Parallelisation of Ford-Fulkerson Algorithm For Maximum Flow Problem

Durga Supriya H L
Information Technology
NITK
Suratkal, Karnataka
durgasupriyahl.201it121@nitk.edu.in

Ranjana Kambhammettu
Information Technology
NITK
Suratkal, Karnataka
ranjanakambhammettu.201it247@nitk.edu.in

Venkata Lalitha Vinukonda
Information Technology
NITK
Suratkal, Karnataka
lalitha.201it265@nitk.edu.in

***Abstract*—The Maximum Flow Problem is a popularly occurring problem in multiple real-life applications like electrical power supply, road network management, airline scheduling etc. A very popularly used algorithm to analyze the maximum flow is the Ford-Fulkerson algorithm. However, it has a high time complexity. Using parallel execution will increase the speed of the execution time, thereby making the use of this algorithm feasible in large real-life problems. This paper explores a parallel approach for the Ford-Fulkerson Algorithm For Maximum Flow Problem and reports the execution times observed.**

I. Introduction

In optimization theory, maximum flow problems involve finding a feasible flow through a flow network that obtains the maximum possible flow rate. The maximum flow problem can be seen as a special case of more complex network flow problems, such as the circulation problem. The maximum value of an s-t flow (i.e., flow from source s to sink t) is equal to the minimum capacity of an s-t cut (i.e., cut severing s from t) in the network, as stated in the max-flow min-cut theorem. In 1955, Lester R. Ford, Jr. and Delbert R. Fulkerson created the first known algorithm to solve the maximum flow problem, the Ford–Fulkerson algorithm. The Ford–Fulkerson method or Ford–Fulkerson algorithm (FFA) is a greedy algorithm that computes the maximum flow in a flow network. The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on. A path with available capacity is called an augmenting path. The time complexity of this algorithm is: $O(VE^2)$. Introducing parallel execution in this algorithm may make the overall runtime faster, and this paper explores how this can be done.

II. Related work

In this section, we give an overview of existing papers which have surveyed and/or compared serial and parallel min-cut/max-flow algorithms.

1) Serial Algorithms: Several papers [1][2][3] provide comparisons of different serial min-cut/max-flow algorithms on a variety of standard benchmark problems. However, many of these benchmark problems no longer reflect the scale and/or structure of modern vision problems solved using min-cut/max-flow, as they consist of small problems and/or grid graphs. Furthermore, they do not include all current state-of-the-art algorithms, while other papers do not include initialization times for the min-cut computation. As shown by Verma and Batra 4], it is important for practical use to include the initialization time, as algorithms (or, specifically, their implementations) may spend as much time on initialization as on the min-cut computation. Finally, existing papers only compare reference implementations of the different algorithms - the exception being that an optimized version of the BK algorithm is sometimes included, e.g. in [3]. However, as implementation details can significantly impact performance [4], it is difficult to separate the practical performance of the algorithm from that of the implementation. As a result, based on the existing literature, it is difficult to assess how much of the performance improvements in more recent algorithms are due to better algorithmic choices versus low-level implementation optimizations.

2) Parallel Algorithms: To our knowledge, a systematic comparison between parallel mincut/max-flow algorithms has not been made. Papers introducing parallel algorithms only compare with serial algorithms 6 or a single parallel algorithm 5 . The most comprehensive comparison so far was made by Shekhovtsov and Hlaváć [7] who included a generic and grid-based parallel algorithm. However, due to the lack of a publicly available implementation, no paper compares with the approach by Liu and Sun, even though it is expected to be the fastest . In addition, all papers use the same set of computer vision problems used to benchmark the serial algorithms. This is not ideal, as the set lacks larger problems and we expect large problems to benefit the most from parallelization. As a result, the performance benefit of parallel algorithms may be underestimated and we lack information about the performance of the parallel algorithms for large problems. In other words, we are likely lacking information about the performance of the parallel algorithms for the size of problems, where parallel algorithms are the most relevant.

III. Serial algorithm

Serial min-cut/max-flow algorithms can be divided into three families: augmenting paths, preflow push-relabel, and pseodoflow algorithms. In this section, we provide an overview of how algorithms from each family work. However, we will

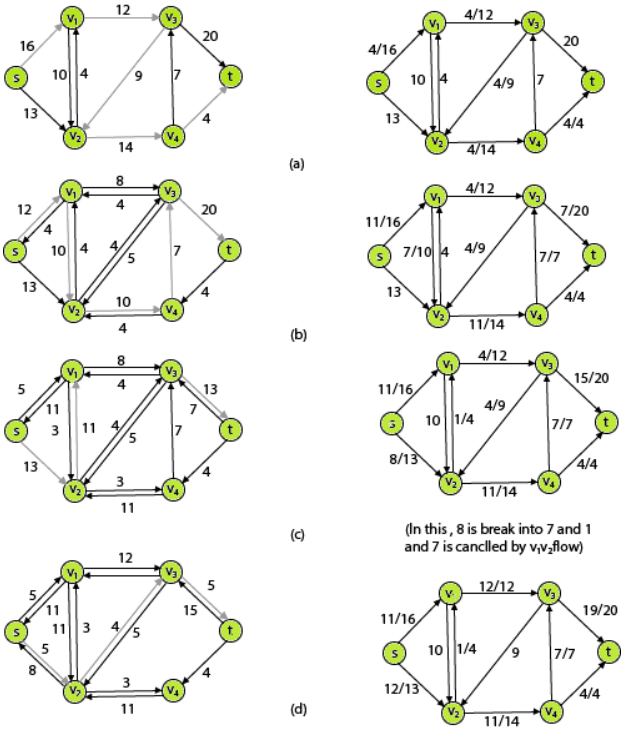

Fig. 1. Example of min cut

first introduce our notation and define the min-cut/max-flow problem.

We define a directed graph $G = (V, E)$ by its set of nodes, V, and its set of directed arcs, E. We let n and m refer to the number of nodes and arcs, respectively. Each arc $(i, j) \in E$ is assigned a non-negative capacity c_{ij}. For min-cut/max-flow problems, we define two special terminal nodes, s and t, which are referred to as the source and sink, respectively. Arcs to and from these nodes are known as terminal arcs.

A feasible flow in the graph G is an assignment of non-negative numbers (flows), f_{ij}, to each arc $(i, j) \in E$. A feasible flow must satisfy the following two constraints: capacity constraints, $f_{ij} \leq c_{ij}$, and conservation constraints, $\sum_{(i,j)\in E} f_{ij} = \sum_{(j,k)\in E} f_{jk}$ for all nodes $j \in V \backslash \{s, t\}$. Capacity constraints ensure that the flow along an arc does not exceed its capacity. Conservation constraints ensure that the flow going into a node equals the flow coming out. A given feasible flow also induces a residual graph where the set of residual arcs, R, is given by

$$R = \{(i, j) \mid (i, j) \in E, f_{ij} < c_{ij} \text{ or } (j, i) \in E, f_{ji} > 0\}.$$

Each of the residual arcs has a residual capacity given by $c'_{ij} = c_{ij} - f_{ij}$ if $(i, j) \in E$ or $c'_{ij} = f_{ji}$ if $(j, i) \in E$. The maximum flow problem refers to finding a feasible flow which maximizes the total flow from the source to the sink.

Finally, an $s-t$ cut refers to partition of the nodes into two disjoint sets S and T such that $s \in S$ and $t \in T$. The sets S and T are referred to as the source and sink sets, respectively. The minimum cut problem, which is the dual of the maximum flow problem, refers to finding an s - t cut which minimizes the sum of capacities for the arcs going from S to T.

A. Augmenting Paths

The augmenting paths (AP) family of min-cut/max-flow algorithms is the oldest of the three families and was introduced with the Ford-Fulkerson algorithm . AP algorithms always maintain a feasible flow and work by pushing flow along so-called augmenting paths, which are paths from s to t in the residual graph. Pushing flow along a path refers to adding a flow, f, to the flow value, f_{ij}, for each arc, (i, j), along the path. To maintain a feasible flow, the maximum flow that can be pushed is equal to the minimum residual capacity along the path. The algorithms terminate when no more augmenting paths can be found.

The primary difference between various AP algorithms lies in how the augmenting paths are found. For computer vision applications, the most popular AP algorithm is the BoykovKolmogorov (BK) algorithm , which works by building search trees from both the source and sink nodes to find augmenting paths and uses a heuristic that favors shorter augmenting paths. The BK algorithm has great performance in practice, but the theoretical run time bound is worse than other algorithms 4 .

In terms of performance, the BK algorithm has been surpassed by the Incremetal Breadth First Search (IBFS) algorithm by Goldberg et al. . The main difference between the two algorithms is that IBFS maintains the source and sink search trees as breadth-first search trees, which results in both a better theoretical run time and better practical performance .

Listing 1. Augmenting path algorithm

```
for (each(u,v) in E)f[u,v]=f[u,v]=0
Build residual network Gf based
on flow f;
while there is an augmenting path p
in Gf do
    c(p) = min (c(u,v):(u,v) in p);
    for each edge (u,v) in p) do
        f[u,v] = f[u,v] + c(p);
        f[v,u]=-f[u,v];
    end for
Rebuild Gf based on new flow f ;
end while
```

B. Preflow Push-Relabel

The second family of algorithms are the preflow push-relabel (PPR) algorithms, which were introduced by Goldberg and Tarjan 224 . These algorithms do not maintain a feasible flow but instead use a so-called preflow, which satisfies capacity constraints but allows nodes to have more incoming than outgoing flow (thereby violating conservation constraints). The difference between the incoming and outgoing flow for a node, i, is denoted as its excess, $e_i \geq 0$. Moreover, PPR algorithms

maintain a labelling, d_i, for every node. If $d_i < n$, it is a lower bound on the distance from node i to t. If $d_i \geq n$, the sink cannot be reached from this node and $n - d_i$ then gives a lower bound on the distance to s. Labels for s and t remain fixed at n and 0 , respectively.

PPR algorithms work by repeatedly applying one of two actions push and relabel. The push operation selects a node with positive excess and pushes flow to a neighbor node with a label of lower value. If no neighbor has a lower label, the relabel operation is used to increase the label of a node by one. The algorithms terminate when no nodes with positive excess have a label $d_i < n$, which means that no more flow can be pushed to the sink. At this point, the minimum s - t cut can be extracted, by inspecting the node labels. If $d_i \geq n$ then $i \in S$, otherwise $i \in T$. Extracting the maximum flow requires an extra step, since the preflow may not be a valid flow. However, this work is generally a small part of the run time 62 and for computer vision applications we are typically only interested in the minimum cut anyway.

The difference between various PPR algorithms lies in the order in which push and relabel operations are performed. Early variants used simple heuristics, such as always pushing flow from the node with the highest label or using a first-in-first-out queue to keep track of nodes with positive excess. More recent versions .

Listing 2. Perflow Push Algorithm

```
Worklist wl = initialisePerflowPush();
while (!wl.isEmpty()){
    Node n = wl.remove();
    n.relabel();
    for(Node w in n.neighbours()){
        pushflow(n,w);
        wl.add(w);
    }
    if (n has excess flow){
        wl.add(n);
    }
}
```

C. Pseudoflow

The most recent family of min-cut/max-flow algorithms is the pseudoflow family, which was introduced with the Hochbaum pseudoflow (HPF) algorithm 30,31]. These algorithms use a socalled pseudoflow and, like the PPR algorithms, do not maintain a feasible flow. A pseudoflow satisfies capacity constraints but not the conservation constraints, as it has no constraints on the difference between incoming and outgoing flow. As with preflow, we refer to the difference between incoming and outgoing flow for a node as its excess, e_i. A positive excess is referred to as a surplus and a negative excess as a deficit. The difference between pseudoflow and preflow is that preflow only allows non-negative excesses.

Pseudoflow algorithms can be seen as a hybrid between AP and PPR algorithms. They maintain a forest of trees, where each node with a surplus or deficit is the root of a tree and all roots must have a surplus or deficit. Let $\mathcal{S}$ and $\mathcal{T}$ denote the forests of trees rooted in s and t, respectively. In each iteration, the algorithms grow these trees by adding nodes with zero excess until an arc, a, connects a tree from $\mathcal{S}$ to a tree from $\mathcal{T}$. This new path from s to t through a is an augmenting path and flow is pushed along it. In contrast to AP algorithms, the only restrictions on how much flow to push are the arc capacities, since pushing flow is allowed to create new deficits or surpluses. If it is not possible to grow a tree - either by adding a free node or finding a connection to a tree in the other forest - the algorithms terminate. As with PPR algorithms, only the minimum cut can be extracted at this point and additional processing is needed to access the maximum flow.

There are two main algorithms in this family: HPF and Excesses Incremental Breadth First Search (EIBFS) . The main differences are the order in which they scan through nodes when looking for an arc connecting $\mathcal{S}$ and $\mathcal{T}$, and how they push flow along augmenting paths. Both have sophisticated heuristics for these choices, which makes use of many of the same ideas developed for PPR algorithms - including a distance labelling scheme to select which nodes from each forest to merge.

IV. Parallel Algorithm

The parallel algorithm and execution will be submitted for the final evaluation. We plan to implement the parallel algorithm as shown in Fig.2.

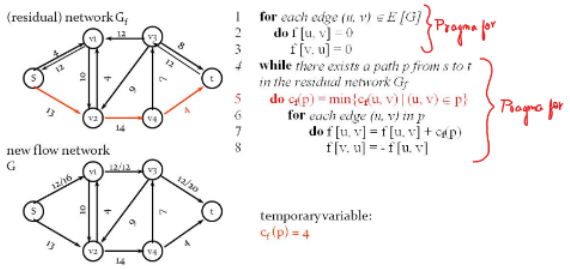

Fig. 2. Parallel Execution

Steps in serial:

Step 1. Initially, label the source $(s, l(s) = \infty)$;

Step 2. Select any node u, that is labeled and un-scanned (If there are not nodes that is labeled and un-scanned, then the current flow is the maximum flow). For all nodes $v \in N(u)$ (where $N(u)$ is the set of all the neighbor nodes of u, i.e. $(u, v) \in E$ or $(v, u) \in E$). If v is unlabeled, then:

- If $(u, v) \in E$ and $f(u, v) < c(u, v)$, then assign the label $(u, +, l(v))$ to node v. where $l(v) = \min(l(u), c(u, v) - f(u, v))$
- If $(v, u) \in E$ and $f(v, u) > 0$, then assign the label $(u, -, l(v))$ to node v. where $l(v) = \min(l(u), f(v, u))$

Then let u be labeled and scanned, meanwhile let v be labeled and un-scanned. If the sink node t is labeled then go to step 3, else return to step 2.

Step 3. let $x = t$, then do the following work until $x = s$.

- If the label of x is $(y, +, l(x))$, then let $f(y, x) = f(y, x) + l(t)$
- If the label of x is $(y, -, l(x))$, then let $f(x, y) = f(x, y) - l(t)$
- Let $x = y$

Then go to step 1 .

The detail of the parallel method is as follows:

Step 1. Initially, label the source $(s, l(s) = \infty)$;

Step 2. For every arc $(u, v) \in E$, only one of the two possible situations may occur:

- u is labeled and un-scanned, v is unlabeled and $f(u, v) < c(u, v)$. If this situation occurs, then assign the label $(u, +, l(v))$ to node v. Where $l(v) = \min(l(u), c(u, v) - f(u, v))$. Let u be labeled and scanned and v is labeled and un-scanned
- v is labeled and un-scanned, u is unlabeled and $f(u, v) > 0$. If this situation occurs, then assign the label $(v, -, l(u))$ to node u. Where $l(u) = \min(l(v), f(u, v))$. Let v be labeled and scanned and u is labeled and un-scanned

If the sink node t is labeled then go to step 3, else return to step 2.

Step 3. let $x = t$, then do the following work until $x = s$.

- If the label of x is $(y, +, l(x))$, then let $f(y, x) = f(y, x) + l(t)$
- If the label of x is $(y, -, l(x))$, then let $f(x, y) = f(x, y) - l(t)$
- Let $x = y$

Then go to step 1 .

We plan to implement parallel execution with 3 different technologies, namely OpenMP, MPI and CUDA and compare the difference in execution time among them.

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. There are several open-source MPI implementations, which fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

V. Results and Analysis

We have implemented the serial algorithm till now. The augmenting path algorithm is used to do the same. Time of execution seems decent when run for small number of nodes but eventually when it comes to real world applications like traffic signals the number of nodes increase by a huge number and so will the time of execution. Below we have considered an example network of nodes from fig [1]. Both manual(fig[1]) and the output from the program implemented are shown.

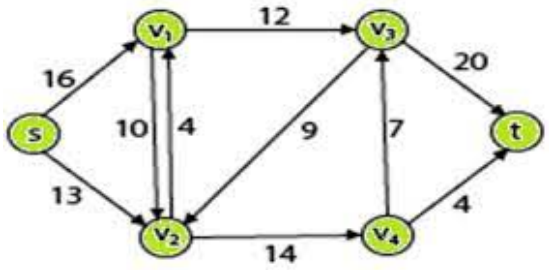

Fig. 3. Network

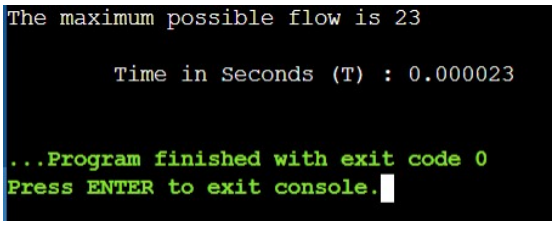

Fig. 4. Output of serial program

VI. Conclusion

From the work done so far, we have concluded that serial execution takes less time for less nodes and edges in a graph, but takes a larger amount of time for a graph with a large number of nodes and edges.

References

[1] Bala G Chandran and Dorit S Hochbaum. "A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem". In: Operations Research 57.2 (2009), pp. 358-376.

[2] B. Fishbain, Dorit S. Hochbaum, and Stefan Mueller. "A competitive study of the pseudoflow algorithm for the minimum s-t cut problem in vision applications". In: Journal of Real-Time Image Processing (JRTIP) 11.3 (2016), pp. 589-609.

[3] Andrew V Goldberg et al. "Faster and More Dynamic Maximum Flow by Incremental Breadth-First Search". In: Proceedings of the European Symposium on Algorithms (ESA). 2015, pp. 619-630.

[4] Tanmay Verma and Dhruv Batra. "MaxFlow Revisited: An Empirical Comparison of Maxflow Algorithms for Dense Vision Problems". In: Proceedings of the British Machine Vision Conference (BMVC). 2012, pp. 1-12.

[5] Miao Yu, Shuhan Shen, and Zhanyi Hu. "Dynamic Parallel and Distributed Graph Cuts". In: IEEE Transactions on Image Processing 25.12 (2015), pp. 5511-5525.

[6] Xiaodong Wu and Danny Z Chen. "Optimal net surface problems with applications". In: Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP). 2002, pp. 1029-1042.

[7] Alexander Shekhovtsov and Václav Hlaváč. "A distributed mincut/-maxflow algorithm combining path augmentation and push-relabel". In: International Journal of Computer Vision (IJCV) 104.3 (2013), pp. 315-342.