# Parallel Maximum Flow Algorithms

Noah Klimisch

*University of Michigan*

**Abstract**

In this paper, we study the maximum flow problem and some parallel algorithms to solve it.

## 1. Introduction

A flow network is a graph $G = (V, E)$ where all edges $(u, v) \in$ E have a non-negative capacity $c_{u,v}$. G has a source $s \in$ V and sink $t \in$ V.
A solution to a flow network in G is a real valued function $f : V \times V \to \mathbb{R}$ that satisfies the following constraints:

$$\text{Capacity Constraint} : f_{u,v} \leq c_{u,v} \forall (u, v) \in E$$
$$\text{Conservation of Flow} : \forall v \in V/\{s, t\} \sum_{u \in V} f_{u,v} = \sum_{u \in V} f_{v,u}$$

The value of a flow is then defined as $\sum_{u \in V} f_{s,u}$. The maximum flow network searches for a flow with the maximum value.

The three parallel algorithms discussed are based on three serial algorithms Ford-Fulkerson, Dinics, and Preflow-Push, which are described in the beginning of each section.

All of the code used in this paper can be found at https://github.com/noahisch/ParallelMaxFlow. The times were taken in Visual Studio Community 2015 on a i7-4710HQ which can run up to 8 threads.

## 2. Ford-Fulkerson

Ford-Fulkerson is the basic algorithm for solving Maximum flow problems. It labels each vertex and pushes flow along the first path found from s to t. However, its simplicity effects its runtime, as it is Psuedo-polynomial.

### *2.1. Serial Algorithm*

The serial algorithm can be broken into 3 steps, where label corresponds to recording the flow to the current vertex.

(1) Unlabel all nodes, push an infinite amount of flow from the source, label source

(2) Select a labeled node that has not been marked, push as much of its flow as it can across all its edges and mark it. If all labeled nodes are marked then end the algorithm, otherwise if the sink is labeled go to step 3.

(3) Push the flow from source to sink, note that this will be based off the sinks label, so in the worst case only one unit of flow is pushed each iteration.

Therefore, the maximum amount of time for each cycle (doing steps 1 2 and 3) is O(E), and a minimum flow of 1 can be pushed each iteration, the runtime is O($|E|f$) where $f$ is the maximum flow of the network.

## *2.2. Parallel Algorithm*

The parallel algorithm has the same step 1 and 3, but instead of choosing one vertex and considering all the connected edges, it considers all available edges (edges between one labeled and unmarked and one unlabeled). It is able to do this because once a vertex has been labeled, flow is never pushed to it again. However, it has to wait for the flow to propagate through each level of the graph. In the worst case each level of the graph has 1 vertex, but in this case it would push all flow through the graph in $O(|V|)$.

The complexity of step 2 is the O($|V|$), so with O($|E|$) processors, the parallel algorithm will run in O($|V|f$) time, which, for a dense graph can be a substantial improvement in runtime. Without $|E|$ processors the runtime of each iteration will not be O(1)[1]

## *2.3. Code Analysis and runtime*

The serial and parallel algorithm were implemented in the same way for steps 1 and 3. For step 2 in the parallel algorithm, a thread pool was created, where whenever a vertex $v$ is labeled all edges to unlabeled neighbors of $v$ are placed in a queue to be completed by the pool.

While the parallel algorithm may be asymptotically superior, having O(E) threads requires a significant hardware investment. Not to mention that since each step 2 must happen at the same time, any amount less than O(E) processors will cause significant slow down due to scheduling and ensuring the correct data is read.
With only 8 threads any graph that is not sparse is orders of magnitude faster with the serial algorithm, and even sparse graphs are still slightly better on the serial.

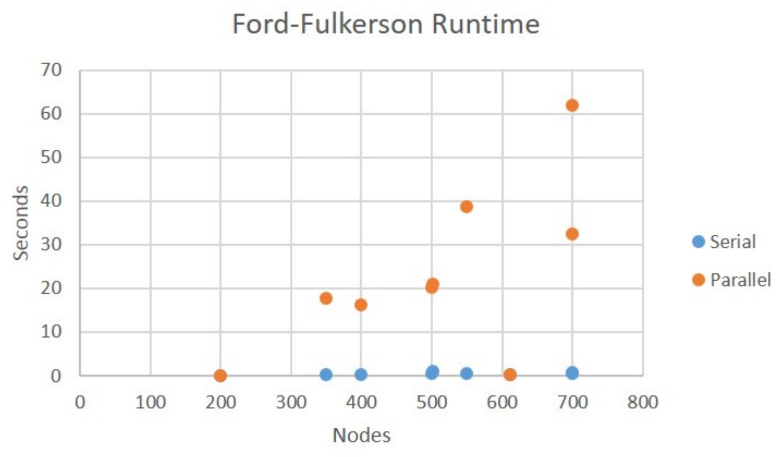

Figure 1: Ford-Fulkerson Node Runtime

While this data does show a trend of increasing runtime, a network around 600 nodes seems to be an outlier. However, this can be accounted for by instead examining the runtime based on edges instead of vertices.

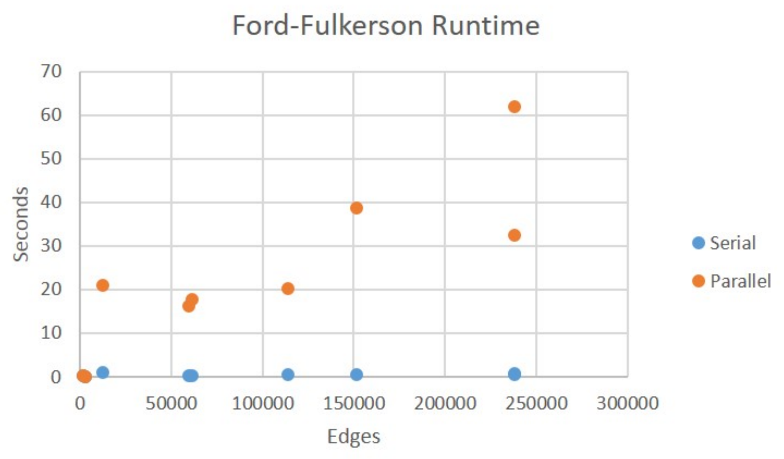

Figure 2: Ford-Fulkerson Edge Runtime

We can now see that the parallel version scales very poorly off edges, because of a significant edge to processor ratio.

## 3. Preflow-Push

While a solution to a flow network must have the conservation of flow constraint, this algorithm relaxes it so that each vertex is allowed to have more flow in then out. It combines this with a height for each vertex, and by restricting flow to only flow downhill Preflow-Push is able to solve maximum flow algorithms in $O(|V|^2|E|)$

### *3.1. Serial Algorithm*

This algorithm has two basic operations, Push and Lift.

- *Push(u, v)*: If the excess of u is greater than 0 and the height of u is equal to the height of v + 1, this will push the maximum amount of flow possible to v.
- *Lift(u)*: If the excess of u is greater than 0, but no neighbors of u are lower than it, it will increase the height of u to be one greater than its lowest neighbor.

This algorithm initializes all the vertices to have an excess of 0 and a height of 0. It then sets the height of the $s$ to be $|V|$ and pushes all available flow. Then it will do all available push and lift operations.

When there are no more pushes or lifts this algorithm will finish and return the maximum flow. The complexity of $O(|V|^2|E|)$ comes from the bound of the number of push/lift operations.

### *3.2. Parllel Algorithm*

This algorithm is slightly adapted from the original Preflow-Push algorithm. The push operation pushes to the lowest neighbor instead of one exactly one less in height.

Since the constraints for pushing is relaxed in this algorithm, the parallel version simply spawns a thread for every vertex, and executes a push/lift operation when one is available.

However, the downside of this is that the bounds on complexity is actually the same as the Preflow-Push serial algorithm. $O(|V|^2|E|)$ since the number of pushes and lifts are still bounded in the same way. [2]

### *3.3. Code Analysis and runtime*

This parallel algorithm was implemented by spawning a thread for each vertex that waits for the excess the node to be greater than 0 (more in flow then out). It then attempts to do a push/lift operation if one is available.

While these two algorithms have the same asymptotic complexity, the parallel version is able to push to more vertices then the serial version, and it can consider multiple vertices at the same time. Even with less than O($|V|$) processors this algorithm has significant speed gains over the serial version.

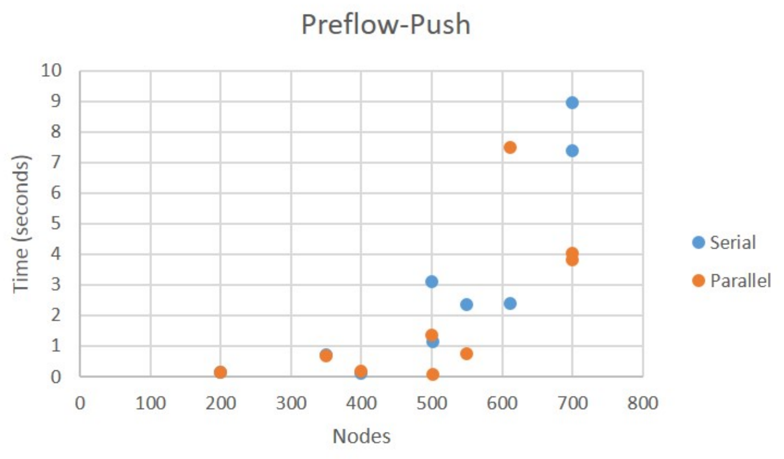

Figure 3: Preflow-Push Runtime

Besides the 600 node test case with very few edges, the parallel version is much faster than its serial counterpart. This is due to being able to modify multiple vertices at once. However, the parallel code is faster also due to the nature of a parallel algorithm. Whenever an verices's excess is above 0, the thread responsible for that vertex can quickly decide what to do, whereas in the serial version, this vertex must be recorded and then wait for the serial algorithm to reach the vertex, and even then it can only push to vertices that have height exactly one less than it.

## 4. Dinic's

Dinic's algorithm transforms the flow network into a layered graph, where each layer only has edges to the next layer, and then solves the max flow for the layered graph by finding a blocking flow. A blocking flow is a flow such that no more flow can be sent to the sink. Each time a blocking flow is found, the a new layered graph is created, and when there is no path from s to t left, the sum of all flows is the maximum flow.

### *4.1. Serial Algorithm*

This algorithm can be broken into 4 steps.

(1) Use BFS to create a layered network

(2) Push as much as capacities will allow from S to the next layer

(3) While there exists unbalanced nodes, push as much as you can forward, and the rest backwards. Blocking the vertex if you cannot push everything forward.

(4) When all nodes are balanced, generate a new layered network.

The complexity of creating a layered network is O(E) ( the complexity of the BFS), and for any layered network they proved that the amount of pushes and returns are $O(|V|^2)$. Since there are at most $O(|V|)$ layered networks (each layered network has the distance from s to t increase by at least 1), this algorithm runs in $O(|V|^3 + |V| \cdot |E|) = O(|V|^3)$.

### *4.2. Parallel Algorithm*

The parallel algorithm uses a Parallel Sum tree. A Parallel Sum tree is a binary tree whose leaf nodes store data, and every parent node's value is the sum of their two children. Using a Parallel Sum Tree allows for multiple threads to access the same data without locking, which will be used to communicate pushes and returns.
Each vertex is assigned four Partial Sum trees, one for flow output, flow received, flow received from vertices pushed to, and the last to coordinate updates. Each edge is assigned one to record flow.
The depths of these trees are $O(\log |V|)$, therefore the complexity of all the pushes/returns in a single layered network is $O(|V|^2 \log |V|)$. However, achieving this complexity with the PS tree can take as much as $O(|V| \cdot |E|)$ processors.

This algorithm then uses Brents theorem, which states that any parallel algorithm with depth d that consists of x elementary operations can be implemented by p processors within a depth of ceil (x / p) + d. Since each unit of time can have all work done in parallel, the total run time of this algorithm can be done with p processors in time ceil(x /p) + d. Where x = $\sum_{i=1}^{d} x_i$, and $x_i$ is the total amount of operations at time i.

With d = $O(|V|^2 \log |V|)$ and x = $O(|V|^3 \log |V|)$, this means that there is a way to achieve a runtime of $O(n^2 \log |V|)$ with $|V|$ processors.

X is $O(|V|^3 \log |V|)$ because each push/return is log n operations due to a PS tree, and there are at most $|V|^2$ push/returns in a layer and $|V|$ layers.

After proving that this is possible, the algorithm uses another PS tree to assign the p processors to their job, to achieve a runtime of $O(|V|^3 \log |V|)/|V| + |V|^2 \log |V| = O(|V|^2 \log |V|)$ parallel run time. [3]

*4.3. Code Analysis and Runtime*

The parallel sum tree is implemented as a vector of unions of ints and the leaf data type. In this way it is able to efficient lookup the leaf nodes in O(1) time, and the data can be stored in a cache-efficient way.
The rest of the algorithm is implemented as described, using a Partial Sum Tree to coordinate with a thread pool to decide which jobs to run.

Asymptotically, a $O(|V|^3)$ should be much slower than a $O(|V|^2 \log |V|)$ algorithm. However, while the cost of the big operations of the parallel Dinic's algorithm is only $O(|V|^2 \log |V|/p)$ the set up and memory costs are much higher than all the other algorithms discussed. The computer used to test these algorithms could not run any of the other test cases without getting a bad allocation error. This is most likely due to the T-in tree, which has $2 \cdot |V| \cdot d_{in}$, where $d_{in}$ is the number of edges in to the vertex, active leaves. In a completed connected graph with 500 nodes, this would mean each T-in tree would have 500,000 active leaves, which requires $2^{20}$ nodes in the Partial Sum tree. Therefore, the time tests were conducted with much smaller test cases.

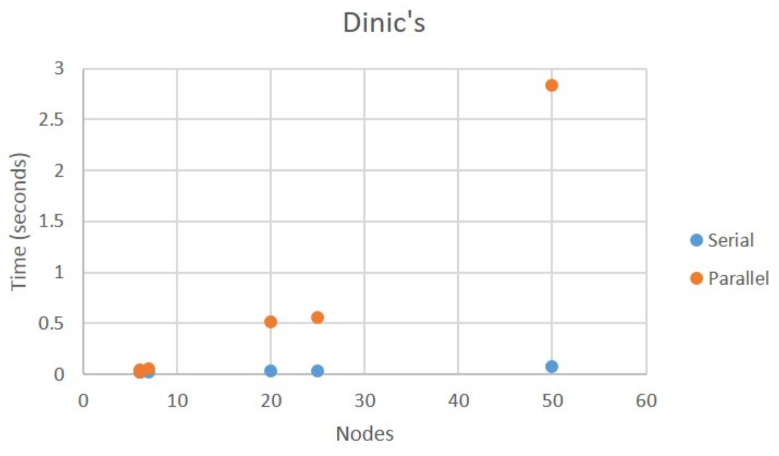

Figure 4: Dinic's Runtime

The setup for the parallel structures of this algorithm take too long for it to compete with the serial $O(|V|^3)$ algorithm, at least for these smaller test cases. This algorithm requires too much memory to be a viable alternative to the serial algorithm.

## 5. Discussion

While the Maximum flow problem may not seem like an easily parallelizable problem, it is evident that there have been successful parallelizations of existing algorithms. The main focus of parallelization for each algorithm has been spawning threads to either manage a specific vertex, or a specific edge. In doing so, the transformed algorithms are able to solve sub problems in the maximum flow that might have been bounded by $O(|V|)$ or $O(|E|)$ in $O(1)$ time.

It is important to notice that the algorithms analyzed in this paper are all bounded by layers. Each algorithm focuses on improving the speed of examining groups of vertices that usually correspond to layers in a layered network. This is inherent in the Parallel Dinic's algorithm, but this is also what the Parallel Ford-Fulkerson algorithm and Parallel Preflow-Push algorithm are doing, since they cannot examine a vertex until the flow from the source has reached that vertex. Therefore, all three parallel algorithms are creating layers, but in the Parallel Preflow-Push algorithm this does not affect the worst case run time of the algorithm, since it does not improve the bounds on pushes and lifts.

By modifying existing algorithms to solve layers more efficiently, these algorithms manage to asymptotically speed up their serial counterparts, however, these existing algorithms are not based on the advantage of modifying an entire layer in $O(1)$ time, and as such these parallel algorithms are far from the optimal. Especially in the case of the parallel Dinic's, which requires too much memory to achieve it's bound. In fact, the serial Dinic's algorithm is actually the fastest algorithm examined here, so the $O(|V|^3)$ algorithm proposed in the paper is more applicable in the common case then the serial counterpart.

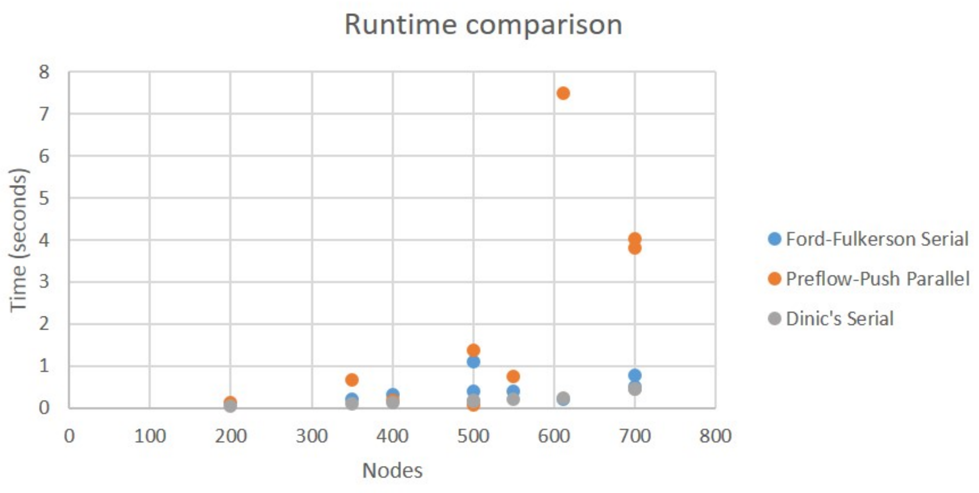

Figure 5: Comparison of the fastest algorithms

## References

[1] Jiang, Zhipeng, Xiaodong Hu, and Suixiang Gao. "A Parallel Ford-Fulkerson Algorithm For Maximum Flow Problem." (n.d.): n. pag. PDF File.

[2] Hang, Bo. "A Lock-free Multi-threaded Algorithm for the Maximum Flow Problem." (n.d.): n. pag. PDF File.

[3] Shiloach, Yossi, and Uzi Vishkin. "An O(n2log N) Parallel Max-flow Algorithm." An O(n2log N) Parallel Max-flow Algorithm. Technion-Israel Institute of Technology, n.d. PDF File. 23 Feb. 2016.