Parallel Dijkstra's algorithm with MPI

Bolzonello Enrico, 2087644

enrico.bolzonello@studenti.unipd.it

I. Introduction

Dijkstra's algorithm [3] is undoubtedly the most famous algorithm for solving the Single-Source Shortest Path (SSSP) problem with weighted graphs.
The problem formulation is as follows: given a graph $G = (V, E)$, we want to find a shortest path from a given vertex $s \in V$ to each vertex $v \in V$ [1]. In other words, we want to identify all minimum costs simple paths between the source and all other vertices.
This problem holds significant real-world relevance, finding application across various sectors, for instance in navigation systems, to help users navigate efficiently between two locations, or in routing protocols, to determine the best path for data packets to travel through a network.
Efficiently addressing the Single-Source Shortest Path (SSSP) problem is vital, and the application of parallel computing has the potential to provide valuable assistance in this regard.

II. Approach for Parallelization

Before discussing the parallel implementation, it's important to highlight that the sequential algorithm in use is not optimized to its fullest potential. Specifically, the implemented $O(n^2)$ version, as detailed in [4], contrasts with the more efficient $O(n \log n)$ version employing a heap-based priority queue. This choice was made for two reasons:

- dependency from external sources. The aim was to avoid relying on any external dependencies apart from OpenMPI, and the prospect of implementing a min-heap was firmly ruled out.
- ease of parallelization.

Dijkstra's algorithm is inherently sequential since at each pass it chooses greedily the next vertex based on the following theorem [4]:

Theorem 1. *Assume that the cost L_i of the shortest paths from s to each vertex i belonging to a given set $S \in V$ with $s \in S (L_s = 0)$ is known and let $(v, h) = \arg\min\{L_i + c_{ij} : (i, j) \in \delta^+(S)\}$. If $c_{ij} \geq 0, \forall (i, j) \in A$ then $L_v + c_{vh}$ is the cost of the shortest path from s to h.*

The consequence of this theorem is that fully parallelizing Dijkstra is not possible, since we want to find the shortest distance for all vertices in the graph. The only way to parallelize is to assign a number of vertices to each processor and in each processor compute the distance from the source.
First let's discuss the data division between processors. Let n be the number of vertices, so the adjacency matrix is $n \times n$, and p be the number of processors. If n is divisible by p then n/p vertices can be assigned to a processor: vertex p has vertices $[(p \cdot n/p), (p \cdot n/p + n/p)]$.

In terms of the adjacency matrix, it corresponds to assigning n/p columns to each processor as illustrated in equation 1.

$$\begin{array}{c}\overbrace{\hspace{5em}}^{n/p}\ \overbrace{\hspace{5em}}^{n/p}\hspace{6em}\\ \begin{bmatrix} a_{00} & a_{10} & a_{20} & a_{30} & .. & a_{n0} \\ a_{01} & a_{11} & a_{21} & a_{31} & .. & a_{n1} \\ a_{02} & a_{12} & a_{22} & a_{32} & .. & a_{n2} \\ a_{03} & a_{13} & a_{23} & a_{33} & .. & a_{n3} \\ .. & .. & .. & .. & .. & \\ a_{0n} & a_{1n} & a_{2n} & a_{3n} & .. & a_{nn} \end{bmatrix}\end{array} \tag{1}$$

To achieve this behaviour in MPI, `Scatter` is the perfect candidate since each processor receives the same amount of columns, but custom MPI datatypes are needed to make the scatter work. The types are two:

- send type, which defines a column of the adjacency matrix. It is achieved with a MPI Vector type with $count = n$, which is the number of values in a column, and $stride = n$, since each value of a column is placed at distance n from the previous one thanks to contiguous memory allocation. The schema of this explanation is represented in Figure 1.
  The `create_resized`is needed to tell MPI where to find the next column, which is one value after the current column.

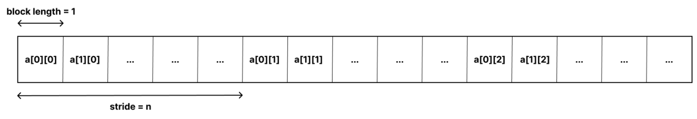

Figure 1: Allocated matrix in memory

- receive type, follows the same logic as the send type but with $stride = n/p$

Now from each processor we can access the data as a normal matrix, being careful that the number of columns is n/p.

The outer loop of the Dijkstra subroutine maintains the same number of executions, what changes is the inside loops since each processor has fewer nodes. Each processor finds the non-visited node closer to the source, sends it to the other processors and computes the minimum of all nodes sent by the processors. This behaviour is achieved by an `Allreduce` with `MINLOC` MPI operation, which works in pairs of values: it finds the minimum value in the former and reports it along with the value that was paired with it [7].
Now every processor has the minimum vertex, so mark it as visited if it belongs to the processor. Subsequently, update all minimum paths and record the predecessor.
Finally, since each processor holds n/p elements both for minimum paths and predecessors, a simple `Gather` suffices to get the full results.

III. Results

The datasets were generated through a Python script utilizing the `networkx` library to produce an Erdós-Rényi random graph, employing two specified parameters as in the documentation [2]:

- n, number of nodes
- p, probability for edge creation

This allowed for two tests: the first fixing p for the scalability on number of nodes, the second, fixing n, to test whether the number of edges exiting a node affects runtime.
All runs have been executed in the Capri platform [6].

I Scalability on n

For the first experiment, p is fixed to 0.5 and the results in Table 1 with different sizes are obtained.

n	file name	time sequential	time 2 processors	time 5 processors	time 10 processors
100	random05.txt	0.000170	0.000471	0.000814	0.001706
500	random04.txt	0.003735	0.006348	0.005087	0.007381
1000	random06.txt	0.015361	0.024530	0.017180	0.018987
2000	random07.txt	0.057728	0.087999	0.061480	0.060349
5000	random08.txt	0.425093	0.567382	0.349344	0.293054
7500	random09.txt	0.943866	1.570785	0.831178	0.711831

Table 1: Runtimes of the first experiment

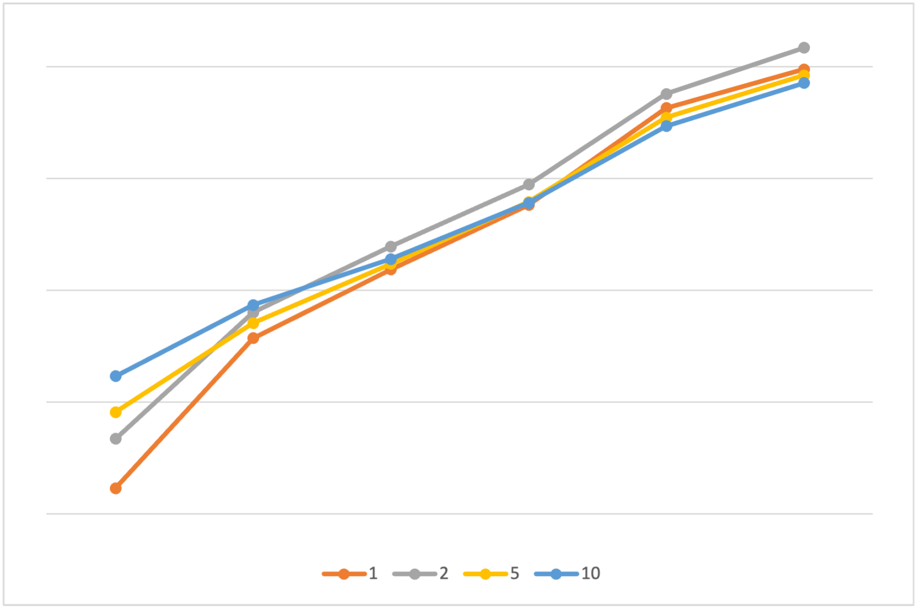

Figure 2: Results plotted on a base-2 logarithmic scale

n	2	5	10
100	0,360934183	0,208845209	0,0996483
500	0,588374291	0,734224494	0,506028993
1000	0,626212801	0,894121071	0,809027229
2000	0,656007455	0,938972023	0,956569289
5000	0,74921834	1,21683212	1,450562012
7500	0,643875425	1,135576254	1,325969226

Table 2: Speed-ups of the first experiment

As we can see in Table 2, the speed-up tends to increase as the number of processors and size of the graph increases. It is interesting to note the fact that with 2 processors the algorithm performs always worst than the sequential algorithm. In all other cases, the speed-up tells us that the use of more processors slightly improves the run-time compared to the sequential algorithm, but not of a factor of p as the aim was.
To better understand the cause of this results, let's analyze the Computing over Communication ratio in Table 3 with $n = 100$.

processors	2	5	10
communication total	0,000219	0,00059	0,000516
execution	0,000263	0,000901	0,00113
total time	0,000482	0,001491	0,001646
ratio	54,56%	60,43%	68,65%

Table 3: Computing over Communication ratio

For simplicity, the execution time also includes the `Allreduce` operation inside the Dijkstra subroutine. The algorithm spends at least 31% of the time on the distribution of the data between processors and the final gathering, but what mainly goes up is the execution time, due to the `Allreduce` since the dataset is the same for all the runs. It is interesting to note that the run with the best speed-up has a Computing-over-Communication ratio of $18,43\%$.

II Constant n, different p

Fixing $n = 5000$ and varying p, the results in Table 4 are obtained.

p	file name	time sequential	time 2 processors	time 5 processors	time 10 processors
0.1	random10.txt	0.214774	0.503825	0.314247	0.268701
0.3	random11.txt	0.288822	0.539787	0.319813	0.260734
0.5	random08.txt	0.425093	0.567382	0.349344	0.293054
0.7	random12.txt	0.365197	0.616920	0.360916	0.270841
0.9	random13.txt	0.261535	0.631541	0.363818	0.283388
1	random14.txt	0.214215	0.660333	0.378930	0.286056

Table 4: Runtimes for the second experiment

In this experiment, it is interesting to observe the behaviour of the run-time as the probability of an edge being created increases. The expected behaviour is that the runtime increases when the probability, and consequently the number of edges exiting from a node, increases.
The runs somewhat follows the theoretical assumption, excluding the run with $p = 0.5$ and 10

p	2	5	10
0.1	0,426286905	0,683456008	0,799304803
0.3	0,53506661	0,903096497	1,107726649
0.5	0,74921834	1,21683212	1,450562012
0.7	0,591968164	1,011861486	1,348381523
0.9	0,414121965	0,718862178	0,922886643
1	0,324404505	0,565315494	0,748856867

Table 5: Speed-ups for the second experiment

processors which is an outlier. A possible explanation for the outlier is that the execution with $p = 0.5$ took place at a distinct time from the other runs, suggesting the possibility that Capri experienced higher congestion during that specific timeframe. The hypothesis proved true since another run executed in a much better time of $0,251142$ seconds.

IV. Conclusions

As anticipated in the second section, Dijkstra's algorithm is inherently sequential, and, as demonstrated by the results, it's parallel formulation performs just slightly better only on big datasets, so the theoretical aim of a speedup in the order of the number of processors is not reached.
Far better algorithms solving SSSP specifically developed for parallel computing exist, one of which is the Δ-stepping algorithm [5].
Furthermore, the requirement that n is divisible by p is too stringent for any real world application, indicating that future work could be done to loosen this constraint.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[2] NetworkX Developers. *fast_gnp_random_graph*. `https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.fast_gnp_random_graph.html`. [Online; accessed 19-December-2023].

[3] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. DOI: `10.1007/BF01386390`. URL: `https://doi.org/10.1007/BF01386390`.

[4] Matteo Fischetti. *Introduction to Mathematical Optimization*. 5th. 2019. ISBN: 9781692792022.

[5] U. Meyer and P. Sanders. "Δ-stepping: a parallelizable shortest path algorithm". In: *Journal of Algorithms* 49.1 (2003). 1998 European Symposium on Algorithms, pp. 114–152. ISSN: 0196-6774. DOI: `https://doi.org/10.1016/S0196-6774(03)00076-2`. URL: `https://www.sciencedirect.com/science/article/pii/S0196677403000762`.

[6] University of Padova Strategic Research Infrastructure Grant 2017. *CAPRI: Calcolo ad Alte Prestazioni per la Ricerca e l'Innovazione*.

[7] RookieHPC. *MPI_MINLOC*. `https://rookiehpc.org/mpi/docs/mpi_minloc/index.html`. [Online; accessed 18-December-2023]. 2019.