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
Rmistrasse 101, 8092 Zřich, SWITZERLAND

C. Trassoudaine[*]

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

ABSTRACT

ABSTRACT

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.

Motivation.

What are we doing and why / What is done in the paper

Related work.

Some references / what do they do /difference with us / make clear what our contribution is

2. BACKGROUND

In this section, we define the Minimum Spanning Tree (MST) problem and introduce the algorithms we will use for experiments including a cost analysis.

Minimum Spanning Tree problem. 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, n is the number of vertices, m the number of edges in the graph and p the number of cores on which the parallel algorithms are run.

[*]The fourth author performed the work while at ETH Zřich

MST sequential algorithms. The three main sequential algorithms to compute minimum spanning trees are Prim's, Kruskal's and Sollin's (also known as Borvka'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 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.

We chose to implement Prim's algorithm with graphs represented as adjacency lists and to use 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. With such a structure, Prim's algorithm computes a minimum spanning tree in $\mathcal{O}(mlogn)$.

Kruskal's algorithm 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 $\mathcal{O}(mlog(n))$.

MST parallel algorithms. The first parallel MST algorithm we consider is a variant of Kruskal's algorithm named

Filter Kruskal. 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 m 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. 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 Sec. 4). 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 $T(m) = \mathcal{O}(\frac{mln(m)}{p} + m)$

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

- 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.

- 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 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.

3. METHOD

4. EXPERIMENTAL RESULTS

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++11 with GCC v4.9.2, CMake v3.3 and OpenMPI v1.6.5. We also use the following dependencies :

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

- Threading Building Blocks (TBB) to write parallel C++ programs that take full advantage of multi-core performance.

- Parallel STL which offers a portable implementation of threaded and vectorized execution of standard C++ algorithms, optimised and validated for Intel(R) 64 processors.

Benchmarks. benchmarks (input sizes)

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

Results. classes of experiments answering to questions compare code to external benchmarks.

5. CONCLUSIONS

What we did / why important

important results

next steps