MINIMUM SPANNING TREE: A PARALLEL APPROACH

Th. Cambier, R. Dang-Nhu, Th. Dardinier

Ecole Polytechnique
Route de Saclay, 91128 Palaiseau Cedex, FRANCE
ETH Zürich, D-INFK
Rämistrasse 101, 8092 Zürich, SWITZERLAND

*C. Trassoudaine**

IMT Atlantique,
655 Avenue du Technople, 29280 Plouzan, FRANCE
EURECOM, Data Science dpt.,
450 route des Chappes, 06410 Biot, FRANCE

ABSTRACT

Many efficient algorithms have been designed to find the minimum spanning tree in connected graphs as this problem is at the heart of a broad scope of applications and other graph theory problems. In this paper we present our implementations of algorithms already present in the literature as well as a new algorithm, Filter Sollin, which combines ideas from Filter Kruskal and parallel implementations of Sollin's algorithm. All the algorithms are tested through the EULER cluster on various benchmarks and compared to the baseline from the Parallel Boost Graph Library.
Keywords: Minimum spanning tree, distributed algorithms, high performance computing cluster

1. INTRODUCTION

A wide range of problems have input sets that can be represented as graphs which is why it is crucial to develop efficient graph algorithms to reduce the computation costs. As graphs are getting larger and larger, classic sequential algorithms are becoming not fast enough to cope with the exploding number of nodes and edges. To further reduce costs, distributed algorithms are ideal to take advantage of the parallel architecture of computers but it is a challenge to correctly handle memory, concurrency or security and one needs to understand parallel software and hardware to design algorithms that scale well. In this paper, we focus on the minimum spanning tree problem which is a classical one in graph theory with applications in network design, cluster analysis or the approximation of NP-hard problems such as the travelling salesman problem.

We implemented five algorithms, two based on Kruskal's sequential implementation and three based on Sollin's and we present here the runtimes and speedups we obtained when running them over various benchmarks on the High Performance Computing cluster EULER.

All our code can be found on our project repository on GitHub at [**?**].

Related work. The implementation of our algorithms is based upon the work in several previously published papers. In [**?**] is described an implementation of the Filter Kruskal algorithm that we present in Section 3.1. In [**?**] are described several parallel implementations of Sollin's algorithm and, notably, Bor-AL and Bor-FAL that we present in Section 3.2. Our implementation of these algorithms is also based upon techniques presented in [**?**]. Both [**?**] and [**?**] were an inspiration when we designed a new algorithm, Filter Sollin that combines the filtering steps from Filter Kruskal and Sollin's parallel implementation (Section 3.2).

2. MINIMUM SPANNING TREE AND SEQUENTIAL ALGORITHMS

In this section, we define the Minimum Spanning Tree (MST) problem and introduce the sequential algorithms upon which are based the parallel implementations of Section 3.

Minimum Spanning Tree. A minimum spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. Here we consider only undirected and connected graphs and we present different algorithms to compute the minimum spanning tree over them.

In the following, we will use V to denote the set of vertices of the graph, E for the set of edges, n for the number of vertices and m for the number of edges.

MST sequential algorithms. The three main sequential algorithms to compute minimum spanning trees are Prim's, Kruskal's and Sollin's (also known as Borůvka's) algorithms. All are greedy approaches :

- Prim : we initialise a tree with a single random vertex and, at each step, we choose the vertex connected to the tree with the smallest-weighted edge and add it to the tree until all vertices are added.
- Kruskal : we sort the edges of the graph by increasing weight and, at each step, we add to the tree the

*The fourth author performed the work while at ETH Zŕich

smallest edge that does not create a cycle.

- Sollin : we initialise each vertex of the graph as a set and, at each step, we add the smallest-weighted edge from every set to another until the tree is formed.

Prim's algorithm computes a minimum spanning tree in $\mathcal{O}(mlogn)$ when implemented with adjacency list representation and a binary heap to keep the order of the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed tree.

Kruskal's algorithm (Algorithm 1) relies on an union-find data structure which maintains a set for each connected component. If we want to add an edge with its extremities in different components, we need to merge the components because they are now connected. The sorting of the edges is done in $\mathcal{O}(mlog(m))$ sequentially and the adding of edges is done in $\mathcal{O}(m)$.

Sollin's algorithm takes $\mathcal{O}(log(n))$ iterations until it terminates because they are V components at the beginning and each iteration reduces the number of components by at least two. Therefore it runs in time $\mathcal{O}(mlog(n))$.

Algorithm 1 kruskal

```
U is the Union-Find structure
sort E by increasing weight
for u, v ∈ E do
    if u and v are in different components in U then
        add u, v to the tree
        join u and v representations in U
    end if
end for
```

3. PARALLEL ALGORITHMS

In this section are presented the parallel algorithms that we implemented for computing the minimum spanning tree in a connected graph. In the following p is the number of cores on which the parallel algorithms are run.

3.1. Parallel Kruskal

We implemented two parallel versions of Kruskal's algorithm presented in Section 2.

Kruskal with parallel sorting. Kruskal's algorithm is inherently sequential as it linearly goes through the edges once these ones have been sorted. Thus, our first implementation of Kruskal's algorithm is the sequential algorithm described in Algorithm 1 preceded by a parallel sorting using TBB (Section 4.1).

Filter Kruskal. The second parallel MST algorithm we consider is a variant of Kruskal's algorithm that is named Filter Kruskal (Algorithm 2).

The idea of this algorithm is to "filter out" some edges that are assumed not part of the MST to reduce sorting time. Firstly, the number of edges is compared to a threshold under which we choose to compute the sequential Kruskal. Then, an edge is chosen as a pivot regarding weight to separate the edges in two weight categories (heavier and lighter edges). A first recursive call is made on the lighter edges. Before making the second recursive calls on the heavier ones, all heavy edges that are within a component of the current forest are removed (filtering step). To finish the algorithm, the results from both calls are merged to form the MST.

In our implementation, the edges are kept in a vector and elements are permuted in-place, as in a quick-sort, without requiring additional amounts of memory which leads to maximal operational intensity. Moreover, the filter and partition functions are both implemented with the partition function from the C++ standard library parallelised with Parallel STL (Section 4.1).

Algorithm 2 filterKruskal

```
if m ≤ threshold then kruskal
else
    find pivot p ∈ E
    partition E in E_≤, E_>
    A_≤ = filterKruskal(E_≤)
    E_> = filter(E_>)
    A_> = filterKruskal(E_>)
    return merge(A_≤, A_>)
end if
```

For the cost analysis of the algorithm, we use $T(m)$ for the runtime of the algorithm with m edges. The choosing of the pivot can be done in $\mathcal{O}(1)$ by sampling 256 edges from the graph using OpenMP and taking the median thanks to Intel TBB (see experimental setup in Section 4.1). The partition of the edges as well as the filtering can both be done in $\mathcal{O}(\frac{m}{p})$ using the partition function from Intel Parallel STL. The first recursive call costs $T(\frac{m}{2})$ and the second one $\mathcal{O}(T(\frac{m}{2}))$ (and far less in practice) because edges have been removed. In the best case, the first recursive call is sufficient to compute the MST and we have $T(m) \leq \mathcal{O}(\frac{m}{p}) + T(\frac{m}{2}))$ which gives $T(m) = \mathcal{O}(\frac{m}{p} + m) = \mathcal{O}(m)$. In the worst case, we have $T(m) \leq \mathcal{O}(\frac{m}{p}) + 2T(\frac{m}{2}))$ which gives $T(m) = \mathcal{O}(\frac{mln(m)}{p} + m)$

3.2. Parallel Sollin

From the three greedy approaches presented in Section 2, Sollin's algorithm is the one that is the most suitable to a parallel adaptation. Here are presented three distributed variants of Sollin's algorithm : Sollin AL, Sollin FAL and

Filter Sollin. All of them are constituted of the same four main steps, iterated until the tree is fully grown :

- Sort the adjacency lists.
- Merge connected components in a single super-vertex.
- Find the edge with minimum weight for each vertex.
- Identify connected components.

Super-vertices are represented thanks to the Union-Find structure already used for Kruskal's algorithm (Section 2). The finding of the edge of minimum weight is done concurrently with OpenMP and the connected component identification is done via pointer-jumping according to [**?**].

Sollin AL. The name Sollin AL comes from the fact that the graph is represented with adjacency lists : a vector keeps for each vertex the list of its incident edges. The vertex array is sorted according to the super-vertex label which leads the vertices with the same super-vertex to be nearby in the array allowing this vertices to be merged efficiently. Then, the adjacency lists are copied in vectors which are more cache-friendly as they allow to avoid false sharing [**?**] and these vectors are sorted concurrently based on the target super-vertex (the super-vertex at the other end of the edge). Thus, self-loops and multiple edges are moved to consecutive locations to be merged.

Sollin FAL. This time, the adjacency lists are replaced by flexible adjacency lists to reduce the cost from sorting before merging the connected components. The flexible adjacency lists are linked lists of adjacency lists composed of the adjacency lists of all the vertices with the same super-vertex : each time we perform the merging step for the connected components, we append the adjacency list of each vertex to their super-vertex adjacency list. Thus, this time, super-vertices keep multiple edges and self-loops in their adjacency lists which simplifies the sorting steps. These edges are now removed during the finding of the edge of minimum weight.

Filter Sollin. Filter Sollin uses the approach already seen in the algorithm Filter Kruskal (Section 3.1) but uses Sollin's algorithm (with the data structure seen in Sollin FAL) instead of Kruskal's when the input size is under a given threshold.

4. EXPERIMENTAL RESULTS

In this section, we present the results of our algorithms on various input sets and discuss their performance against the baselines over different benchmarks.

4.1. Experimental setup

All experiments are run through the EULER cluster on an Hewlett-Packard XL230k Gen10 equipped with two 18-core Intel Xeon Gold (6150 processors, 2.7-3.7 GHz) and 192 GB of DDR4 memory clocked at 2666 MHz. We are using C++17 with GCC v5.2.0, CMake v3.11.4 and OpenMPI v1.6.5. Our algorithms are compiled with the optimisation flag O3. We also use the following dependencies :

OpenMP. OpenMP which is an API that supports shared memory multiprocessing programming in C++.

TBB. Threading Building Blocks (TBB) for the sorting function that takes full advantage of multi-core performance.

PSTL. Parallel STL (PSTL) to have a threaded and vectorized execution of the standard C++ partition function, optimised and validated for Intel(R) 64 processors.

LSB. LibSciBench (LSB) which is a framework containing a performance measurement library for parallel applications and is coupled with an R script for the post-processing of the gathered data [**?**] [**?**].

4.2. Input Sets

The experiments have been performed on the three following input sets :

ER. Erdős-Rényi (ER) random graphs where we specify the number of nodes of the graph and the probability of existence of a vertex between every pair of nodes. The probability is chosen greater than $\frac{ln(n)}{n}$ which is the threshold to have almost surely a connected graph.

PA. Preferential Attachment (PA) graphs where we specify the number of nodes of the graphs and the number of neighbours that a node attaches itself to when added in the graph. In order to have a connected graph, the graph is initialised with three connected vertices and new vertices are added one by one and connected to existing vertices with higher probability for the ones with higher degrees. This type of graph models better existing inputs from the real world such as representations of social network connections.

USA. USA road networks from 9th Implementation DIMACS Challenge dataset which are real input sets from the road networks in various part of the USA. These graphs are connected and undirected.

4.3. Benchmarks

We tested our algorithms on three benchmarks :

Dense graphs. Around 10K nodes and 10M edges with the topology of PA or ER.

Sparse graphs. Around 50K nodes and 2.5M edges with the topology of PA or ER.

USA graphs. Around 300K nodes and 800K edges with the topology of the USA road networks dataset.

We also used a benchmark of smaller graphs with the topology of PA with either 1000 nodes and 100K edges or

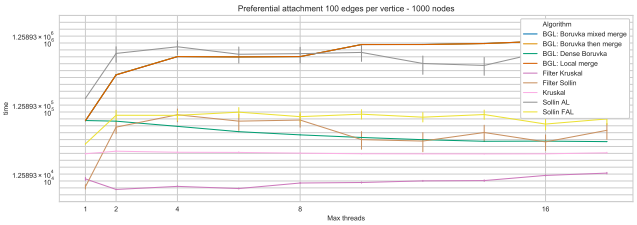

Fig. 1. Runtime graph on PA(1 000, 100)

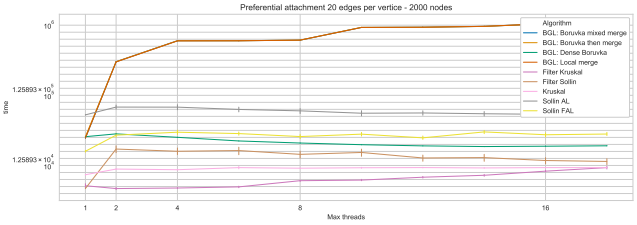

Fig. 2. Runtime graph on PA(2 000, 20)

2000 nodes and 40K edges to compare our algorithms to the baseline.

4.4. Baselines

We compare the results of our algorithms to those of the Parallel Boost Graph Library (PBGL) v1.62.0 which is an extension of the BGL for distributed computing.

4.5. Difficulties

4.6. Results

We present here the results of our experiments on the benchmarks that we studied For every experiment, we generated a large number of random graphs on which three runs were made for each in order to have an appropriate standard deviation.

Baselines.

On Figures 1 and 2 are represented the runtimes of our algorithms in comparison to the ones from the baseline algorithms (Section 4.4). From these figures, we see that

Dense graphs.

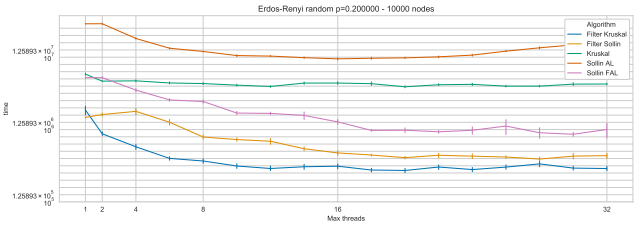

Fig. 3. Runtime graph on ER(10 000, 2 000)

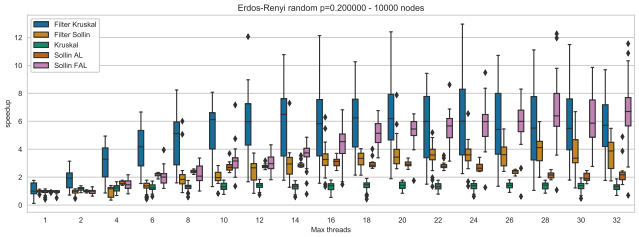

Fig. 4. Speedup graph on ER(10 000, 2 000)

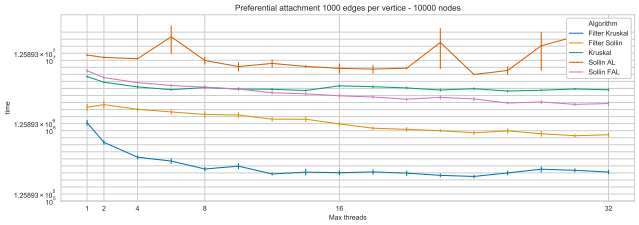

Fig. 5. Runtime graph on PA(10 000, 1 000)

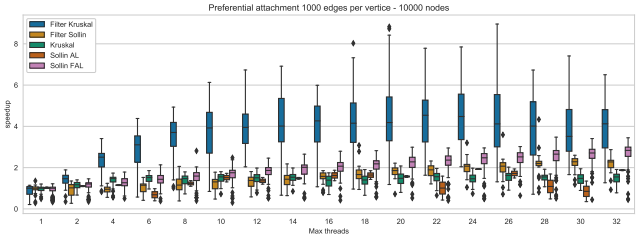

Fig. 6. Speedup graph on PA(10 000, 1 000)

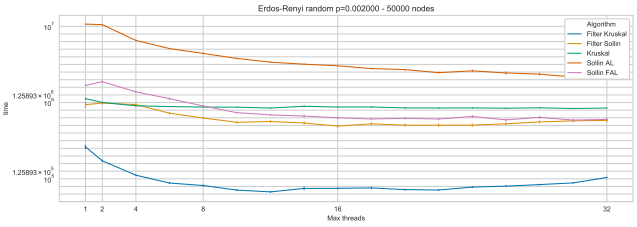

Fig. 7. Runtime graph on ER(50 000, 100)

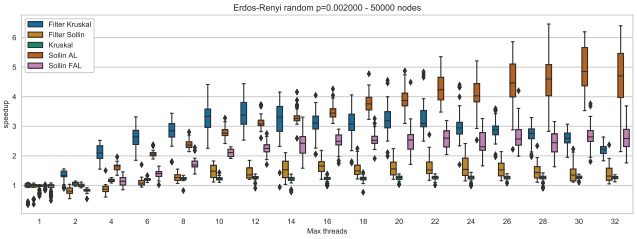

Fig. 8. Speedup graph on ER(50 000, 100)

Sparse graphs.
USA graphs.

5. CONCLUSIONS

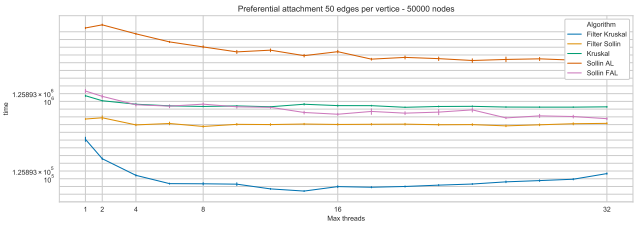

Fig. 9. Runtime graph on PA(50 000, 50)

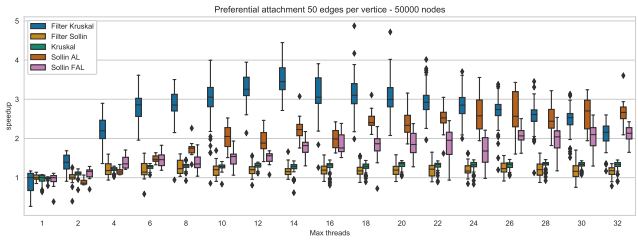

Fig. 10. Speedup graph on PA(50 000, 50)

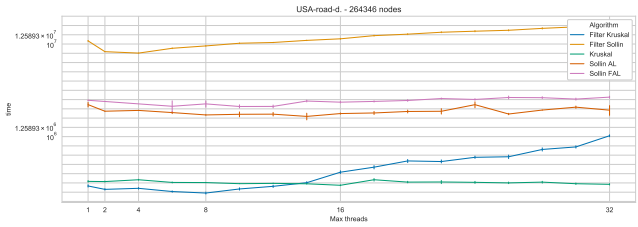

Fig. 11. Runtime graph on USA NY (260K nodes, 730K edges)

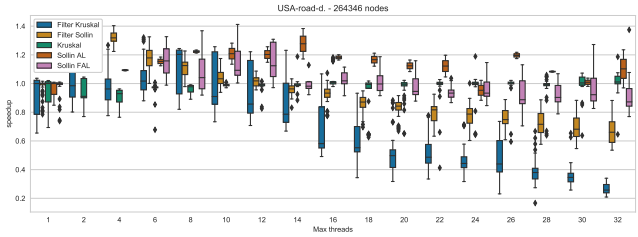

Fig. 12. Speedup graph on USA NY (260K nodes, 730K edges)

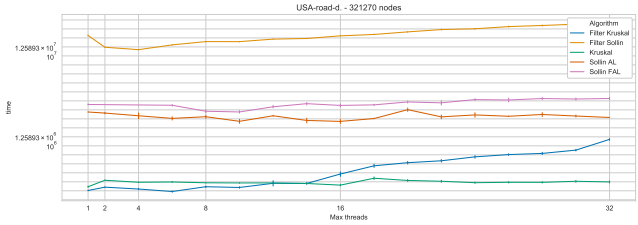

Fig. 13. Runtime graph on USA BAY (320K nodes, 800K edges)

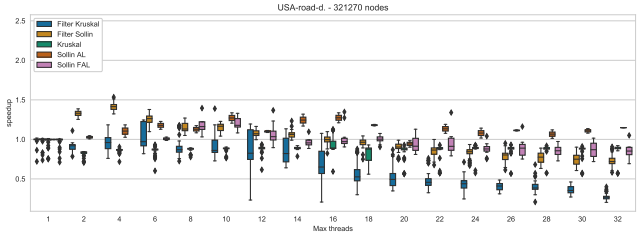

Fig. 14. Speedup graph on USA BAY (320K nodes, 800K edges)