30 parallel dijkstra: 6799 milliseconds sequential dijkstra: 0 milliseconds output equals:  False
node count: 40 parallel dijkstra: 11948 milliseconds sequential dijkstra: 0 milliseconds output equals:  False
node count: 50 parallel dijkstra: 19122 milliseconds sequential dijkstra: 0 milliseconds output equals:  False
node count: 60 parallel dijkstra: 26630 milliseconds sequential dijkstra: 1 milliseconds output equals:  False
node count: 70 parallel dijkstra: 167370 milliseconds sequential dijkstra: 2 milliseconds output equals:  False
node count: 80 parallel dijkstra: 593498 milliseconds sequential dijkstra: 3 milliseconds output equals:  False
node count: 90 parallel dijkstra: 110600 milliseconds sequential dijkstra: 3 milliseconds output equals:  False
node count: 100 parallel dijkstra: 440518 milliseconds sequential dijkstra: 5 milliseconds output equals:  False
node count: 110 parallel dijkstra: 769909 milliseconds sequential dijkstra: 7 milliseconds output equals:  False
node count: 120 parallel dijkstra: 2257315 milliseconds sequential dijkstra: 8 milliseconds output equals:  False
node count: 130 parallel dijkstra: 6706111 milliseconds sequential dijkstra: 10 milliseconds output equals:  False
node count: 140 parallel dijkstra: 8767596 milliseconds sequential dijkstra: 13 milliseconds output equals:  False
node count: 150 parallel dijkstra: 17144207 milliseconds sequential dijkstra: 15 milliseconds output equals:  False
```

Figure 4: Results of the experiments in Command Prompt