MPI-BASED STUDY OF DIJKSTRA'S AND BELLMAN-FORD'S ALGORITHMS FOR THE SHORTEST PATH PROBLEM

Hamilton Carrillo-Nuñez and Paulina Aguirre

Lecture: Design of parallel and high computer performance
Department of Computer Science, ETH, Zürich

ABSTRACT

The aim of this project is to develop and implement a parallelization scheme, based on the Message-Passing-Interface (MPI), of two different algorithms to find the shortest path, given a source node, to all other nodes in a graph. The well-known Dijkstra's [1] and Bellman-Ford's [2, 3] algorithms are considered. Our findings suggest that the former requires more communication, leading the latter to a better scaling when the number of processors increases.

1. INTRODUCTION

Given an initial point A and a target point B, what is the shortest path (SP) between them? Human-beings will try an intuitive and heuristic approach in order to answer this questions. However, the SP problem is very general and can take many forms. For instance, in the case of a route map the SP can be represented as the shortest distance or the faster route. On daily basis, SP algorithms are widely used in many route map applications such as Google or Apple maps, or any software helping to choose a route uses some form of a SP algorithm.

In this project, the classical Dijkstra's [1] and the Bellman-Ford's [2, 3] algorithms are considered. Dijkstra's algorithm iteratively extracts one node from a min-priority queue and performs relaxations to this node's neighbors. The algorithm requires many iterations, being more robust to any parallelization. The latter is not the case of Bellman-Ford which involves fewer iterations, making it highly parallelizable. Many attempts to develop parallel versions of Dijkstra's [4, 5] and Bellman-Ford's [6, 7] algorithms can be found on the literature. In spite of their successful implementation, there is room for improvement in terms of speed-up and scalability [6, 7].

In order to understand the limitation in parallelization of the aforementioned algorithms, this paper is organized as follows. In section 2, the SP algorithms are introduced. In section 3, the graph generation is described, as well as its transformation to an adjacency matrix notation which is more convenient for the parallelization strategy adopted from Ref. [8]. The latter has been implemented using the MPI library and details are given in section 4. Benchmarking and discussion of our simulations are presented in section 5. Finally, conclusions are drawn in section 6.

Algorithm 1: Dijkstra's sequential

```
input : Graph, source
output: dist [ ], prev [ ]
create vertex set Q;
// Step 1: initialize graph;
for each vertex v in Graph do
    if (source, v) exist then
        dist [v] ← AdjMat(source,v)
    else dist[v] ← ∞ ;
    add v to Q
dist[source] ← 0;
// Step 2: relax edges repeatedly;
while Q is not empty do
    u ← vertex in Q with min dist[u] to source;
    remove u from Q;
    for each neighbor v of u do
        alt ← dist[u] + AdjMat(u,v);
        if alt < dist[v] then
            dist[v] ← alt;
            pred[v] ← u
```

2. SHORT PATH ALGORITHMS

Dijkstra's and Bellman-Ford's sequential algorithms are described in the pseudo code Algorithms 1 and 2, respectively. The main code blocks are indicated by the comments "Steps" 1, 2, and 3. Steps 1 and 2 correspond to initialization of the graph and the relaxation procedure, respectively. In contrast to Dijkstra's, the Bellman-Ford can handle negative weights, having an additional third step to check if there are any negative cycles. In both algorithms, the graph weights are stored in an adjacency matrix (AdjMat), which will be defined later in section 3.

Dijkstra's algorithm: the graph is initialized as de-

Algorithm 2: Bellman-Ford's sequential

```
input  : Graph, source
output: dist [ ], prev [ ]
// Step 1: initialize graph;
for each vertex v in Graph do
    dist [v] ← ∞
dist[source] ← 0;
// Step 2: relax edges repeatedly;
for i ← 1 to Number of vertices−1 do
    for each edge (u, v) in Graph do
        alt ← dist[u] + AdjMat(u,v);
        if alt < dist[v] then
            dist[v] ← alt;
            pred[v] ← u
// Step 3: check for negative-weight cycles;
for each edge (u, v) in Graph do
    alt ← dist[u] + AdjMat(u,v);
    if alt < dist[v] then
        error: "Graph contains negative cycles"
```

scribed in Algorithm 1, i.e. from the source vertex to every other vertex on the graph, and each vertex is added to a priority queue Q. Assuming that $u-$th vertex is being visited with accumulative weight d_u, defined as the sum of weights of the previous visited vertices, the next vertex is chosen by computing the minimum $d_v = w_{uv} + d_u$, where w_{uv} is the weight between the unvisited $v-$th neighboring vertex directly connected to the $u-$th vertex. The visited vertex is then removed from Q. Continuing this process of marking the visited vertices, and moving onto a closest unvisited one, the destination will be eventually reached by following the shortest path.

Bellman-Ford algorithm: the graph is initialized by setting to infinite all distances to the source vertex. In this case there is no priority queue. Therefore, excluding the source vertex, a loop over all vertices is needed as described in Step 2 in Algorithm 2. Then for each visited $u-$th vertex, the next vertex is chosen by computing the minimum $d_v = w_{uv} + d_u$, where the $v-$th is selected after a loop over all neighbors of u. As mentioned above, an extra step for checking negative cycles is required.

3. GRAPH TO ADJACENCY MATRIX

3.1. Graph generation

For comparison purposes, each of the algorithms will target the same problem, i.e. a two-dimensional graph. The graph is created by means of the python library called *NetworkX* [9]. The vertices are randomly generated, being exported to a file with their corresponding position (x, y) and neighbors. For a fairly comparison of both algorithms only undirected graph with positive weights were considered. An example of such a graph is shown in Fig. 1 with an example of shortest path starting at vertex 68 and ending at vertex 56. The clearer is the color of the vertex, the closer is to the end vertex.

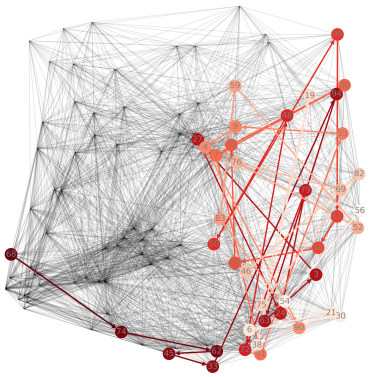

Fig. 1. Example of an undirected graph showing the shortest path. The starting at and ending vertices are 68 and 56, respectively. Observe that the clearer is the color of the vertex, the closer is to the end vertex. The next vertex is indicated by an arrow.

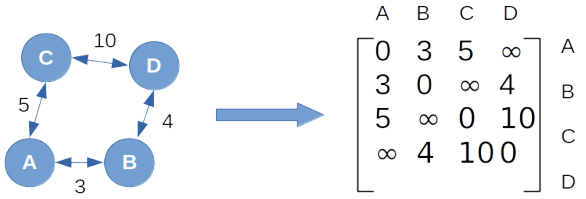

Fig. 2. Graph mapped into an adjacent matrix. Convenient for the parallel strategy explained in section 4 and schematically presented in Fig. 3.

3.2. Adjacency matrix

The graph file is then read using an interface developed during this project. This interface facilities the calculation of the edge weights, defined as the absolute distance between two neighbor vertices, $w_{uv} = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$. Fig. 2 shows the mapping of an undirected graph of four vertices into a 4×4 adjacency matrix. The weights of unconnected vertices are set to infinite, e.g. $w_{AD} = \infty$. This numerical representation of a graph is convenient for the parallelization strategy adopted in this project, using the MPI library, explained in the following section.

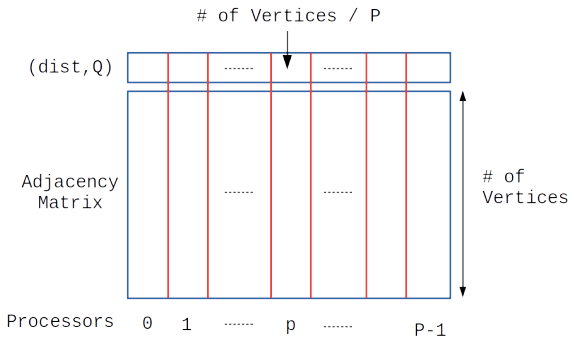

Fig. 3. Splitting the adjacent matrix for the number of processors P. In case of Dijkstra's algorithm, the variable $dist$ and priority queue Q (see Algorithms 1) are also divided over P.

Algorithm 3: Dijkstra's Parallel

```
...;
// In red global variables for all processor;
// Step 2: relax edges repeatedly;
while Q is not empty do
    uloc ← vertex in Qloc with min distloc[uloc];
    MPIAllgather of uloc, distloc[uloc];
    distloc ← vertex with min distloc[uloc];
    Transform umin to u;
    remove umin from Qloc;
    remove u from Q;
    for each neighbor vloc of u do
        alt ← dist[u] + AdjMatLoc(u,vloc);
        if alt < dist[vloc] then
            distloc[vloc] ← alt;
            predloc[vloc] ← u
```

4. PARALLELIZATION STRATEGY

Parallelization of Dijkstra's and Bellman-Ford's algorithms are carried out using the MPI library. The aim is the optimization of their running time. The parallelization strategy was adopted from Ref. [8], being the same for both algorithms and mainly consisting of splitting the adjacency matrix over a number of processors or P cores, as shown in Fig. 3. In case of Dijkstra's algorithm, the variable $dist$ and priority queue Q (see Algorithms 1) are also divided over P. Each part of the adjacent matrix and $dist/Q$ array/set will be then assigned to a processor local variable. Respectively, Algorithms 3 and 4 shows the modifications introduced to Algorithms 1 and 2 due to the MPI parallelization. One can observe that the level of parallelization for Bellman-Ford's is less complex than the one for Dijkstra's. The latter requires to carefully extract the global graph vertex u from the local $distloc[uloc]$. Notice that the global graph variables for all processors are in red.

Algorithm 4: Bellman-Ford's Parallel

```
...;
// In red global variables for all processor;
// Step 2: relax edges repeatedly;
for i ← 1 to Number of vertices−1 do
    for each edge (u, vloc) in Graph do
        alt ← distloc[u] + AdjMatLoc(u,vloc);
        if alt < distloc[vloc] then
            distloc[vloc] ← alt;
            predloc[vloc] ← u
// Step 3: check for negative-weight cycles;
for each edge (u, vloc) in Graph do
    alt ← distloc[u] + AdjMatLoc(u,vloc);
    if alt < distloc[vloc] then
        error: "Graph contains negative cycles"
MPIAllreduce of distloc
```

4.1. Benchmarking

One of the most important parts of the project is to carry out a detailed benchmarking to observe the performance of our parallelization strategy. This will be quantified in running times and in indirect optimization measurements obtained from them, such as speed-up. The latter will allow us to scrutinize the limitation of our approach by looking at the respond to strong and week scaling. For such a purpose, the running time of each algorithm for a certain graph on one processor and on P processors have to be measured. However, the running time at each measurement can vary for the same graph and P depending on the work load of the system. In order to obtain an accurate estimation of the average running time, 100 simulations were performed for each of the experiments. In our case, they were run in two nodes, each node with 28 cores. Each machine is an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.54 GHz with a cache of 35840 K. Our main findings are presented below.

5. DISCUSSION

In order to carry out our experiments, we have generated 4 graphs with 1000, 3000, 5000, and 8000 vertices randomly located. The number of edges is of the order of 10^4 to 10^7. For instance, the graph with 8000 vertices has approximately 3.4×10^7 edges. As previously mentioned, the execution time has been estimated over 100 simulations for each of these graphs on a given number of processors, being $P = 50$ the maximum considered.

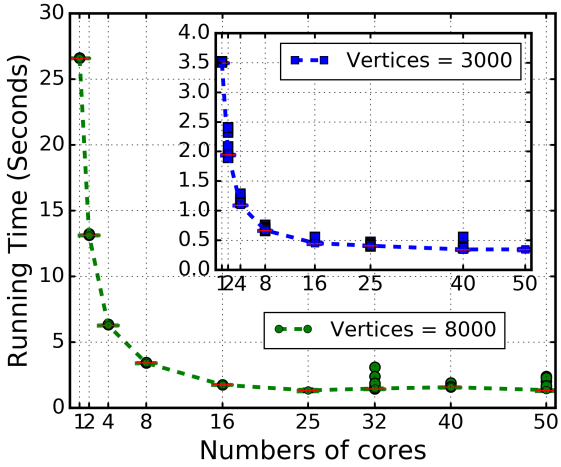

Fig. 4. Dijkstra's algorithm execution time as function of number of cores/processors for a graph with 3000 (inset) and 8000 vertices.

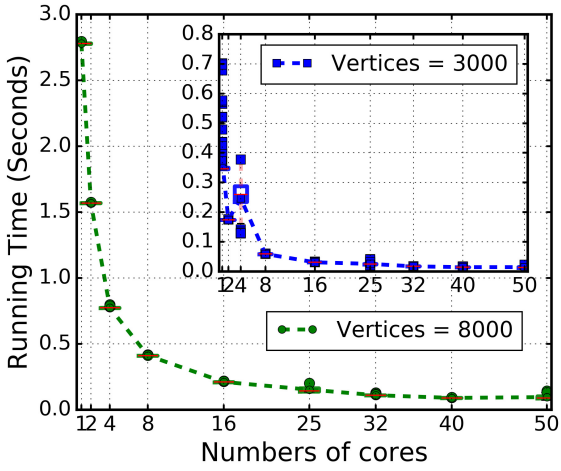

Fig. 5. Bellman-Ford's algorithm execution time as function of number of cores/processors for a graph with 3000 (inset) and 8000 vertices.

Figs. 4 and 5 shows the execution running time as function of processors for Dijkstra's and Bellman-Ford's algorithms, respectively, on graphs with 3000 (inset) and 8000 vertices. Both cases show good scaling up to 25 cores. Although the time variability is more pronounced for lower execution running times, as observed for the graph of 3000 vertices. The variability of times for larger graphs is weaker, particularly for Bellman-Ford's. For higher number of processors ($P > 25$) the time hardly decreased. However, in case of Dijkstra's, one can see a stronger variation of the running time compared to Bellman-Ford's. This effect is independent of the algorithm. It happens when the two nodes start communicating. By increasing the number of processors, more and more communication prevails so that is hard to reduce the execution time, or in other terms to increase

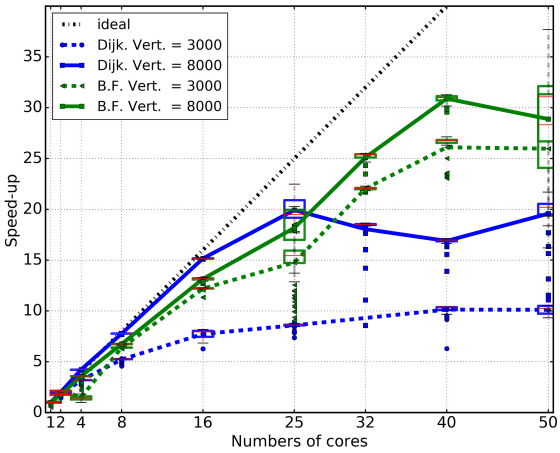

Fig. 6. Speedup for Dijkstra's (blue lines) and Bellman-Ford's (green lines) algorithms. Graphs of 3000 (dashed-lines) and 8000 (solid-lines) vertices. The black dashed-line denotes the ideal speed-up.

the speed-up.

Fig. 6 shows the speed-up, following Amdahl's law, for the times plotted in Figs. 4 and 5 . Solid and dashed curves are for graphs with 8000 and 3000, respectively. We can observe that the speed-up corresponding to Dijkstra's is slightly higher up to $P = 25$, but then Bellman-Ford's shows better scaling for larger number of processors. However, all speed-up curves stagnate. Interestingly, the speed-up curves corresponding to Bellman-Ford's are comparable for both graphs. In contrast, by increasing the number of processors, the difference in the speed-up also increases in case of Dijkstra's, being approximately $2\times$ for $P = 50$. In the range of processors considered here, one can observe that Bellman-Ford's algorithm scales better due to its less complexity of parallelization.

In order to perform a weak scaling study, the size of problem (i.e. the number of vertices in a graph) must now vary when increasing the number of processors. Fig. 7 shows the scaled speed-up following Gustafson's law. Graphs with 1000, 3000, 5000, and 8000 vertices were considered. Respectively, the number of processors for each of the graphs is $P = 2$, 16, 25, and 50. The speed-up for both algorithms has almost a linear behavior indicating a fair scaling. The statistical boxed are also plotted to display the variability of each experiment.

From both Figs. 7 and 6, we could conclude that Dijkstra's algorithm might require more resources in terms of processors to manage larger graphs when compared to Bellman-Ford's. The reason for this difference might be that the Dijkstra's algorithm requires higher communication between processors compared to the Bellman-Ford's.

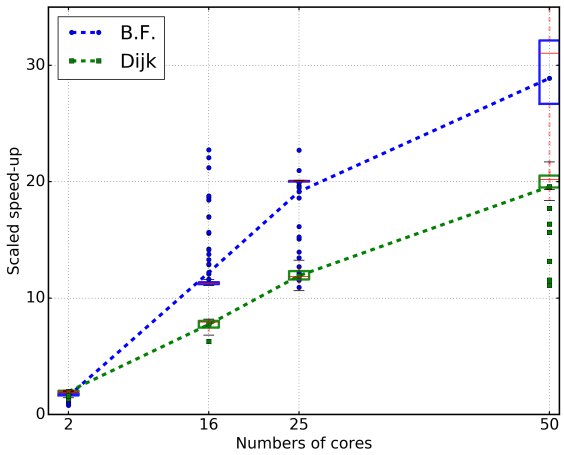

Fig. 7. Weak scaling for Dijkstra's and Bellman-Ford's algorithms calculated by Gustafson's law. Graphs with 1000, 3000, 5000, and 8000 vertices were considered. Respectively, the number of cores/processors for each of the graphs is $P = 2$, 16, 25, and 50.

6. CONCLUSIONS

We have implemented a parallelization scheme, based on MPI, for the well-known Dijkstra's and Bellman-Ford's algorithms for searching the shortest path of a graph. Both strong and weak scaling have been studied to scrutinize the capabilities of our approach. It was found that Dijkstra's execution times are higher that those corresponding to Bellman-Ford's for the same size of the problem and number of processors. Due to less communication and complexity in parallelization, Bellman-Ford's algorithm performs better than Dijkstra's for both strong and weak scaling, particularly for large number of processors, i.e. $P > 25$, and graphs.

Finally, we would like to point out that our approach is limited by the number of vertices in a graph. The latter is due to the vanilla treatment of the adjacency matrix for this project. We believe that by handling the adjacency matrix with more computational sophistication, e.g. using sparse matrices, the same parallel strategy could be employed to tackle larger graphs than those considered in this work.

7. REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[2] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[3] L. R. Ford, "Network flow theory," *Rand Corporation*, p. 923, 1956.

[4] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, "Single-source shortest paths with the parallel boost graph library," .

[5] X. Han, Q. Sun, and J. Fan, "Parallel dijkstra's algorithm based on multi-core and mpi," *Applied Mechanics and Materials*, vol. 441, pp. 750–753, 2013.

[6] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 889–901.

[7] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris, "Employing transactional memory and helper threads to speedup dijkstra's algorithm," in *In ICPP*, 2009, pp. 388–395.

[8] A Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, chapter 10, p. 429, Pearson, 2 edition, 2003.

[9] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, August 2008, pp. 11–15.