Parallelization of Network Flow

Neo Camanga, Federico Baron, Eric Hartwell, Brian Moon, John Pham
University of Central Florida
COP 4520 Parallel and Distributed Processing

Abstract—**In graph theory, the field of Network Flow determines the maximum flow possible from the source node to the sink node. In a network/transportation flow graph, each edge represents a flow and capacity. Network flow's various real-world applications include infrastructure management such as roads, pipes, or electrical systems, or things more abstract such as team pairing, or ecology systems such as food webs. There are many algorithms which serve to find the maximum flow of a graph. In this pa- per, we explore three algorithms and we parallelize them to in- crease efficiency to find the max flow. The algorithms are Ford- Fulkerson, Edmonds-Karp, and Dinics.**

Index Terms—**Network Flow, Parallel computation, Ford-Fulkerson, Edmonds-Karp, Dinics**

I. Introduction

Research pertaining to the max flow solution of network flow problems traces back to Ford and Fulkerson's paper published in 1962. Since then several other algorithms have been published that are capable of computing these max flows with increasingly better run times. In 1972, Edmonds and Karp proposed a modification to the original Ford-Fulkerson algorithm which helped stabilize the run time and guarantee completion. Around the same time, Dinitz, proposed his own solution to max flow problem. Each of these algorithms uses the augmenting paths approach to finding max flow, and these types of algorithms have not been heavily parallelized in the past. Our research attempts to discover parallelization options for these algorithms and compares them to their sequential counterparts as well as each other.

II. Ford Fulkerson

A. Sequential Algorithm

Ford-Fulkerson (FF) is a greedy algorithm to get the maximum flow of a graph. The typical implementation of FF is through using Depth First Search (DFS) in order to continuously augment paths through the residual graph, where an augmenting path is a path of edges in the residual graph with unused capacity greater than 0 from the source to the sink. [2]

To augment the graph means to find a path from the source to the sink in which The other algorithms described in the other sections are variations based off of FF; however, the goal of this section of the project seeks to parallelize the traditional DFS approach [3] of FF as other implementations such as Edmond Karp or Dinics takes advantage of specific graph structures.

1) Pseudo Code of Sequential: Note if there are n nodes, each node will be indexed from 0 through $n-1$. In the algorithm, the Source node is indexed as 0 while the Sink node is indexed as $n-1$. See Algorithm 1

B. Parallel Algorithm

1) Research: The approach used in a parallel algorithm of Ford-Fulkerson already posed intimidating challenges. Unlike its sibling algorithms, Edmond-Karp in particular, FF is traditionally implemented using DFS, not breadth-first-search (BFS). Without the luxury of a an easily-parallelizable algorithm, research had to be done to convert a recursive DFS approach into an iterative one.

During a run of FF, the algorithm attempts to find augmenting paths by conducting two primary steps: searching for a valid flow-augmenting path, and then proceeding to update the flow-augmenting paths on a backward pass.

2) Algorithmic Approach: Three options of parallelization were considered during the process of Ford-Fulkerson:

1) Depth-First Search
2) Augmenting Paths
3) Backwards Update

Attempting to create an iterative version of DFS would have involved the usage of additional overhead in order to simulate recursion, such as the introduction of a stack.

Rather than navigating through the difficulties of directly turning DFS parallel, a hybrid solution was attempted, where the process of traversing a graph's adjacency matrix was combined with labeling the current flow running through the network flow graph. In the end the attempted solution combines the greedy approach from augmenting the paths by overlapping threads across different connected edges.

3) Initial Approach: The adjacency matrix used to represent the flow and the capacity could be parsed in two different ways. Instead of utilizing a traditional row $\rightarrow$ column approach to navigate through the matrix, instead a column $\rightarrow$ row method will be chosen in order to avoid overhead issues that could arise with multiple threads. The Initial Approach is the same as the Final Approach below with the exception of the direction of "sweeping" when labeling a node. In the Initial Approach, threads would sweep from left-to-right in the capacity matrix column-by-column. Unfortunately, this particular method of labeling the nodes raises contention issues in the form of multiple threads attempting to acquire a lock if for example: we were labeling the flow from node S to A.

4) Final Approach: The chosen parallel approach allows multiple threads to search through the graph's adjacency matrix and interdependently adjust potential flow as all threads traverse the matrix together column-by-columns.

In the Final Approach, threads would sweep from top-to-bottom in the capacity matrix. This is an improvement compared to the Initial Approach since there will be no contention

Algorithm 1 Ford-Fulkerson Pseudo Code

```
function AUGMENT
    $tN \leftarrow sinknode$
    $augNum \leftarrow Integer$
    while $tN$ is not 0 do
        if $tN.thisNode$ then
            $flow[tN.prevNode][tN.thisNode] \leftarrow$ flow[tN.prevNode][tN.thisNode] + augNum
        else
            $flow[tN.thisNode][tN.prevNode] \leftarrow$ flow[tN.thisNode][tN.prevNode] - augNum
        end if
        $tN \leftarrow tN$'s previous node
    end while
end function


function LABEL
    if $add$ : then
        $pFlow \leftarrow$ min(ith-node's potentialFlow, cap[i][j] - flow[i][j])
    else
        $pFlow \leftarrow$ min(ith-node's potentialFlow, flow[j][i])
    end if
end function
```

```
Node:
$prevNode \leftarrow$ Integer
$thisNode \leftarrow$ Integer
$add \leftarrow$ Boolean
$potentialFlow \leftarrow$ Integer

FordFulkerson:
$capacity[][] \leftarrow$ Capacity Graph
$flow[][] \leftarrow$ Flow Graph
$nodeLabels \leftarrow$ Array of Nodes
$numNodes \leftarrow$ Number of nodes in graph

$sink \leftarrow$ Integer

for i in range 2: do
    for i in range numNodes: do
        for j in range numNodes: do
            if jth-node is labeled then

                if ith-node is not labeled and $flow[j][i] < cap[j][i]$ then Label() where $add$ is true
                end if
                if ith-node is not labeled and $flow[j][i] > 0$ then Label() where $add$ is false
                end if
            end if
        end for
        if sink node is labeled then Augment() then do outermost for-loop again
        end if
    end for
end for

```

when it comes to labeling a node. This occurs since each thread will attempt to assign a label to a different node.

The method of labeling was discovered independent of existing research papers while tracing through multiple scenarios of multiple threads attempting to augment the graph. An important conclusion was that when augmenting a path, a single thread must complete this without interruption. This means once a thread reaches the sink node, it locks the other threads from augmenting.

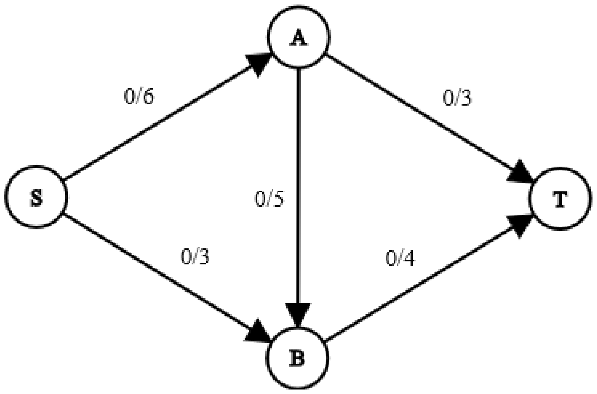

Fig. 1. Network Flow Graph

$$\begin{array}{c|cccc} & S & A & B & T \\ \hline S & 0 & 6 & 3 & 0 \\ A & 0 & 0 & 5 & 3 \\ B & 0 & 0 & 0 & 4 \\ T & 0 & 0 & 0 & 0 \end{array}$$

Fig. 2. Adjacency Matrix of Fig. 1's capacity

$$\begin{array}{c|cccc} & S & A & B & T \\ \hline S & 0 & 0 & 0 & 0 \\ A & 0 & 0 & 0 & 0 \\ B & 0 & 0 & 0 & 0 \\ T & 0 & 0 & 0 & 0 \end{array}$$

Fig. 3. Starting Adjacency Matrix of Fig. 1's flow

5) Tracing Through The Final Approach: Figure 1 and 2 will be used as an example to illustrate how the parallel approach works. There is also another adjacency matrix of the same size as the capacity matrix which keeps track of the flow after augmenting a graph. This flow matrix has a starting state where all the values are initialized to 0. The notation flow[0][1] means get the element on the 0th row and 1st column of the flow matrix. The notation cap[0][1] means get the element on the 0th row and 1st column of the capacity matrix (this would be 6 on Figure 2).

S is the Source node and T is the Sink node. Assume the use of 2 threads X and Y. The notation of X:cap[1][2] means thread X is looking at the capacity of 5. Begin all threads in the 0th-index row (S).

Initially, the threads will be at X:cap[0][1] and Y:cap[0][2] (Note that the Source node column (cap[_][0]) can be skipped since there are no incoming edges to the source node. So the threads can be X:cap[0][0] and Y:cap[0][1] but it is not done here in the example as it is not as helpful for demonstration nor is it efficient). The two threads will then sweep through the rows from top-to-bottom in order to label nodes.

A label will be in the form of a tuple (Ex. (A, +, 6)) which describe, respectively, what node it came from, what operation (add or subtract) to do when augmenting, and the potential flow of the labeled node. Some things to note is that the Source node is always labeled as (_, +, ∞); once the Sink node is labeled, start the augmentation of the graph by backtracking through the labels of the nodes and use the potential flow labeled on the Sink node when augmenting each of the edges.

We only label a node if it is not already labeled and:

1) (Forward Edge) The flow of the edge from previous node (prev) to current node (curr) is less than the capacity from prev to curr.
   For i, j, where they are the index of the prev node and curr node respectively, $flow[i][j] < cap[i][j]$. This would result in a labeling of (prev, +, min{prevPotentialFlow, cap[i][j] - flow[i][j]}). prevPotentialFlow is simply the third item of the tuple in the label of the previous node.
2) (Backward Edge) The flow of the edge from the curr node to the prev node is greater than 0.
   For i, j, where they are the index of the prev node and curr node respectively, $flow[j][i] > 0$. This would result in a labeling of (prev, -, min{prevPotentialFlow, flow[j][i]})

Using the rules above, for thread X currently at X:cap[0][1], X will label node A because the previous node S is labeled (with (_, +, ∞)) and $X : flow[0][1] < X : cap[0][1]$. With this, node A will be labeled with (S, +, 6). For thread Y currently at Y:cap[0][2], Y will label node B because the previous node S is labeled and $Y : flow[0][1] < Y : cap[0][2]$. Hence, node B will be labeled with (S, +, 3). The graph and matrix are shown below.

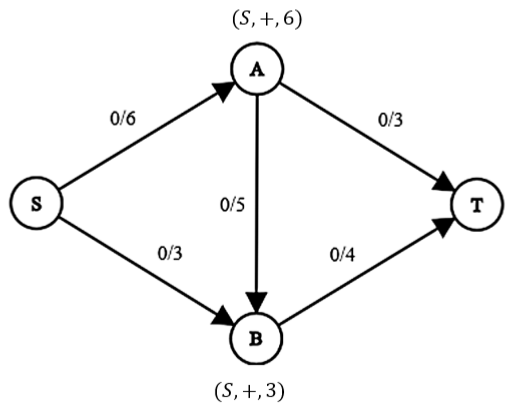

Fig. 4. Threads X and Y simultaneously labeling nodes A and B.

$$\begin{array}{c|cccc|}
 & S & A & B & T \\
S & 0 & \multicolumn{2}{|c|}{6 \quad\; 3} & 0 \\
A & 0 & 0 & 5 & 3 \\
B & 0 & 0 & 0 & 4 \\
T & 0 & 0 & 0 & 0
\end{array}$$

Fig. 5. X:cap[0][1] and Y:cap[0][2] on an initial sweep

After the two threads in this example finished labeling in the 0th-index row, both threads move to the 1st-index row in the same respective column. So X:cap[1][1] and Y:cap[1][2] are the next elements the threads will attempt to label. It will be found that X any Y will not label A and B since A and B was already labeled in the previous iteration. Hence, the graph in this iteration will look the same as Figure 4.

$$\begin{array}{c|cccc|}
 & S & A & B & T \\
S & 0 & 6 & 3 & 0 \\
A & 0 & \multicolumn{2}{|c|}{0 \quad\; 5} & 3 \\
B & 0 & 0 & 0 & 4 \\
T & 0 & 0 & 0 & 0
\end{array}$$

Fig. 6. X:cap[1][1] and Y:cap[1][2] on next sweep

The threads will continue to sweep down. When it reaches the last row, the threads shift over right 1 column and goes back to the 0th-index to do another top-to-bottom sweep. If the threads shift over past the number of columns, it cycles back to the 0th-index column and continues.

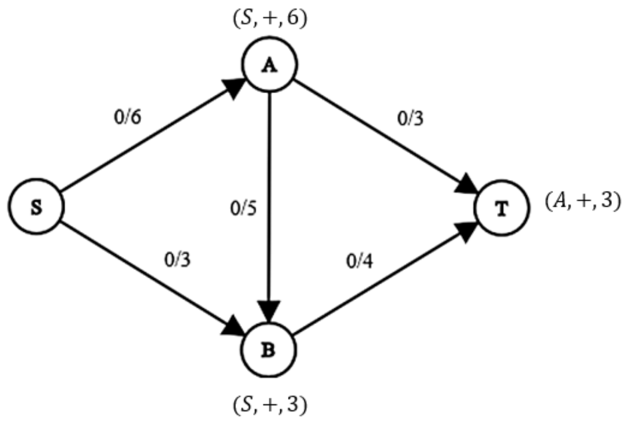

Fig. 7. Labeling sink

$$\begin{array}{c|cccc|}
 & S & A & B & T \\
S & 0 & 6 & 3 & 0 \\
A & 0 & 0 & \multicolumn{2}{|c|}{5 \quad\; 3} \\
B & 0 & 0 & 0 & 4 \\
T & 0 & 0 & 0 & 0
\end{array}$$

Fig. 8. X:cap[1][1] and Y:cap[1][2] on sweep

After some amount of sweeps, the threads go to the 1st-index row (A) with X:cap[1][2] and Y:cap[1][3]. Y:cap[1][3] will label the Sink node with (A, +, 3). Since Sink is labeled, augment the graph by backtracking through the labels and use the potential flow of the Sink node label, which is 3, on all the edges. Add or subtract that potential flow based on the 2nd item in the tuple of the label that is currently being scanned. The augmenting ends once the Source node is reached. This was the first augmentation of many that will occur in the graph. Below is the first augmented graph of many.

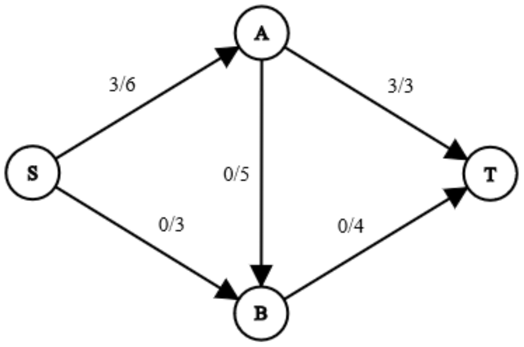

Fig. 9. The graph after augmenting

The algorithm continues augmenting the graph until an iteration cannot label any nodes after a top-to-bottom sweep. When the algorithm ends, the maximum flow is found by finding the sum of the flow column of the sink node.

$$\begin{array}{c|cccc} & S & A & B & T \\ \hline S & 0 & 4 & 3 & 0 \\ A & 0 & 0 & 1 & 3 \\ B & 0 & 0 & 0 & 4 \\ T & 0 & 0 & 0 & 0 \end{array}$$

Fig. 10. Adjacency Matrix of flow after finished algorithm

Hence, the max flow of the graph in this example is 7.

6) *Pseudo Code of Parallel:* Note if there are n nodes, each node will be indexed from 0 through $n-1$. In the algorithm, the Source node is indexed as 0 while the Sink node is indexed as $n-1$. See Algorithm 2. See Algorithm 1 for the Augment and Label functions.

C. Testing & Result

These test results were performed on UCF's Eustis3 server utilizing open-source datasets [5]

Nodes	Max Flow	Sequential Time	Parallel Time
50	828	5ms	
100	1251	10ms	
500	7143	182ms	
750	10930	396ms	
1000	13476	866ms	
2000	25027	6532ms	
3000	39170	18448ms	
4000	58517	53893ms	
5000	69349	117045ms	
10000	142610	784219ms	

Algorithm 2 Ford-Fulkerson Pseudo Code

```
Node:
$prevNode \leftarrow$ Integer
$thisNode \leftarrow$ Integer
$add \leftarrow$ Boolean
$potentialFlow \leftarrow$ Integer

FordFulkerson:
$row \leftarrow$ AtomicInteger
$capacity[][] \leftarrow$ Capacity Graph
$flow[][] \leftarrow$ Flow Graph
$nodeLabels[] \leftarrow$ Array of Nodes
$numNodes \leftarrow$ number of nodes in graph
$threadLimit \leftarrow$ Max number of threads
$tNum \leftarrow$ Identifying Thread number 0-indexed

for offset in range numNodes: do
    i $\leftarrow$ ((offset * threadLimit) + tNum) % numNodes
    for $row$ in range numNodes: do
        j $\leftarrow$ $row$
        if jth-node is labeled then

            if ith-node is not labeled and $flow[j][i] < cap[j][i]$ then Label() where $add$ is true
            end if
            if ith-node is not labeled and $flow[j][i] > 0$ then Label() where $add$ is false
            end if
        end if
waitForAllThreadsBeforeProceeding()
    end for
    if sink node is labeled then Only one thread Augment()
waitForAllThreadsBeforeProceeding()
do outermost for-loop again
    end if

end for
```

III. Edmonds-Karp

In 1972 Edmonds and Karp published a paper detailing a specific protocol using the Ford-Fulkerson method for computing max flow and min cut. The Edmonds-Karp (EK) algorithm is one of the earliest solutions for solving the maximum flow in a network of nodes that can be represented as a graph. Unlike its predecessor, it has a well defined specification on how to perform searches for augmenting paths. While Ford-Fulkerson may use any search method, often DFS, EK requires the use of BFS to find augmenting paths which improves theoretical runtimes from $\mathcal{O}(E|f|)$, where f is the max flow of the graph, to $\mathcal{O}(V^2E)$.

A. Sequential Algorithm

In order to test our parallel algorithm on correctness and speed, we implemented the EK algorithm in Java by editing an existing version of the algorithm written by one of our fellow classmates, Pedro Contepelli, who has given us full permission to use and edit his code for our research as needed. The algorithm uses three different classes to maintain a graph (*Graph*), its edges (*Edge*), and its nodes (*Node*).

A *Graph* is instantiated from a text based input file holding the matrix representing the graph and the number of *Node*s it has. The source and the sink are assumed to be 0 and $n-1$ respectively. The *Graph* also has a method *getAugPath()* which returns an augmenting path in the graph.

From there *main()* runs the EK algorithm to compute the max flow for the graph. It does this by finding the maximum flow that can be pushed through the obtained path and then adds each edge's flow to the forward flow of the edge and subtracts it from the reverse flow of the edge. It should be noted that this algorithm is also useful in solving the min-cut problem as well as the max flow problem because they are both the same problem [Ford-Fulkerson].

B. Parallel Algorithm

We believe there are two distinct ways to transform the sequential Edmonds-Karp algorithm into a version that applies some concurrency to the algorithm. The first is strictly through parallelization of the BFS algorithm to decrease the augmenting paths search time, and the second is by using fine-grained locking techniques to build augmenting paths in parallel using several threads running BFS concurrently on a single graph data structure. We will discuss each approach and explain why we chose attempt to build a concurrent graph using Java's parallel libraries.

1) Parallel Breadth-First-Search: Implementing a parallel BFS was the first thought that came to mind when attempting to create a parallel implementation of the EK algorithm. BFS is an incredibly time intensive task running in $\mathcal{O}(V+E)$ time while the rest of the algorithm simply adds a multiple of the BFS run time because calculations are performed on the path BFS finds. If we could speedup the search rate for an augmenting path then we could potentially see impressive gains in speedup for the overall EK algorithm. Taking this approach would prove difficult, however, as other research says that BFS is not easily parallelizable which lead us to parallelization [6].

2) Parallel Augmenting Paths: It is possible to allow multiple threads to try to update the graph based on the augmenting path they have found. There are two possible ways to handle these graph updates in a concurrent environment. We could use coarse-grained locks on the graph data structure while the flows and capacities are being updated to ensure mutual exclusion. This approach could produce significantly less speedup than desired due to the locks potentially forcing the program into a sequential state. Another option would be to create a fine-grained locking system that only locks the current augmented path edges while they are being updated.

Using a coarse-grained lock would allow BFS computations for augmenting paths to be done in parallel while locking the calculations necessary to update the flows and capacities of each of the edges in the augmenting path. Unfortunately, this method would not be particularly useful because each augmenting path found in parallel would then be discarded when one of the threads successfully updates the graph. This parallel version would be significantly easier to write than implementing a parallel BFS, but could lead to the algorithm being nearly sequential as mentioned earlier so we will try another method.

Applying fine-grained locking techniques to the *Graph* will allow it to become a more versatile concurrent data structure. Instead of refusing access to the entire *Graph* we can simply lock the parts of the graph that we need to perform calculations on, namely the path generated by BFS, while other threads are free to perform work on parts of the *Graph* that have not yet been locked.

3) Fine-Grained Concurrent Graph: To build a fine-grained concurrent data structure we have to take careful consideration in how we implement each field and method associated with the object. For the parallel EK algorithm we will need to create a shared memory *Graph* that is capable of being searched by several threads and then having updates to the residual capacities and flows of those paths applied in parallel. To achieve this we will lock each edge of the augmenting path as it is being added to our augmenting path variable to ensure no other threads try to make residual capacity or flow updates to the edges in the current thread's augmenting path. In short, we need to add a lock to *Edge* and require threads to acquire this *Edge*'s lock before continuing. We define the specifications we think are required to create a concurrent *Graph* in pseudo code in the Algorithm 3

As described, the function locks the edges of the path as it builds it so that other threads cannot try to access the same $ConcurrentEdge$ at the same time. These classes are very specific to the EK algorithm and require the run method to be run to for each of these class methods to work properly, specifically the $getAugmentingPath$ method relies on the $edmondsKarp$ method from Algorithm 4 to unlock each $ConcurrentEdge$ in the path before starting another BFS.

C. Testing & Results

Preliminary results show that the parallel algorithm runs in approximately the same time as the sequential algorithm, but formal testing is still in progress.

Algorithm 3 Concurrent Classes

```
class $ConcurrentGraph$

function GETAUGMENTINGPATH
    $path$ ← array of $Edges$
    $queue$ ← empty queue of $Nodes$
    add source node to $queue$
    while $queue$ is not empty do
        remove node from $queue$
        for each edge in the current node do
            if meets augmenting path requirements then
                try to acquire edge lock
                if successful then
                    add edge to augmenting path
                    add node connected to edge to $queue$
                else
                    add removed node back to queue
                end if
            end if
        end for
    end while
end function


class $ConcurrentNode$
$edges$ ← array of edges

class $ConcurrentEdge$
$u$ ← integer
$v$ ← integer
$flow$ ← integer
$capacity$ ← integer
$reverse$ ← $Edge$
$lock$ ← reentrant lock
function LOCK
    do $lock$.lock
end function
function UNLOCK
    do $lock$.unlock
end function
```

Algorithm 4 Edmonds-Karp Parallel Implementation

```
function EDMONDSKARP
    $graph$ ← constructor creates graph from file
    $maxFlow$ ← integer
    while true do
        $path$ ← $getAugmentingPath$
        if $path$ does not contain the sink then return
        else
            $forwardFlow$ ← $INFINITY$
            for each $ConcurrentEdge$ in $path$ do
                $forwardFlow$ ← minimum residual flow
            end for
            for each $ConcurrentEdge$ in $path$ do
                add $forwardFlow$ to $flow$
                subtract $forwardFlow$ from $reverseFlow$
            end for
            $maxFlow$ ← $maxFlow + pushFlow$
            for each $ConcurrentEdge$ in $path$ do
                do $ConcurrentEdge.lock$.unlock
            end for
        end if
    end while
end function
```

IV. Dinics

A. Sequential Algorithm

1) Introduction: Dinic's algorithm is a flow algorithm that was created by Dr. Yefim Dinitz, and is notable for being strongly polynomial with an $\mathcal{O}(V^2E)$ run-time. Dinic's algorithm innovates on other flow algorithms with its use of level graphs and blocking flows. Dinic's algorithm uses BFS and DFS in order to achieve this superior run-time; bread-first search for building a level graph, and depth-first search with blocking flows in order to find the maximum flow efficiently.

2) Algorithm Overview: In what follows, we are going to walk the through the algorithm, step by step, with examples from our own implementation. *To provide a brief overview of the original Dinic's algorithm, the pseudo-code has been provided below.*

Input: A network G = ((V, E), c, s, t).
Output: An s-t flow f of maximum value.

1) Set $f(e) = 0$ for each $e \in E$.
2) Construct G_L from G_f of G. If $dist(t) = \infty$, stop and output f.
3) The text in the entries may be of any length.

The algorithm begins with the assumption that we want to make positive progress from the source node towards the sink node. We achieve this goal by using a level graph (Figure 11),

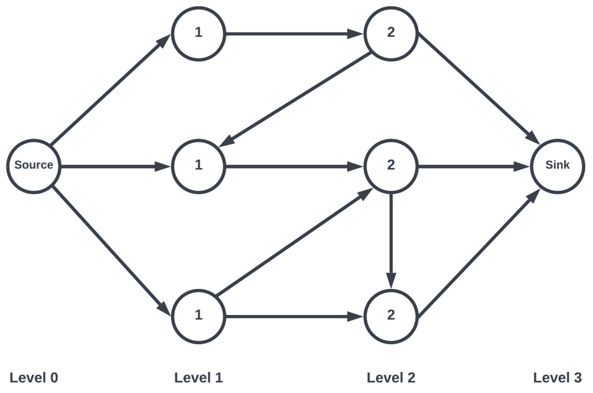

Fig. 11. Network Flow Graph

which keeps track of whether the edges are making progress towards the sink or not. This level graph is built using BFS, since with BFS we can easily calculate the minimum distance, or level, from the source node to other nodes. BFS achieves this by looking at the directed edges that point toward neighboring nodes from the current node, and if a level has not been set for those nodes, a level will be assigned. The neighboring nodes will get added to a queue for the next iteration of the BFS, where the same steps will take place, except this time, we will be setting the next set of neighboring nodes to level + 1. [7] *See pseudo-code below for a more detailed overview of how the Dinic's BFS would create a level graph:*

Once the level graph is completed using BFS, we are able to see if we're progressing forward by looking at whether the node level increases as we traverse the graph. Then we perform

Algorithm 5 BFS Pseudo Code

```
Node:
$level$ ← Integer
$edges$ ← Array of nodes
$capacityRemaining$ ← Integer
function BFS
    $queue$ ← empty queue of nodes
    $queue$ ← queue + source node
    $level$ ← 1
    while queue is not empty do
        for for i in range 0, queue.size: do
            $currentNode$ ← queue.poll
            $neighbors$ ←findNeighbors(currentNode.edges)
            for neighbor in neighbors do
                $neighbor.level$ ← level
                $queue$ ← queue + neighbor
            end for
            $level$ ← level + 1
        end for
    end while
end function

function FINDNEIGHBORS(edges)
    $neighbors$ ← empty list of nodes
    for neighbor in edges do
        if $neighbor.capacityRemaining > 0$ and level is not 0 then
            $neighbors$ ← neighbors + neighbor
        end if
    end for
    return neighbor
end function
```

a DFS that looks for the paths from the source node to the sink node, filling up the capacity of the edges until we reach a "blocking flow". When we reach that point, we compute our max flow. We then repeat these steps until we reach a point where we can no longer keep going. [1]

B. Parallel Algorithm

1) Research: First, we decided to tackle parallelizing the BFS portion of Dinic's algorithm. A parallel BFS will allow for a concurrent construction of the level graph used in the algorithm [4]. To do this, we researched various ways in which standard BFS is parallelized; we noticed two possible approaches:

1) Using shared memory with a FIFO queue data structure.
2) Using shared memory with a bag data structure.

2) Final Approach: Since insertion and deletion from a queue both take $\mathcal{O}(1)$ time, while bags have an $\mathcal{O}(Log(N))$ worst case run time, we decided to implement the queue approach. We chose to spawn threads, or parallelize the task, at the point in which we enter the while loop in the BFS, which loops until the atomic queue is not empty. For each node that becomes available in the queue, a waiting thread will take that node, and compute the neighbors while adding them to the queue for the next available thread to process. *See pseudo-code below for a more detailed overview:*

Algorithm 6 Parallel BFS Pseudo Code

```
 1: $runIndex(Atomic) \leftarrow 0$
 2: function Run
 3:     $firstEntry \leftarrow$ true
 4:     while firstEntry or runIndex is 0 do
 5:         $firstEntry \leftarrow$ false
 6:         try:
 7:         $runIndex \leftarrow$ runIndex + 1
 8:         while queue size is greater than 0 do
 9:             $node \leftarrow$ q.poll
10:             for each edge in graph[node]: do
11:                 $capacity \leftarrow$ edge.remainingCapacity
12:                 if capacity is greater than 0 and level[edge]
    == -1 then
13:                     $level[edge] \leftarrow$ level[node] + 1
14:                     q.offer(edge)
15:                 end if
16:             end for
17:         end while
18:         finally:
19:         $runIndex \leftarrow$ runIndex - 1
20:     end while
21: end function
```

C. *Testing & Results*

Sequential vs. parallel Dinic's results coming soon..

V. Conclusion

Network flow is pretty cool. By parallelizing them we advanced human civilization to a new frontier, if you will, of human knowledge. We have now reached the point of singularity and we are infinite beings. With this paper, this is what we now offer to society, to humans, and to the universe. (Conclusion still pending)

VI. Appendix

A. Challenges

Some of the biggest challenges we have faced in this research process was understanding the relevant literature with regards to each of our algorithms.

B. Completed Tasks

- Ford Fulkerson
  - Sequential and Parallelizable Version of FF
  - Algorithm Sketch of Parallel FF
  - Potential Optimizations of Parallel FF researched
  - Successful Testing of Sequential FF with Test-Cases
- Edmund Karp
  - Sequential algorithm code
  - Parallel algorithm code
- Dinics

C. Remaining Tasks

- Add Ford-Fulkerson, Edmonds-Karp, Dinics papers' references

References

[1] William Fiset. *Max Flow Dinic's Algorithm — Network Flow — Graph Theory*. Youtube. 2018. URL: https://www.youtube.com/watch?v=M6cm8UeeziI&t=505s.

[2] William Fiset. *Max Flow Ford Fulkerson — Network Flow — Graph Theory*. Youtube. 2018. URL: https://www.youtube.com/watch?v=LdOnanfc5TM.

[3] *Maximum flow - Ford-fulkerson and Edmonds-Karp*. URL: https://cp-algorithms.com/graph/edmonds_karp.html.

[4] *Parallel Breadth First Search*. URL: https://en.wikipedia.org/wiki/Parallel_breadth-first_search.

[5] SumitPadhiyar. *Sumitpadhiyar/PARALLEL*$_FORD_FULKERSON_G$ $ParallelizationofFordFulkersonalgorithmonGPU$. URL: https : / / github . com / SumitPadhiyar / parallel_ford_fulkerson_gpu.

[6] Vibhav Vineet and P. J. Narayanan. "CUDA cuts: Fast graph cuts on the GPU". In: *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. 2008, pp. 1–8. DOI: 10.1109/CVPRW.2008.4563095.

[7] WilliamFiset. *WilliamFiset/Dinic's Algorithm*. URL: https://github.com/williamfiset/Algorithms/tree/master/src/main/java/com/williamfiset/algorithms/graphtheory/networkflow/examples.