Introduction to High Performance Computing and Optimization
Winter 2019

Dr. Uwe Prüfert, Friederike Röver
Technische Universität Bergakademie Freiberg
Institute of numerical Analysis and Optimization

TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
The University of Resources. Since 1765.

HPC Project: Find an optimal way on a graph.

Introduction or "How road navigation works" Anyone who has had the luxury of looking up directions on how to get somewhere on a GPS based navigation tool has likely ignored or disregarded the intricacies of how such a path is calculated.

The entire premise of GPS based navigation tools is using a giant graph with nodes (modeling crossings of roads or bus/train stations) and weighted edges (modeling roads/railway lines with time that is typically necessary to come from node to node by car/bus/train or simply the distance between the nodes connected by the edge) to figure out the fastest or shortest way to connect start (node) and end (node).

The short (actually only two and a quater pages) paper by E. W. Dijkstra written in 1959 out-lines the basic methodology that navigation tools use to preform this calculation.

Conclusion: The data structure of all GPS based navigation tools is a giant graph with lots of nodes and edges. To find the best way from node A to node B, well-known standard tools from discrete optimization can be used. That's all.

However, given the large amounts of data that would be necessary to analyze a large graph and keep track of all its nodes and edges, it is impressive that such a calculation can be performed by such tools in such a short time. Note that offline tools must perform this calculations (just-in-time) on your cellphone!

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
9      dist[source] ← 0
10
11     while Q is not empty:
12         u ← vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u:           // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] ← alt
20                 prev[v] ← u
21
22     return dist[], prev[]
23
```

Pseudocode for Djikstra's Algorithm (Source https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

For a first reading we suggest en.wikipedia.org.

Task Write a parallel C or C++ program to find the shortest path in a randomly weighted graph using **MPI**.

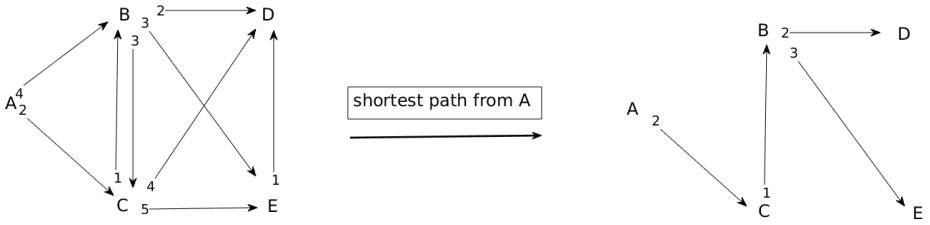
Document your program by using comments.

Perform **weak** and **strong parallel scalability** tests on the TUBAF Cluster.

Document your results.

Hints

(i) For debugging it is advisable to create a small graph with a known solution, e.g.:



(ii) Using the the `rand()` function to create the random graph use integers.

(iii) For the scalbility tests, choose the size of the graph large enough.

(iv) What do you expect for the weak and strong parallel scalability? What are you observations concerning the parallel scalability?

Rules

(i) The deadline is 21.02.2020 at 24.00.

(ii) The programming language should be C or C#.

(iii) The Project can be handed in by groups of max. 2 students.

(iv) Send your program together with the documentation of the performance tests and compile instructions to Friederike.Roever@math.tu-freiberg.de.

(v) Include the solution of the example graph using 2 processes.

(vi) To obtain full points (15) your program must scale in the expected way.