Second Assignment Report

High Performance Computing I

2019/2020

João Francisco Veríssimo Dias Esteves
s2679663

November 19, 2019

1 Introduction

This assignment implements a parallel version of Borůvka's *Minimum Spanning Tree* (MST) algorithm using MPI.

2 Implementation

The original algorithm consists of finding the lowest-weighted edges for each vertex and merging the vertices at their ends to form a super-component, or a subtree. This is repeated until all nodes belong to the same component/tree, if the graph is connected, or there are no more edges to analyze. The approach taken here to parallelize it uses the same concept, distributing the edges to different processes and having each one find the lowest-weighted edge for each subtree. Since disconnected graphs must also be supported, the result obtained is a *Minimum Spanning Forest*, or MSF for short, rather than a MST.

The subtrees have been represented by a disjoint-set data structure (*struct Forest*), where each vertex has an associated parent and, for efficiency reasons regarding subtree merges, rank. This allows for constant-time subtree root lookup and merge.

The algorithm goes as follows:

1. Read graph into P0
2. Declare an empty graph for the final MSF
3. Distribute edges from P0 to all the available processes
4. While MSF's number of edges is lower than MSF's number of vertices minus 1 AND new edges were added to the MSF in P0 in the previous iteration (if there was one)
    - 4.1. For each vertex, find the lowest weighted edge connecting different subtrees among the local edges
    - 4.2. Iteratively send all found edges in the previous step to P0 (fig. x)
    - 4.3. Broadcast from P0 to all processes
    - 4.4. For all lowest weighted edges found this iteration, merge their associated subtrees together and, if process is P0, add them to the MSF

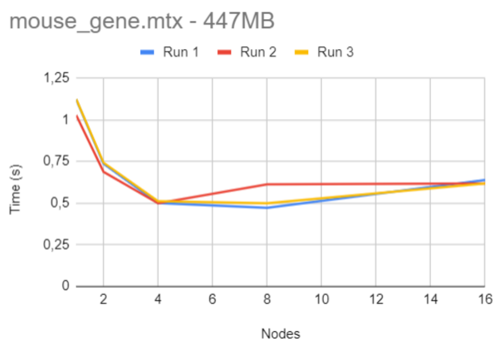
The MatrixMarket files are used as input, where each non-zero entry is an edge between vertexes indexed by the respective row and column. The data is read into a *Graph* struct containing the number of vertices, the number of edges and a dynamic array of all the edges, each being a struct with the source vertex, target vertex and weight.

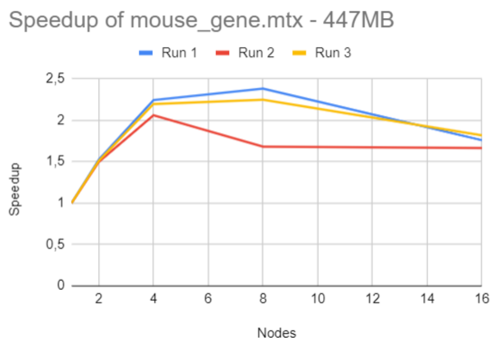
3 Benchmarks

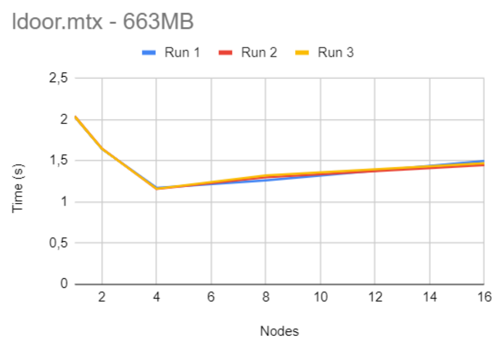
All results obtained from code compiled with OpenMPI using G++ 6.3.0 and the -O2 optimization flag. The times measured didn't take into account the time required to read each file. All configurations were ran 3 times. The configurations were a total combination of the following variables:

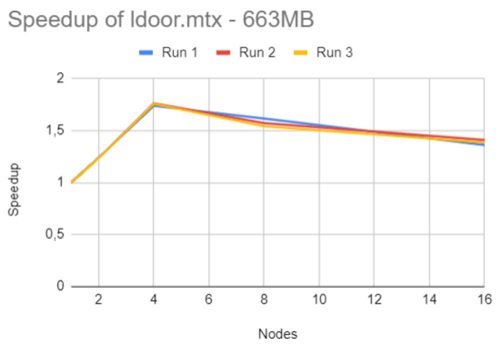
- Nodes: 1, 2, 4, 8, 16
- MatrixMarket inputs: Belcastro/mouse_gene (29M edges), GHS_psdef/ldoor (42M edges), Schenk/nlpkkt240 (760M edges)

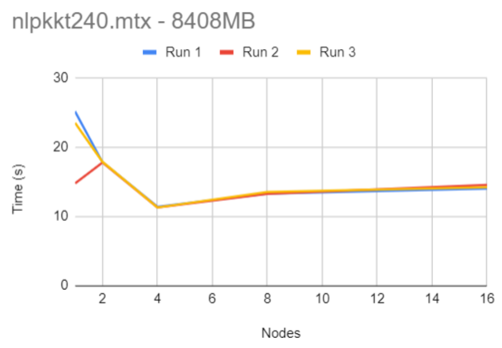
3.1 Results

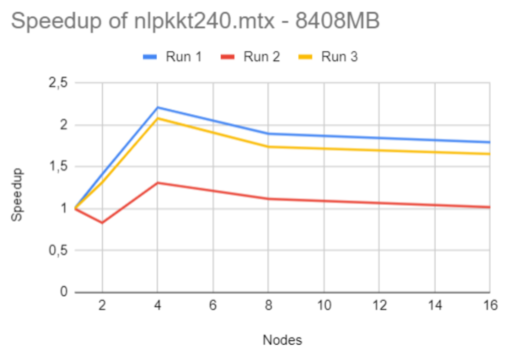












3.2 Analysis

The results were fairly consistent. In general, the algorithm scales well from 1 to 2 and even 4 nodes, but afterwards to 8 and 16 nodes the communication overhead is too great and the performance degrades gradually.