DP_Kruskal: a concurrent algorithm to maintain dynamically minimum spanning trees

Daniel Benedí[0009−0005−4295−9484], Amalia Duch[0000−0003−4371−1286], Edelmira Pasarella[0000−0001−8315−4977], and Cristina Zoltan[0000−0001−5303−6371]

Computer Science Department,
Universitat Politècnica de Catalunya
Barcelona Tech
daniel.benedi@estudiantat.upc.edu,{duch, edelmira}@cs.upc.edu

Abstract. We introduce a parallel, distributed and concurrent adaptation of Kruskal algorithm (the DP_Kruskal algorithm) that keeps the input graph distributed along a pipeline and computes online its minimum spanning tree (or forest). We provide a working implementation in the Go programming language that, due to its communication channels and co-routines, fits naturally to our algorithm. We show experimentally that DP_Kruskal scales well to a large number of processes and that it is competitive against the classic sequential Kruskal algorithm and the parallelized version of the Filter_Kruskal algorithm. Our experimental study is done for a large class of dynamic random graphs –including several densities and sizes– and some real dynamic graphs and it reveals that DP_Kruskal is also competitive to maintain the minimum spanning tree of dynamic graphs.

Keywords: Spanning trees · Parallel algorithms · Concurrency · Dynamic pipelines · Dynamic graphs.

1 Introduction

The problem of finding the minimum spanning tree (MST) of a graph is a well known problem in graph theory with many practical applications. In order to improve the efficiency of classical sequential minimum spanning tree algorithms several ways to parallelize them have been studied leading to several algorithms [1,4,12]. Independently of the topology of the input graphs, all these algorithms are generally designed for working either on *in-memory* and/or on static graphs –contrary to dynamic ones. Moreover, since usually the size of nowadays graphs don't fit into the memory of one processor, algorithms capable to work in distributed memory settings are essential in the quest to find an efficient solution to the minimum spanning tree problem.

In this work, we provide a Kruskal-based parallel and concurrent algorithm: DP_Kruskal, defined on the Dynamic Pipeline Model(DPM) [11,13,10]. Besides, we show experimentally that DP_Kruskal is competitive in practice –specially when dealing with dense graphs, where other algorithms fail.

Moreover, the `DP_Kruskal` algorithm is suitable to compute the Dynamic_MST of a fully dynamic graph as well as to maintain it actualised by computing efficiently every update on it. Under this framework the computation of the Dynamic_MST problem turns out to be more natural and intuitive as well as more efficient in several cases since the `DP_Kruskal` algorithm discards –early in (and all along) its execution– several edges.

2 The `DP_Kruskal` Algorithm

A *dynamic pipeline* (DP) consists on different (stateful) stages that run in parallel and are connected by means of communication channels that are in charge of the synchronisation of the procedure [11]. The `DP_Kruskal` algorithm distributes the input graph along a DP and there are two types of channels, the event channel and the graph channel carrying events and MSTs, respectively. Figure 1 depicts the state of an instance of `DP_Kruskal`. The stages of the DP are defined as follows:
Input (I): Whenever this stage receives an event it passes it to the rest of the pipeline through the event channel, if $\texttt{op} = \texttt{mst}$ it also passes the empty set through the graph channel.
Output (O): When an event `mst` arrives to this stage, it outputs the MST_G.
Generator (Gen): Whenever an event $(\texttt{op}, \texttt{e} = \{\texttt{v}, \texttt{w}\})$ arrives to this stage by the event channel, if $\texttt{op} \in \{\texttt{insert}, \texttt{update}\}$, a new instance of Filter stage, $F(v)$, is spawned and added to the pipeline between the last filter and *Gen*. The edge e is stored in the local memory of $F(v)$. Besides, whenever $\texttt{op} = \texttt{mst}$, it passes the MST that arrives by the graph channel to the output stage O.
Filter ($F(v)$): The parameter v is called the *root* of the filter stage. Whenever an event $(\texttt{op}, \texttt{e} = \{\texttt{v}, \texttt{w}\})$ arrives to $F(v)$, the following things can happen: (i) If $\texttt{op} = \{\texttt{insert}, \texttt{update}, \texttt{delete}\}$ and e is incident to the root v, the event is treated in $F(v)$ according to `op`. On the contrary, if it is not incident, the event is passed through the event channel to the next stage of the pipeline; (ii) If $\texttt{op} = \texttt{mst}$, a partial MST is computed and passed to the rest of the pipeline through the graph channel by computing the MST of the graph resulting of the union of the MST that arrives through the graph channel with the tree in $F(v)$.

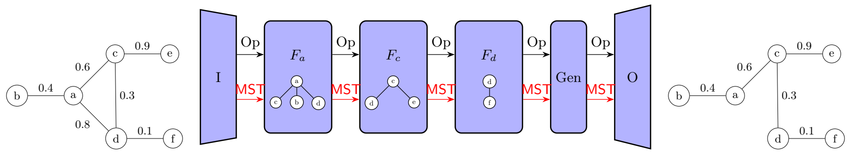
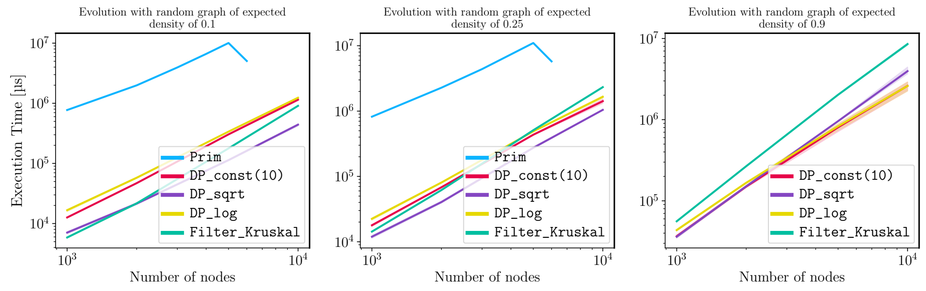


Fig. 1: A `DP_Kruskal` at some point of its execution, the input graph and the MST that it returns. The graph is distributed along filters F_a, F_c and F_d.

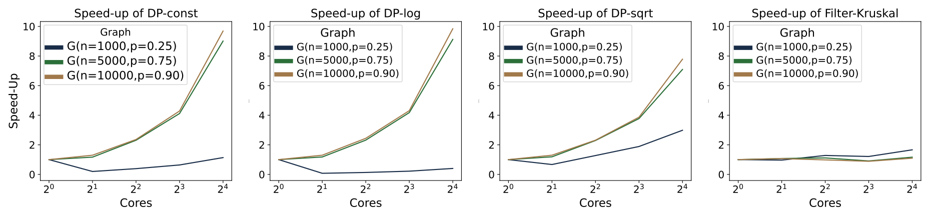
3 Implementation and Experiments

As is standard in parallel applications [2], we recorded the elapsed wall-clock time $T(k, n, m)$—the time elapsed from the start of the execution of `DP_Kruskal` by the first processor until its end by the last processor—for a graph with n vertices and m edges using k processors. Notably, all experiments assumed no prior knowledge of the incoming graph, unlike the classic fully dynamic model where the number of vertices is known in advance, allowing ad-hoc optimizations.

Static MST on random graphs. We generated random graphs where each edge has a probability p of being present, as introduced by Gilbert [5]. Instead of visiting every edge individually, we used Batagelj's method [3], which involves generating a random number to determine how many edges to skip based on the probability p. For each combination of nodes $(10^3, 2 \cdot 10^3, 10^4, 5 \cdot 10^4)$ and edge probability $(0.10, 0.25, 0.50, 0.75, 0.9)$, 20 random graphs were generated.



(a) Comparison of DP_Kruskal with DP_Kruskal_const(10), DP_Kruskal_log, DP_Kruskal_sqrt, Prim and Filter_Kruskal on a single core.



(b) A comparison of the speed-ups of DP_Kruskal and Filter_Kruskal in multicore.

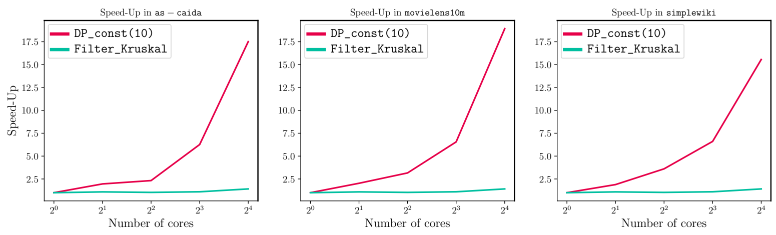
Fig. 2: Experimental results on static random graphs.

To study the effect of the number of roots in each filter, we consider three options: a constant number of roots DP_Kruskal_const, $\log(n)$ roots DP_Kruskal_log, and $\sqrt{n}$ roots DP_Kruskal_sqrt on a single core (where n is the number of vertices of the input graph). We also compared DP_Kruskal with Filter_Kruskal [9], and a message-passing Prim [7] implementation. Filter_Kruskal was chosen for its simple parallelization, while the Prim algorithm was selected for using the same type of parallelism as DP_Kruskal, with results taken from their paper. Experimental results in Figure 2a show that for small graphs, Filter_Kruskal is faster while as graph size and density increase, DP_Kruskal_sqrt becomes the best option. Figure 2b compares different versions of DP_Kruskal with parallelized Filter_Kruskal on a multi-core set-up. We calculated absolute speedup as the execution time of the sequential implementation divided by the parallel implementation $T(k, n, m)$. For graphs with a small expected number of edges, parallel `Filter_Kruskal` can outperform any `DP_Kruskal` variant. As graph density or the number of vertices increases to real-world sizes, `DP_Kruskal` significantly outperforms `Filter_Kruskal` in both speedup and efficiency.

Dynamic MST on real graphs. We compare `DP_Kruskal` and `Filter_Kruskal` on realistic dynamic graphs obtained from `https://DynGraphLab.github.io/` [6]. After each operation of insertion or deletion of an edge, we have asked in for an actualisation of the MST. In Table 3a, we observe that `DP_Kruskal` is effective for maintaining the MST. Figure 3b demonstrates its excellent performance in a parallel environment, showcasing significant scalability.

Dataset	n	op.	`Filter_Kruskal`	`DP_Kruskal`
`as-caida`	31379	119468	1h 30min	1h 19min
`movielens10m`	49847	384585	1h 39min	1h 20min
`simplewiki`	100312	889016	17h 8min	11h 29min

(a) Time, 1 core, 10 roots per filter



(b) Multicore comparison of speed-ups

Fig. 3: Experimental results on real dynamic graphs where n is the number of vertices and op. the number of operations.

Experiments were run at the RDLab-UPC cluster `https://rdlab.cs.upc.edu/` across different nodes with various cores: Intel Xeon: E5-2450 (16), X5675 (12), X5670 (12), X5660 (12), and X5550 (8). Jobs ran with 16GB of RAM and a variable number of cores depending and executed 10 times, average time was reported, with a 24-hour timeout `DP_Kruskal` uses `Golang` 1.20. Available on https://github.com/danielbenedi6/MasterThesis.

4 Final Remarks

We propose the `DP_Kruskal` algorithm. Our preliminary experiments on various random and real graphs demonstrate that our algorithm is particularly competitive for dense graphs, showing improved performance over the `Filter_Kruskal` algorithm for graphs with over $2 \cdot 10^3$ vertices. It also scales well, enhancing efficiency and speed up to 16 cores. Future work will involve comparing our algorithm with a `MapReduce`-based competitor [8] despite not being able to find a working version; evaluating other MST algorithms within the `DP_Kruskal` model; assessing different implementation languages; conducting extensive experiments with larger datasets and testing adaptability to various dynamic graph models.

References

1. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. Journal of Parallel and Distributed Computing **66**(11), 1366–1378 (2006)
2. Bader, D.A., Moret, B.M., Sanders, P.: Algorithm engineering for parallel computation. In: Experimental Algorithmics, pp. 1–23. Springer (2002)
3. Batagelj, V., Brandes, U.: Efficient generation of large random networks. Physical Review E **71**(3) (Mar 2005). https://doi.org/10.1103/physreve.71.036113, http://dx.doi.org/10.1103/PhysRevE.71.036113
4. Durbhakula, S.M.: Parallel minimum spanning tree algorithms and evaluation. arXiv preprint arXiv:2005.06913 (2020)
5. Gilbert, E.N.: Random Graphs. The Annals of Mathematical Statistics **30**(4), 1141 – 1144 (1959). https://doi.org/10.1214/aoms/1177706098, https://doi.org/10.1214/aoms/1177706098
6. Kathrin Hanauer, M.H., Schulz, C.: Recent advances in fully dynamic graph algorithms – a quick reference guide. ACM Journal of Experimental Algorithmics **27**(1), 1–45 (2022), https://doi.org/10.1145/3555806
7. Lončar, V., Škrbić, S., Balaž, A.: Parallelization of minimum spanning tree algorithms using distributed memory architectures. In: Yang, G.C., Ao, S.I., Gelman, L. (eds.) Transactions on Engineering Technologies. pp. 543–554. Springer Netherlands, Dordrecht (2014)
8. Nowicki, K., Onak, K.: Dynamic graph algorithms with batch updates in the massively parallel computation model. In: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 2939–2958. SIAM (2021)
9. Osipov, V., Sanders, P., Singler, J.: The filter-kruskal minimum spanning tree algorithm. In: 2009 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 52–61 (2009). https://doi.org/10.1137/1.9781611972894.5, https://epubs.siam.org/doi/abs/10.1137/1.9781611972894.5
10. Pasarella, E., Vidal, M.E., Zoltan, C.: Comparing mapreduce and pipeline implementations for counting triangles. Electronic proceedings in theoretical computer science **237**, 20–33 (2017)
11. Pasarella, E., Vidal, M.E., Zoltan, C., Royo Sales, J.P.: A computational framework based on the dynamic pipeline approach. Journal of Logical and Algebraic Methods in Programming **139**, 100966 (Jun 2024). https://doi.org/10.1016/j.jlamp.2024.100966, http://dx.doi.org/10.1016/j.jlamp.2024.100966
12. Tripathy, A.R., Ray, B.: A new parallel algorithm for minimum spanning tree (mst). International Journal of Advanced Studies in Computers, Science and Engineering **2**(5), 7 (2013)
13. Zoltan, C., Pasarella, E., Araoz, J., Vidal, M.: The dynamic pipeline paradigm (2019), https://hdl.handle.net/11705/PROLE/2019/017