Parallel and Distributed Algorithms and Programs (APPD) Programming Project

Adrien Obrecht
`adrien.obrecht@ens-lyon.fr`

Clara Marcille
`clara.marcille@ens-lyon.fr`

DEADLINE: *Wednesday, December 5, 23h59 (Paris time)*

Flow networks are ubiquitous in computer science. They allow the modelling of many real-world systems such as communication networks. Associated algorithmic developments have had countless applications in solving combinatorial problems, an example being the reknown perfect matching problem on bipartite graphs solved by the Hopcroft-Karp algorithm using flow techniques.

The goal of the programming project whose associated detailed instructions you are currently reading is to implement two algorithms computing maximum flows of flow networks and then evaluate their scalability and relative performance. Each algorithm will come with its own sequential and (distributed) parallel versions. Note that pdf report of the evaluation of performance of your algorithms will count for most of your points, so we recommand starting to write it **while** coding and not at the end !

Project outline

Just as during the practical classes, skeleton and solution files are provided and will be found under the names `max_flow-skeleton.c` and `max_flow-solution.c`. Only the latter is to be modified.

The project is divided in four parts whose themes and relative weights in the final grade are listed below. They are preceded by a description of the provided infrastructure, an introduction to flow networks and the Ford-Fulkerson method, and followed by guidelines.

- Sequential implementation of the Edmonds-Karp algorithm (10 points).
- Parallel implementation of the Edmonds-Karp algorithm (10 points).
- Sequential implementation of the Dinic algorithm (20 points).
- Parallel implementation of the Dinic algorithm (20 points).
- Performance evaluation (40 points).

Flow networks and the Ford-Fulkerson method

To introduce flow networks and the associated maximum flow problem, have a glimpse at the following situation: consider a pipe system with a source from which water flows and a target point where water is to be retrieved. The system consists of pipes structured using crosspoints. Depending on their sizes, the pipes can only take so much water at a time: their capacities are constrained. What is the maximum rate at which water can flow from the source to the target through the pipes ?

This kind of situations is everywhere: electricity in wires, production rate in factories, bandwidth in communication networks, etc.

A *flow network* is a directed graph $G = (V, E)$ in which two distinct vertices $s, t \in V$ are identified and respectively called the *source* and *target* of the network, along with a function $c\colon V \times V \to \mathbb{N}$, where V is a set of *vertices*, $E \subseteq V \times V$ a set of (directed) *edges* and c a *capacity function* associating a positive *capacity* to each edge of the network and a null value otherwise.

A *flow* on a flow network G is a function $f\colon E \to \mathbb{N}$ satisfying two constraints: the *edge rule*, or *capacity constraint* states that the flow of an edge $e \in E$ is at most its capacity:

$$0 \leq f(e) \leq c(e);$$

the *vertex rule*, or *flow conservation constraint* states that the flow entering any non-source and non-target vertex equals the outgoing flow:

$$\forall v \in V \backslash \{s, t\}, \sum_{u \in V, (u,v) \in E} f(u, v) = \sum_{u \in V, (v,u) \in E} f(v, u).$$

The *value* $|f|$ of a flow f on a flow network is defined as the difference between the outgoing and ingoing flows at the source:

$$|f| = \sum_{u \in V, (s,u) \in E} f(s, u) - \sum_{u \in V, (u,s) \in E} f(u, s).$$

A flow is called *maximum* when its value is the greatest possible among all flows of the network.

To simplify the algorithms, it will be assumed that flow networks have neither *loops*, nor *anti-parallel* edges: $(v, v) \notin E$ and if $(u, v) \in E$ then $(v, u) \notin E$.

The most basic method to compute maximum flows relies on the following idea. Given an existing flow, find a path from the source to the target such that every edge along the path still has some available capacity. To encode the notion of "still available capacity" we define residual capacities and networks.

Given a flow network G with some flow f on it, the *residual capacity* $c_f\colon V \times V \to \mathbb{N}$ is defined by

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) \text{ if } (u, v) \in E \\ f(v, u) \text{ if } (v, u) \in E \\ 0 \text{ otherwise.} \end{cases}$$

Note that the residual capacity can be positive for anti-edges of the network. This ensures that on top of adding more flow at an edge, removing some is also possible.

The *residual network* G_f of a flow network G with flow f is defined by $G_f = (V, \{(u, v) \mid c_f(u, v) > 0\})$ with capacity the residual capacity of G with f.

An *augmenting path* P for a flow on a flow network is a path from the source to the target in the residual network. The *capacity* of an augmenting P is defined by $c_f(P) = \min_{e \in P} c_f(e)$. It is the maximum amount of flow that can be added at all edges of the path.

The most basic method of finding maximum flows is the *Ford-Fulkerson* one, which goes as follows:

Algorithm 1: Ford-Fulkerson method

```
Input  : $G = (V, E)$ a flow network with capacity $c\colon V \times V \to \mathbb{N}$
Output : $f$ a maximum flow of $G$
1 $f(e) \leftarrow 0, \forall e \in E$                                  /* Initialization */
2 while there is an augmenting path $P$ in $G_f$ do
3     $f(u, v) \leftarrow f(u, v) + c_f(P), \forall (u, v) \in E$ such that $(u, v) \in P$
4     $f(u, v) \leftarrow f(u, v) - c_f(P), \forall (u, v) \in E$ such that $(v, u) \in P$
```

The two considered algorithms are based on this general method:

- the *Edmonds-Karp* algorithm instantiates the method by using a *Breadth-First-Search* (BFS) algorithm to find augmenting paths among shortest paths;
- the *Dinic* algorithm searches for as many "parallel" augmenting paths as possible before updating the flow, where "parallel" has a particular meaning that will be explained below.

The reader interested in proofs, complexity analysis, and technical details may open any textbook on graph algorithmics.

Generating a flow network

The script named `create-flow.py` can be used to generate random flow networks. It requires the following four arguments in this exact order:

- the number of vertices,
- the number of edges,
- the maximum capacity of an edge,
- the name of the output file.

A sample network with 10 vertices, 40 edges, and maximum capacity 5 can be output in `graph.graph` by executing the following command:

```
python3 create-flow.py 10 40 5 graph.graph
```

An example output is the following:

```
10 40 0 6
2 (6,1) (9,4)
3 (0,1) (4,4) (5,1)
6 (0,1) (1,2) (4,1) (6,4) (7,1) (9,2)
7 (0,2) (1,1) (2,3) (4,5) (5,1) (6,5) (9,5)
3 (0,1) (5,1) (9,4)
4 (0,1) (7,5) (8,2) (9,3)
3 (1,5) (4,1) (5,3)
5 (0,2) (1,3) (4,5) (6,1) (9,2)
5 (2,1) (3,4) (4,5) (6,1) (7,1)
2 (1,4) (8,4)
```

The first line gives some basic characteristics of the graph: vertex and edge numbers, and source and target indices. The ith line for $i > 1$ describes the adjacency list of the $i-2$th vertex. It starts with the length of the list, and continues with a pair made of a vertex index and some capacity for every edge in the list.

You can also visualize this graph using the Graphviz package :

```
python3 graph_to_dot.py graph.graph graph.dot
```

```
dot graph.dot -Tsvg -o out.svg
```

This will first convert the graph to the `dot` format, then produce an svg file out of it. Graphviz may be installed on ubuntu simply by running :

```
sudo apt install graphviz
```

Distribution of the graph

In this project, the graphs you will work with will be distributed among the different processes of a communicator. This means that each process only has the information about the edges of the vertices it handles, and each process has either $\lceil \frac{N}{P} \rceil$ or $\lfloor \frac{N}{P} \rfloor$ vertices to handle. This is a usual assumption. As in general the number of edges M can be of order $\Omega(N^2)$, it is possible that each process can handle data of size $\Theta(N)$ but not $\Theta(M)$. In this case no process can locally store the whole graph in memory and a distribution such as proposed is necessary. As a consequence each vertex of the graph will have two indices:

- a global index as provided in the files generated by the script `create-flow_network.py`;
- a local index relative to the local data of the process that handles it.

For example the third vertex of the second process may have global index $\lceil \frac{N}{P} \rceil + 2$ but local index 2. To better navigate the code, you can rely on the following utility functions where `N` is always the total number of vertices:

- `j = local_of_global_index(v,N)` which returns the local index `j` from the the global one `v`.
- `v = global_of_local_index(j,N)` which returns the global index `v` from the the local one `j`.
- `p = proc_of_vertex(v, N)` which returns the rank of the process owning the (globally indexed) vertex `v`.

The distribution of the data has been done for you using the function `parse_graph` that reads a graph from a file. The following populated variables are provided:

- `N` and `M`: the total number of vertices and edges in the graph;
- `s` and `t`: the global indices of the source and target in the graph;
- `[in/out]_adj`: two arrays that contains the global indices of vertices adjacent to local vertices using *ingoing* or *outoing* edges depending on the specific array.
- `[in/out]_adjBeg`: two arrays containing the information to navigate `[in/out]_adj`: the global indices of vertices adjacent to local vertex `i` using outgoing edges are contained in:

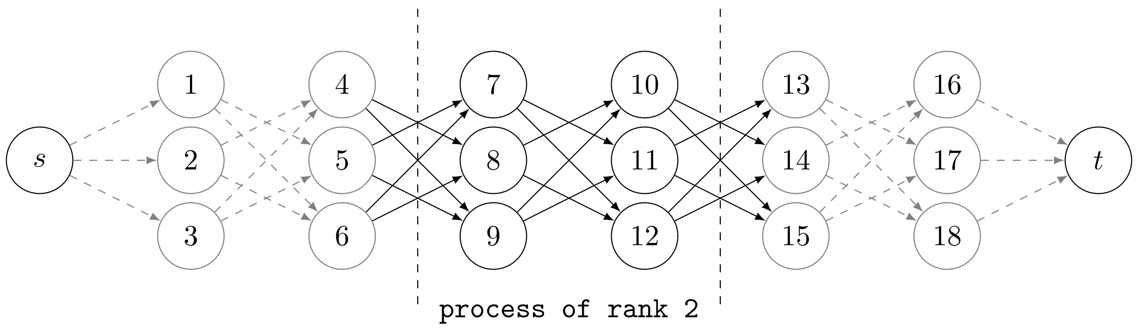
  `out_adj[out_adjBeg[i]]`, `out_adj[out_adjBeg[i]+1]`, ..., `out_adj[out_adjBeg[i + 1]-1]`.

  If there are n local vertices, the global indices of the last local vertex are in

  `out_adj[out_adjBeg[n-1]]`, `out_adj[out_adjBeg[n-1]+1]`, ..., `out_adj[out_adjBeg[n]-1]`.

  In particular `out_adjBeg[n]` is well-defined.
- `[in/out]_w`: two arrays of weights for the local edges which are accessed just as `[in/out]_adj`.
- `[in/out]_adj_size`: two integers containing the number of local ingoing or outgoing edges. It should be used as a parameter for memory allocation.

To summarize: the `i`th outgoing local edge has its global indexed endpoint in `out_adj[i]` and its weight in `out_w[i]`. If `i` $\in$ [`out_adjBeg[j]`, ..., `out_adjBeg[j+1]` $- 1$] then this edge has `j` for its locally indexed startpoint. An illustration can be found in Figure 1.



```
                               inAdjBeg 0,2,4,6,8,10,12
                              outAdjBeg 0,2,4,6,8,10,12
                                  inAdj 5,6,4,6,4,5,8,9,7,9,7,8
                                 outAdj 11,12,10,12,10,11,14,15,13,15,13,14
local_of_global([7,8,9,10,11,12],18)[0,1,2,3,4,5]
```

Figure 1: Some graph and the related structures

Part 1

Performance evaluation (40 points)

Before diving into the algorithms that you will implement, lets talk a bit about what we expect from the pdf report.

Question 1 *Propose an evaluation of the scalability and relative performance of your sequential and parallel implementations of the Edmonds-Karp and Dinic algorithms. You should choose by yourself the quantities you want to study in this part.*

You will write a small report to describe your methodology, the experiments you did, and your results: no more than 4-5 pages. Do not forget adding the files necessary for us to run your tests: reproducibility is of the essence.

You will use Simgrid in order to estimate the efficiency of the algorithms using different number of processors. We recommand starting with a very low latency (or 0) and a well connected network to see the scaling of performance of your parallel algorithms. Expect a significant improvement with parallelisation only for bigger graphs.

Once this works, you can start playing with SimGrid settings, change the latency, bandwith but also the connectivity of the platform : you can use rings, or some well-chosen processor topologies. We provide you two scripts to generate complete graph topologies and ring topologies, but you may create a custom one, or find more platform examples here. You can also see the impact of changing properties about your graph : the number of vertices, the density of edges or even the ordering of the vertices.

You might use a python notebook or any alternatives to generate some graphs or any choosen representation of your data, but please compile everything relevant inside the pdf report. **Any** experiment will be rewarded as long as it is well explained and motivated. Even if some things don't work : talk about it !

Part 2

The Edmonds-Karp sequential and parallel algorithms (10 + 10 points)

Algorithm 2: The Edmonds-Karp sequential algorithm

```
Input   : $G = (V, E)$ a flow network with capacity $c\colon V \times V \to \mathbb{N}$
Output  : $f$ a maximum flow of $G$
1 repeat
2     Do a BFS from the source with the following outputs for each vertex:
          1. distance to the source as given by the BFS,
          2. parent towards the source in the BFS tree,
          3. capacity of the edge shared with the parent.
3     if the BFS found an augmenting path then
4         Find the capacity $c$ of the augmenting path from the target to the source
5         Backtrack and update the flow along the augmenting path
6 until no augmenting path is found by the BFS
7 Compute the value of the flow out of the source
```

Question 2 *Implement a sequential version of the Edmonds-Karp algorithm as described in Algorithm 2. To do so, fill the function* `max_flow_edmonds_karp_seq` *in* `max_flow-solution.c`*.*

Algorithm 3: The Edmonds-Karp parallel algorithm

```
Input   : $G = (V, E)$ a flow network with capacity $c\colon V \times V \to \mathbb{N}$
Output  : $f$ a maximum flow of $G$
1 repeat
2     Execute a parallel BFS from the source with outputs:
          1. depth to the source
          2. parent towards the source
          3. capacity over the edge shared with said parent
            ▷ the updates to the depths, parents, and capacities should be trigerred locally; still,
          after the BFS, all processes should store global versions of them
3     if the BFS found an augmenting path then
4         Find the capacity $c$ of the global augmenting path from the target to the source
5         Backtrack and update the local flow along the augmenting path
6 until no augmenting path is found by the BFS
7 Compute and broadcast the value of the flow out of the source
```

The BFS step deployed in this algorithm is very close from the one you have implemented during classes.

Question 3 *Implement a parallel version of the Edmonds-Karp algorithm as described in Algorithm 3. To do so, fill the function* `max_flow_edmonds_karp_par` *in* `max_flow-solution.c`*.*

To help you visualize the execution of your algorithm, you may generate an execution trace with simgrid. This might help you debug code or understand slowdowns. We provided you instructions about generating traces aswell as a script for visualizing them in the `traces/` folder. If you feel like this figures give good insights about your algorithm, feel free to include them in the report !

The Dinic algorithm

The main issue with the parallel version of the Edmonds-Karp algorithm is the following: its first part, the BFS algorithm, is a real parallel algorithm, but the second part is not: each process does its own local traversal of the augmenting path to update the edge flow.

A solution to this issue is to search for as many augmenting paths as possible, doing the search and the flow updates in parallel. To ensure compatibility between the many found augmenting paths, work is done in the layered network.

Given a flow network G with source and target s and t and flow f, the *layered network* of the residual network of G is defined by

$$G_f^l = (V, \{(u,v) \in E(G_f) \mid d(s,v) < d(s,t) \text{ and } d(s,v) = d(s,u)+1\})$$

where d is the distance function in the residual network G_f. An example of layered network can be found in Figure 1.

Hence the residual network is reduced to successive layers, each new layer being one step further from the source and closer to the target. Because edges in the layered network can only go from one layer to the next, the layered network is actually a *Directed Acyclic Graph*, or DAG. That makes it easier to compute a maximum flow of it. The most basic way is using a *Depth-First-Search* (DFS) algorithm: start a DFS from the source. Each time the target is reached, the DFS describes an augmenting path. Backtrack on this path is performed to update the flow.

At this point all edges on the path whose residual capacities equals the capacity of the path will have no remaining capacities: the DFS cannot use them anymore. Such edges are called *saturated.* A vertex of the layered network is called *saturated* if all its outgoing edges are either saturated or point to saturated vertices.

The DFS starts again at the startpoint of the first saturated edge of the augmenting path. When the whole layered network is saturated, the DFS ends, the flow has been updated, and a new BFS can be performed to compute a new layered network.

Part 3 — Dinic sequential algorithm (20 points)

Algorithm 4: The Dinic sequential algorithm

```
   Input  : $G = (V, E)$ a flow network with capacity $c\colon V \times V \to \mathbb{N}$
   Output : $f$ a maximum flow of $G$
1  repeat
2      Execute a BFS computing distances from the source
3      if the BFS found an augmenting path to the target then
           ▷ start of the DFS
4          $v \leftarrow$ source
5          $c \leftarrow +\infty$
6          repeat
7              Do a DFS step:
8              if $v$ has an unsaturated neighbor $w$ such that $(v,w)$ is not saturated either then
9                  $c \leftarrow \min(c, c(v,w))$
10                 make $v$ the parent of $w$
11                 $v \leftarrow w$
12             else
13                 Mark the reached vertex as saturated
14                 $v \leftarrow$ parent[v]
15                 $c$ must be updated backward
16             if $w =$target then
17                 Backtrack from target to source along the found augmenting path, updating the flow and
                     marking the edges saturated by the path as such
18                 $v \leftarrow$ source
19         until $v =$ source and $v$ saturated
20 until no augmenting path is found in the DFS
21 Compute the value of the flow out of the source
```

Some parts of this pseudo-code are meant to be described at a high-level of abstraction. Line 15 for example implies the existence of some data-structure to remember the values of c at every vertex on the path that is constructed in the DFS. Some design choices have to be made and detailed in an appropriate manner through comments and documentation. The code is still more detailed than what can be found on Wikipedia. Similarly this is meant to be so to give a direction towards an actual implementation of the algorithm. As long as you follow the basic idea behind the Dinic algorithm of doing a BFS followed by a DFS feel free to depart from the pseudo-code above.

Question 4 *Implement a sequential version of the Dinic algorithm as described in Algorithm 4. To do so, fill the function* `max_flow_dinic_seq` *in* `max_flow-solution.c`*.*

Part 4

The Dinic parallel algorithm (20 points)

The last algorithm to implement is a parallel version of the Dinic algorithm. No pseudo-code is provided, only some guidelines based on the sequential version:

1. the BFS must be parallel, but this is in no way sufficient,
2. the DFS can be considered as follows: an augmenting path is built by going from layers to layers, backtracking when reaching a dead end, and updating the flow when reaching the target. This can be done in parallel: one DFS is started on each of the source's neigbors, the different processes keeping track of which edge is locked by which DFS,
3. be mindful of communication overhead,
4. start from the sequential version and work step by step, even if it implies intermediary stupid steps to check the code is functional.
5. Even if the parallel version is less efficient than the sequential one, write about it, explain what you think is happening. You will be rewarded for that.

Question 5 *Implement a parrallel version of the Dinic algorithm following the guidelines above. To do so, fill the function* `max_flow_dinic_par` *in* `max_flow-solution.c`*.*

Guidelines

- Source files are available at the "portail des études".

  In case something is updated you will be informed by email.
- Additional source files should not be needed. In case you do add some make sure that you modify the Makefile accordingly. Submissions must include all the source files (`*.c`) and header files (`*.h`), including the non-modified skeleton.
- **Never ever try to modify** `max_flow-skeleton.c`.
- Makefiles should work using mpicc (or smpicc) on any of the lab computers (Room E001). **A non-compiling code will grant its author a 0 points grade**. Code testing all along the project is encouraged to avoid such a situation.
- Remove any non-necessary `printf` or similar functions from your code as it might interact with the automatic testing of your code.

- Using the C language and the MPI library for programming is mandatory. Do not use non-standard C libraries not existing in all Linux/Unix distributions. Using any other language or script is proscribed except for the performance evaluation part.
- Projects are due on the course's webpage in the form of a tar file with the format `[surname]-[name].tar.gz`

Grading Policy

Your codes will be evaluated according to the following criteria:

- **Correctness**: your program should do as required. In case bugs or incorrect answers are found points will be lost.
- **Speed**: your program performance will be compared to other people's code. Nice optimisation ideas might get you points when they are well documented, even though their impact is minimal. However, do not change the algorithms! Maximum flow algorithms other than the Edmonds-Karp and Dinic algorithms should not be considered.
- **Code quality**: source files will be looked at for indentation consistency, variable and procedure naming, structure (one function having 1000 lines is a bad idea), and, most importantly, documentation in the form of comments. Feel free to encapsulate the arrays parsed from the graph file into a neat structure.

Section 1 will be graded using the following criteria:

- **Relevance of the study**.
- **Methodology of the evaluation**: experiments should be explained and coherent with the quantities under study.
- **Presentation of the results**: figures, tables, and others are welcomed.
- **Conclusions**: consequences should be drawn from the experiments.

Late policy: For each started hour of missing the deadline, a loss of 5 points (out of 100) is to be expected.