Parallel and Distributed Algorithms and Programs (APPD)
Programming Project

Julien Braine Laureline Pinault

16/10/2019

Source files and documents are available at
`perso.ens-lyon.fr/laureline.pinault/APPD-19-20/Project/`
DEADLINE: *Friday, December 13, 23h59 (Paris time)*

Finding a *minimum spanning tree* (MST) of a graph is a fundamental problem in graph theory with diverse practical applications in computer and communication network design, as well as indirect applications in fields such as computer vision and cluster analysis, etc.

The Minimum Spanning Tree problem can be described as follows. Given an undirected, connected graph $G = (V, E)$ together with a *weight function* w that assign every edge of E a positive weight, the objective is to find a *minimum spanning tree* of G, which is a tree T with vertex set V and minimizing the function $w(T) = \sum_{e \in E(T)} w(e)$. (The correct name should be *minimum weighted spanning tree*; however, almost all literatures use minimum spanning tree for short).

In this project, we revisit two classical algorithms of finding MST in graphs: Prim's and Kruskal's algorithms. Then we investigate some parallel versions of these algorithms using different types of communication networks.

Generating a weighted Graph

To create a random graph, we provide a script named `create-graph.py` that takes the number of vertices $|V|$ in the graph as the first argument, the number of edges $|E|$ in the graph as the second argument, the maximum weight of the edges as the third argument, and the name of the output file for the graph as the fourth argument. Now, try to create a sample graph by typing

`./create-graph.py 10 40 20 graph.txt`.

This will create a graph with 10 vertices and 40 edges, with integer weights between 1 and 20, and write it in the file `graph.txt`. The graph created will be connex (so don't try with n vertices and $n-2$ edges otherwise it will never finish), so you can assume that the graph is connex for your implementations (i.e. you will need to compute a tree and not a forest). Moreover, in your code, you can use `unsigned int` variables as we won't test your programs with very high values for weights and number of nodes.

As usual, we provide a skeleton code `mst-skeleton.c` that you should not touch, and a solution file `mst-solution.c` that you are expected to fill out with your solutions.

Open up the file `mst-solution.c`. There is a skeleton implementation of a function with the signature

`computeMST(int N, int M, int *adj, char *algoName)`.

The arguments of the function correspond to the following:

- **N**: The number of vertices in the graph.
- **M**: The number of edges in the graph.
- **adj**: The adjacency matrix of the graph. It means that `adj[i*N+j]` or `adj[j*N+i]`, for two variables `i` and `j` between 0 and $N-1$, will return the weight of the edge between vertices i and j if it exists, 0 otherwise.
- **algoName**: The name of the algorithm to be executed.

Each MPI rank has the number of vertices and the number of edges (`N,M`), however the adjacency matrix is distributed between all the processes by blocks of rows. It means that, with P processes, process 0 will have the first $\lceil \frac{N}{P} \rceil$ lines of the adjacency matrix (so `adj` will be an array of size $N * \lceil \frac{N}{P} \rceil$), process 1 will have lines $\lceil \frac{N}{P} \rceil + 1$ to $2\lceil \frac{N}{P} \rceil$, and so on, with last process having the last $N - (P-1)\lceil \frac{N}{P} \rceil$ lines of the adjacency matrix. Be careful with that as the array size can be different for the last process. Moreover, we will always run the algorithm with $P < \frac{1+\sqrt{4N}}{2}$ to ensure that all processes have a part of the matrix (or simply: $P \leq \sqrt{N}$).

The function is divided into 4 sections, each corresponding to a specific algorithm name. For the algorithm name **prim-seq**, you will implement a sequential version of Prim's algorithm given in Section 1. For the algorithm name **kruskal-seq**, you will implement a sequential version of Kruskal's algorithm given in Section 1. For **prim-par**, you will implement parallel Prim's algorithm, using all-to-all communication routines (MPI_Bcast, MPI_Reduce, ...). The algorithm is described in Section 2. For **kruskal-par**, you will implement parallel Kruskal's algorithm using point-to-point communication (MPI_Send and MPI_Recv). The algorithm is described in Section 3.

In any case, the goal of the algorithm is to compute the minimum spanning tree of the graph. The process with rank 0 must have the entire minimum spanning tree and you must print a line `i j\n` each time you add a new edge in the (final) tree, that is between vertices i and j with $i < j$. In case of equality between two edge weights, you **must** choose the edge which is the smallest according to the lexicographical order between the pair of vertices that are linked by the edges. For example, if you have two edges of same weight between vertices 4-8 and 10-3, you must choose the edge 10-3 and print `3 10\n` as output of your program, then (if 4-8 can still be added to the tree) `4 8\n`. This is really important as we will use this output to test the validity of your algorithms. We insist on the fact that only the final tree (if you look at parallel Kruskal's algorithm, you will need to compute several trees and merge them) must be printed, by process of rank 0. For Prim's algorithm, the vertex with which you start the spanning tree **must be** the vertex 0, so that you all compute a tree in the same order.

Part 1

Prim's and Kruskal's Algortithms

We first recall two classical algorithms to find a minimum spanning tree (MST) due to Prim and Kruskal. Both algorithm takes as input a connected weighted graph G and return a MST of G. In the following, if two vertices x, y are adjacent, we write $x \sim y$.

PRIM(G)

Input: $G = (V, E)$
Output: a MST of G

1. Initialize:
   (a) Create a tree T with $V(T) = \{x\}$ for an arbitrary $x \in V$ and $E(T) = \emptyset$.
   (b) Create an array D of length $|V|$ where for every $y \in V$,

$$D[y] = \begin{cases} \min_{z \in T, z \sim y} w(y, z) \text{ if } y \text{ is adjacent to } T, \\ 0 \quad \text{otherwise.} \end{cases} \tag{1}$$

2. While $V(T) \neq V$, repeat:
   (a) Find y s.t. $D[y]$ is minimum among all positive elements of D. Add y into $V(T)$ and yz into $E(T)$, where $z \in T$ and $w(y, z) = D[y]$.
   (b) Update D according to (1).

Return: T.

KRUSKAL(G)

Input: $G = (V, E)$
Output: a MST of G

1. Initialize:
   (a) Create a "tree" T with $V(T) = V$ and $E(T) = \emptyset$.
   (b) Create a sorted list E' of (weighted) edges of E.
2. While $E' \neq \emptyset$ and T is not a tree, repeat:
   (a) Select $e \in E'$ with minimum weight (i.e., the first element of E').
   (b) If adding e to T does not create any cycle, then add e to T.
   (c) Remove e from E'.

Return: T.

Question 1

Implement the sequential version of Prim's and Kruskal's algorithms as described above.

Part 2

Parallel Prim's Algorithm with All-to-One Routines

In this section, we consider a parallel version of Prim's Algorithm, running on p processors, $P_0, ..., P_{p-1}$, where each $P_i, i > 0$ is connected to P_0. The algorithm is as follows.

PARALLEL-PRIM(G)

Input: $G = (V, E)$
Output: a MST of G

1. Initialize:
   (a) Partition V into $V_0, ..., V_{p-1}$ such that each contains n/p vertices. Assign V_i and E_i to Processor P_i, where E_i is the set of edges incident with at least one vertex in V_i (which is equivalent to assigning to P_i the rows of the adjacency matrix of G corresponding to V_i).
   (b) P_0 creates a tree T with $V(T) = \{x\}$ for an arbitrary $x \in V_0$ and $E(T) = \emptyset$, and broadcasts to all other processors.
   (c) Each P_i creates an array D_i of length n/p where for every $y \in V_i$,

$$D_i[y] = \begin{cases} \min_{z \in T, z \sim y} w(y, z) \text{ if } y \text{ is adjacent to } T, \\ 0 \quad \text{otherwise.} \end{cases} \tag{2}$$

2. For each processor P_i, find y s.t. $D_i[y]$ is minimum among all positive elements of D_i, and find yz s.t. $z \in T$ and $w(y, z) = D[y]$.
3. Every processor $P_i, i > 0$ sends its edge e_i to P_0.
4. P_0 selects $\min(e_i)$, adds it to T, and broadcasts it to all other processors.
5. Each P_i updates its array D_i according to (2).
6. Repeat Steps 2–5 until $V(T) = V$.

Return: T.

Question 2 *Implement Prim's algorithm in parallel using all-to-one routines as proposed above.*

Part 3

Distributed Kruskal's Algorithm with Point-to-Point Communication

We first describe a subroutine called Merging, which runs on two processors, to merge two MSFs into a single MFS.

MERGE(F_1, F_2)

Input: two set of edges F_1, F_2 (not necessary disjoint) on V.
Output: a MSF of $(V, F_1 \cup F_2)$.

1. P_2 sends F_2 to P_1 then terminates itself.
2. P_1 runs and return KRUSKAL($(V, F_1 \cup F_2)$).

The following Parallel version of Kruskal's algorithm uses a point-to-point communication network with p processors.

PARALLEL-KRUSKAL(G)

Input: $G = (V, E)$
Output: a MST of G

1. Initialize: Partition V into $V_0, ..., V_{p-1}$ such that each contains n/p vertices, and let E_i be the set of edges incident with at least one vertex of V_i. Assign E_i to Processor P_i for every i (which is equivalent to assigning to P_i the rows of the adjacency matrix of G corresponding to V_i).
2. For each processor P_i, run KRUSKAL((V, E_i)) to obtain F_i.
3. Use subroutine MERGE until there is only one processor left. Return the tree of this processor.

Question 3 *Implement Kruskal's algorithm in parallel using point to point communications to merge MSFs as above.*

Part 4

Evaluation of the different algorithms

The goal of this section is to make you think past the asymptotic complexity of algorithms. Indeed, many algorithms have good asymptotic complexity but are not useful in practice. It can also be the case that two different algorithms with the same complexity go faster than the other depending on the input, or platform parameters when we deal with lots of computational nodes and communications.

Question 4 *Propose an evaluation of the scalability and performance of both parallel Prim's and Kruskal's algorithms.*

For this question, we encourage you to use Simgrid in order to estimate the efficiency of the algorithms using different number of processors. You can also try to provide some well-chosen processor topologies or play with the parameters like latency of bandwidth in order to show the limits of one algorithm over the other. We also recall that you can use the script `create-graph.py` to test different inputs with different densities, or matrix sizes if you want to compare the parallel algorithms with the sequential ones (remember Gustafson's law!).

You will provide a small report (4-5 pages should be sufficient) to describe your methodology, the experiments you did (with the files necessary to run them if needed) and your results.
Make sure to include the following points in your report:

- Methodology of the evaluation: are the scenarios created according to an interesting idea of evaluation? is the methodology and the experiments clearly explained?
- Presentation of the results: figures, tables, ... that are clearly readable and well described.
- Conclusions: what can you say about your results, what do your experiments show about the algorithms?

Guidelines

- Source files for the skeleton codes are available at `perso.ens-lyon.fr/laureline.pinault/APPD-19-20/Project/`. In case we update or change something, we will put the updated files to the website, and inform you of this update by e-mail.
- You need to use the C language and the MPI library for programming. Do not use any non-standard C libraries that do not exist in all Linux/Unix distros; code these functions by yourself. Do not use any other language, scripts.
- Normally, you should not need any additional source files. If you like to have more files, make sure that you modify the Makefile accordingly. Your submission must include all the source files (*.c), header files if exists (*.h), I will be using mpicc as the compiler, and running your code on a real parallel machine for benchmark. In any case, never ever try to modify `mst-skeleton.c`.
- Make sure that the Makefile that you provide compiles the code using mpicc on any of the lab computers (Salle E001). If the code does not compile with the Makefile, you will get 0 points.
- Do not print anything else in the code, except the edges of the MST, one per line, as described in the introduction. We will use this output to test the correctness of your code.
- Send your submission as a single tar file with the format `[surname]-[name].tar.gz` to `julien[DOT]braine[AT]ens-lyon.fr` and `laureline[DOT]pinault[AT]ens-lyon.fr`. In the subject line, put "APPD PROJECT [SURNAME] [NAME]". If you need to resubmit, do it by simple versioning. For instance, for the second submission, put "APPD PROJECT [SURNAME] [NAME] V2" in the subject, etc. We will use only the latest version of each submission.