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 including roads, pipes, electrical systems, even fields 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 paper, we explore three algorithms and we parallelize them to increase 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

Network Flow is a topic in graph theory that addresses the flow within a specific variant of a directed graph. The graph always has a Source node where all the flow originates and a Sink node where flow attempts to travel to. Flow is transferred from one node to the next through edges connecting them. Each edge has a capacity stating maximum amount of flow that can go through that edge; once that edge reaches its capacity, no more flow can go through it. A popular usage with Network Flow is finding the max flow, the maximum potential amount of flow from the Source to the Sink, for any graph as quickly and efficiently as possible.

Research pertaining to the max flow solution of network flow problems traces back to L.R. Ford and D. R. 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, Jack Edmonds and Richard Karp proposed a modification to the original Ford-Fulkerson algorithm which helped stabilize the run time and guarantee completion. Around the same time, Yefim Dinitz, proposed his own solution to max flow problem. Each of these algorithms use 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 that computes 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]

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 [4] 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

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 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:

1) Search for a valid flow-augmenting path
2) Update the flow-augmenting paths on a backward pass

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. For most of the development for this endeavor, the Ford-Fulkerson team had insufficient experience for this.

Rather than navigating through the difficulties of directly turning DFS parallel, a hybrid solution was attempted; 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.

1) 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.

In the Initial Approach, threads would sweep from left-to-right in the capacity matrix column-by-column. Unfortunately,

Algorithm 1 Ford-Fulkerson Pseudo Code

```
1: function AUGMENT
2:     tN ← sinknode
3:     augNum ← Integer
4:     while tN is not 0 do
5:         if tN.thisNode then
6:             flow[tN.prevNode][tN.thisNode] ← flow[tN.prevNode][tN.thisNode] + augNum
7:         else
8:             flow[tN.thisNode][tN.prevNode] ← flow[tN.thisNode][tN.prevNode] - augNum
9:         end if
10:        tN ← tN's previous node
11:    end while
12: end function
13:
14:
15: function LABEL
16:    if add : then
17:        pFlow ← min(ith-node's potentialFlow, cap[i][j] - flow[i][j])
18:    else
19:        pFlow ← min(ith-node's potentialFlow, flow[j][i])
20:    end if
21: end function
1: Node:
2: prevNode ← Integer
3: thisNode ← Integer
4: add ← Boolean
5: potentialFlow ← Integer
6:
7: FordFulkerson:
8: capacity[][] ← Capacity Graph
9: flow[][] ← Flow Graph
10: nodeLabels ← Array of Nodes
11: numNodes ← Number of nodes in graph
12:
13: sink ← Integer
14:
15: for i in range 2: do
16:    for i in range numNodes: do
17:        for j in range numNodes: do
18:            if jth-node is labeled then
19:
20:                if ith-node is not labeled and flow[j][i] < cap[j][i] then Label() where add is true
21:                end if
22:                if ith-node is not labeled and flow[j][i] > 0 then Label() where add is false
23:                end if
24:            end if
25:        end for
26:        if sink node is labeled then Augment() then do outermost for-loop again
27:        end if
28:    end for
29: end for
30:
```

this particular method of labeling the nodes raises contention issues in the form of multiple threads attempting to acquire a lock. For example, in Figure 1 for the process of labeling the flow from node A to B and S to B causes contention of the labeling of B.

2) 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-column. Utilizing a CyclicBarrier, we implement a wait-free correctness policy as the results of the method calls depend on every single thread successfully reaching that particular bottleneck.

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

The work of each thread is load-balanced by having each thread start on a column that are evenly spaced from each other (i.e. if you have 2 threads and a 4x4 capacity, one thread will be on column 0, the other thread will be on column 2.) In other words, this means we can forego the usage of locks, as the threads remain in their respective lanes while traversing the graph for augmenting paths, only stopping to wait for the other threads to get to the CyclicBarrier.

The method of labeling was discovered independently of existing research papers, although not completely novel [3], while tracing through simulations of multiple threads augmenting the graph. An important observation was that a single thread must complete this without interruption: once a thread reaches the sink node, it blocks the other threads from augmenting by forcing them to wait at the CyclicBarrier.

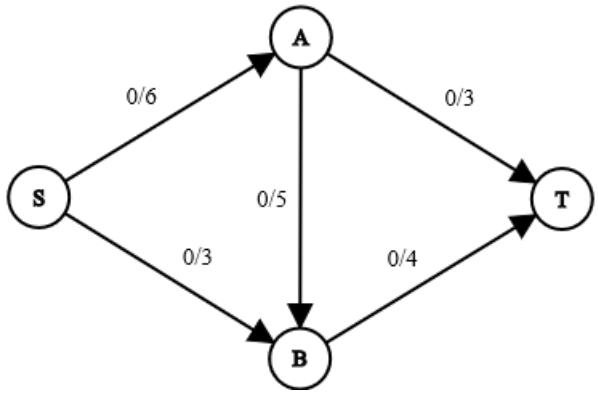


Fig. 1. Network Flow Graph

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

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

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

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

3) 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 $flow[0][1]$ denotes the element on the 0th row and 1st column of the flow matrix. The notation $cap[0][1]$ denotes 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). In the actual procedure of the code, the threads would be load-balanced where X is on column 0 and Y is on column 2, but for the sake of quick demonstration, the threads will be on column 1 and 2, respectively.

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 respectfully describes:

1) What node it came from,
2) What operation (add or subtract) to do when augmenting
3) The potential flow of the labeled node

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.

A node is only labeled if it is unlabeled in addition to:

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, the indices 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, the indices 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:

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

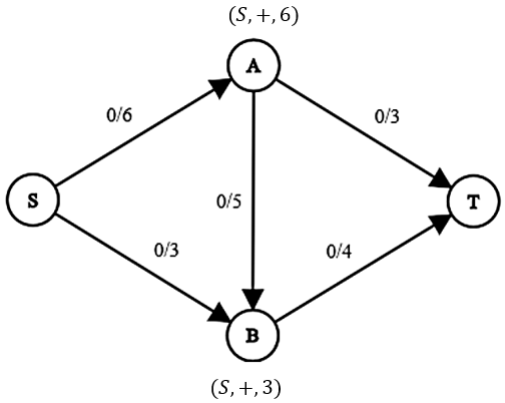


Fig. 4. Threads X and Y simultaneously labeling nodes A and B.

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

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

After the two threads in this example finish labeling in the 0th-index row, both threads move to the 1st-index row in the same respective column. $X : cap[1][1]$ 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.

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

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

The threads will continue to sweep down. At the last row, the threads shift over right 1 column and back to the 0th-index for another top-to-bottom sweep. If the threads shift over past

the number of columns, it wraps around back to the 0th-index column and continues.

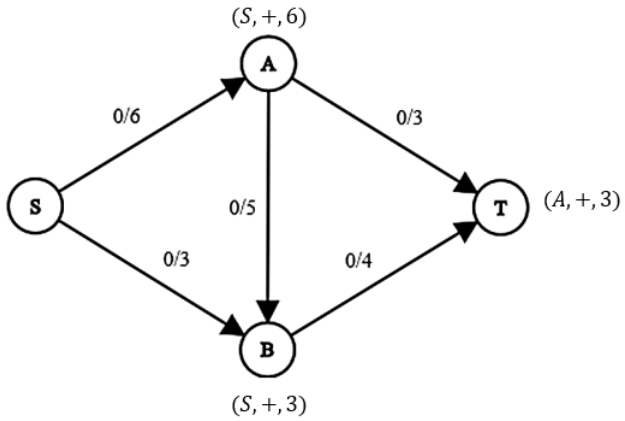


Fig. 7. Labeling sink

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

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

After an eventual 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) once eventually reached. Since the sink is labeled, that thread will backtrack through the graph utilizing the labels as breadcrumbs. Potential flow is added or subtracted 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.

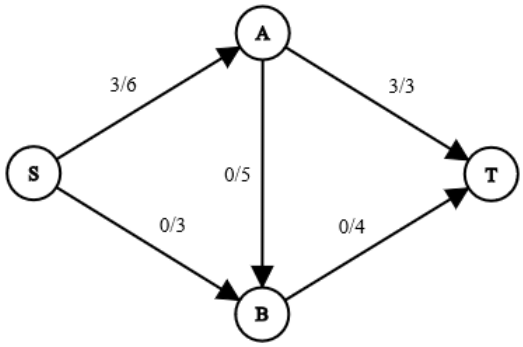


Fig. 9. The graph after augmenting

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.

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

Fig. 10. Adjacency Matrix of flow after finished algorithm

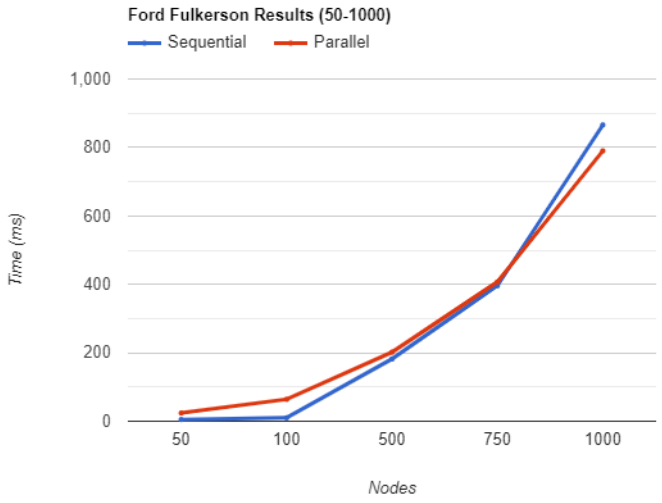
Hence, the max flow of the graph in this example is 7.

4) Pseudo Code of Parallel: Note if there are n nodes, each node will be indexed from 0 through $n-1$. 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 & Results

These test results were performed on UCF's Eustis3 server utilizing open-source data sets [6]. The results on the table are an average of 3 runs.

Nodes	Max Flow	Sequential Time	Parallel Time
50	828	5ms	24ms
100	1251	10ms	64ms
500	7143	182ms	202ms
750	10930	396ms	407ms
1000	13476	866ms	790ms
2000	25027	6532ms	3848ms
3000	39170	18448ms	12407ms
4000	58517	53893ms	32107ms
5000	69349	117045ms	52792ms
10000	142610	784219ms	422210ms



Algorithm 2 Ford-Fulkerson Parallel Pseudo Code

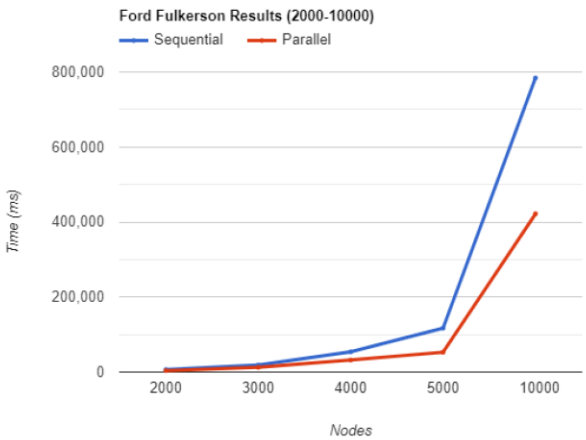
```
Node:
prevNode ← Integer
thisNode ← Integer
add ← Boolean
potentialFlow ← Integer

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

for offset in range numNodes: do
    col ← ((offset * threadLimit) + tNum) % numNodes
    for row in range numNodes: do
        j ← row
        if jth-node is labeled then

            if node at this col is not labeled and flow[row][col] < cap[col][row] then Label() where add is true
            end if
            if node at this col is not labeled and flow[col][row] > 0 then Label() where add is false
            end if
        end if
waitForAllThreadsBeforeProceeding()
    end for
    if sink node is labeled then
barrier.wait() // Have threads wait until all arrive
Only one thread Augment()
barrier.wait() // Have threads wait until all arrive
do outermost for-loop again
    end if

end for
```



The testing results only begin to show noticeable changes once the node count begins to approach the 500 mark. The parallel implementation of Ford-Fulkerson begins to experience an exponential growth in speed after reaching that threshold, with the 10000 node testcase gaining a 85.74% increase in speed from the sequential implementation, which is 1.857 times faster. By Amdahl's Law, it can be determined that the parallel program implementation of Ford-Fulkerson is 52.76% concurrent.

While a parallel implementation of Ford-Fulkerson is viable, from research, practicality it costs more resources than is worth for graph sizes most smaller applications would deal with; from a logical standpoint, creating a parallel Ford-Fulkerson implementation should be more applied to larger use-cases that depend on massive graph sizes.

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 [7].

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 Initial Approach: 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.

4) Fine-Grained Concurrent Graph Final Approach: Through some preliminary testing it was clear that the initial approach had some efficiency short comings. After analyzing the algorithm, it is clear that there is a path bottleneck, especially for smaller graphs. In larger graphs, as each thread builds a path the only thing that will cause delay is colliding with another path. The current algorithm is naive and waits for its desired path to become available which undoubtedly increases the run time. This can be mended by requeuing the locked edge and moving on to the next available edge in the queue so as not to delay progress and to also not miss out on any possible paths. It is possible that all edges that a thread wants to acquire are locked, but this scenario is far less likely to happen so significant speedup is expected. This requires a change to the $getAugmentingPaths$ function in *Concurrent Classes* which can be observed in the pseudocode Algorithm 5. There are no necessary changes to the overall Edmonds-Karp algorithm described in Algorithm 4.

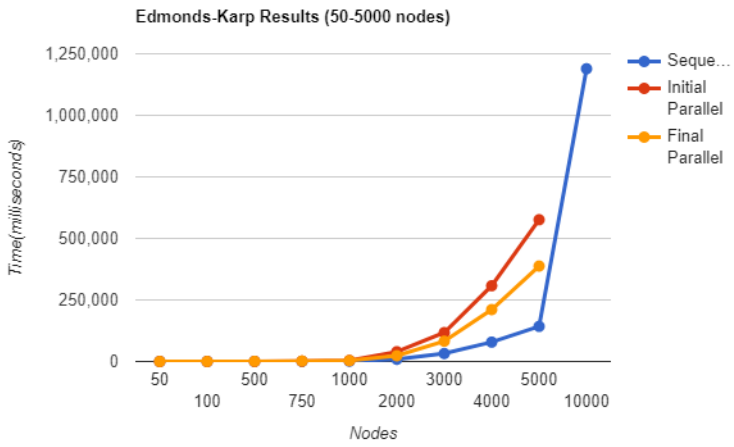
C. Testing & Results

These test results were performed on UCF's Eustis3 server utilizing open-source data sets [6] running on eight threads. The results on the table are an average of 3 runs.

It can be seen from the table the the initial parallel approach performed worse than its sequential counterpart. For the sake of time, the ten thousand node test was foregone for the initial approach. After some analysis of the algorithm it became clear that, while having the correct solution, it would be much slower than the sequential version due to the overhead of running eight threads. This was the inspiration for finding the final approach to the parallelization of the Edmonds-Karp algorithm.

The final approach algorithm performed better than the initial approach overall as expected. Interestingly, for graphs with less than 1000 nodes, the final approach algorithm was slower than the initial approach. This could be because of the sparsity of the test graph, but is undetermined.

Nodes	Max Flow	Sequential Time	Initial Parallel Time	Final Parallel Time
50	828	3ms	11ms	36 ms
100	1251	11ms	22ms	79ms
500	7143	123ms	403ms	604ms
750	10930	253ms	1456ms	1656ms
1000	13476	1227ms	3968ms	3467ms
2000	25027	8837ms	39514ms	23382ms
3000	39170	32302ms	117157ms	82435ms
4000	58517	78587ms	307633ms	210706ms
5000	69349	142469ms	575808ms	387658ms
10000	142610	1189164ms	N/A	N/A



The final 10,000 node test could not be completed due to not enough heap space. Unfortunately, neither attempt at parallelization was able to improve the run time when compared with the sequential algorithm. This may be due to the structure of the graph being an adjacency list rather than an adjacency matrix. Future research might consider this idea.

Algorithm 3 Initial Concurrent Classes

```
1:  class ConcurrentGraph
2:
3:  function getAugmentingPath
4:      path ← array of Edges
5:      queue ← empty queue of Nodes
6:      add source node to queue
7:      while queue is not empty do
8:          remove node from queue
9:          for each edge in the current node do
10:             if meets augmenting path requirements then
11:                 try to acquire edge lock
12:                 add edge to augmenting path
13:                 add node connected to edge to queue
14:             end if
15:         end for
16:     end while
17: end function
18:
19: class ConcurrentEdge
20: u ← integer
21: v ← integer
22: flow ← integer
23: capacity ← integer
24: reverse ← Edge
25: lock ← reentrant lock
26: function LOCK
27:     do lock.lock
28: end function
29: function UNLOCK
30:     do lock.unlock
31: end function
```

Algorithm 4 Edmonds-Karp Parallel Implementation

```
1:  function edmondsKarp
2:      graph ← constructor creates graph from file
3:      maxFlow ← integer
4:      while true do
5:          path ← getAugmentingPath
6:          if path does not contain the sink then return
7:          else
8:              forwardFlow ← INFINITY
9:              for each ConcurrentEdge in path do
10:                 forwardFlow ← minimum residual flow
11:             end for
12:             for each ConcurrentEdge in path do
13:                 add forwardFlow to flow
14:                 subtract forwardFlow from reverseFlow
15:             end for
16:             maxFlow ← maxFlow + pushFlow
17:             for each ConcurrentEdge in path do
18:                 do ConcurrentEdge.lock.unlock
19:             end for
20:         end if
21:     end while
22: end function
```

Algorithm 5 Final Concurrent Classes

```
1:  function getAugmentingPath
2:      path ← array of Edges
3:      queue ← empty queue of Nodes
4:      add source node to queue
5:      while queue is not empty do
6:          remove node from queue
7:          edgeQ ← queue of Edges
8:          for each edge in the current node do
9:              if meets augmenting path requirements then
10:                 try to acquire edge lock
11:                 if successful then
12:                     add edge to augmenting path
13:                     add node connected to edge to queue
14:                 else
15:                     add edge to edgeQ
16:                 end if
17:             end if
18:         end for
19:         while edgeQ is not empty do
20:             Edgee ← edgeQ.poll
21:             if meets augmenting path requirements then
22:                 try to acquire edge lock
23:                 if successful then
24:                     add edge to augmenting path
25:                     add node connected to edge to queue
26:                 else
27:                     add edge to edgeQ
28:                 end if
29:             end if
30:         end while
31:     end while
32: 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),

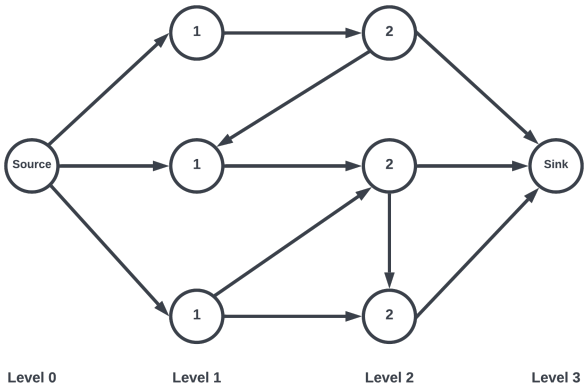


Fig. 11. Network Flow Graph

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. [8] *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 6 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 [5]. 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.

After extensive research, a few inherent issues with implementing a concurrent depth first search algorithm were discovered. The main issue being that depth first traversals follow sequential paths through a graph, going as deep as possible, before checking the neighboring nodes. This is perfect for augmenting flow in the Dinics algorithm, but poses a problem for our parallel implementation as threads may cross paths. To handle this, we decided to lock intersecting nodes, while

waiting for a thread to finish sending blocking flow along a path, before unlocking the intersecting node to allow the previous to continue its traversal.

2) Final Approach: BFS: 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 7 Parallel BFS Pseudo Code

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

DFS: To implement our parallel DFS, we needed to handle the case when two threads encounter an intersecting node. This is due to the fact that if two threads have overlapping paths, that thread could be traveling down an invalid path, and in turn over-calculate the max flow. When two threads encounter the same node, one thread must wait until the other thread finishes sending blocking flow down its path. Once the thread has finished augmenting the edges, it can release the lock, allowing the waiting thread to continue. When the waiting thread begins to traverse the graph, it first checks to ensure there is still capacity along its edges as a safety guard against over-calculating the maximum flow.

The current implementation needs more research and development to ensure correctness as the graph grows in size, as well as how they perform on various platforms with varying number of threads. On certain platforms, with extremely large graphs, we observed unexpected results on 2 rare occasions. We currently hypothesize that this is due to traffic at intersecting nodes, or threads not traversing valid paths due to the current checking condition for a given path, both producing a max flow value that is slightly under calculated. These occurrences are rare, and only seen with 16 threads on an Apple M1 processor. Our results table was produced observing only correct output using 8 threads on UCF's eustis3 server. *See pseudo-code below for a more detailed overview:*

Algorithm 8 Parallel DFS Pseudo Code

```
1: function RUN
2:     for long f=dfs(source, next, INF); f!=o; f=dfs(source,
   next, INF) do
3:         maxFlow += f
4:     end for
5: end function
6:
7: function DFS
8:     numEdges ← Integer
9:     edge ← Edge
10:    cap ← Long
11:    if at == t then
12:        flow
13:    end if
14:    numEdges ← graph[at].size()
15:    for ; next[at]¡numEdges; next[at]++ do
16:        if cap > 0 && level[edge.to] == level[at] + 1 then
17:            this.lock(edge.from)
18:            try:
19:            cap ← edge.remainingCapacity()
20:            if cap > 0 && level[edge.to] == level[at] + 1
   then
21:                bottleneck ← dfs(edge.to, next, min(flow,
   cap))
22:                if bottleneck > 0 then
23:                    edge.augment(bottleneck)
24:                    return bottleneck
25:                end if
26:            end if
27:            catch(Exception e)
28:            e.printStackTrace()
29:            finally:
30:            this.unlock(edge.from)
31:        end if
32:    end for
33:    return 0
34: end function
```

C. Dinics Conclusion

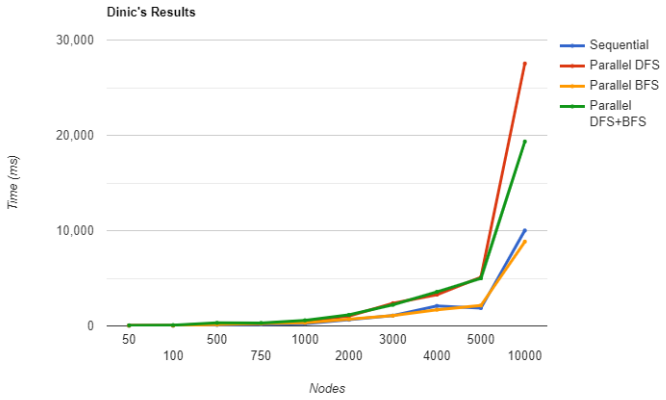
The Dinics algorithm proved to be, by far, the fastest sequential algorithm out of the three we selected for this research paper, easily outperforming both Ford-Fulkerson and Edmonds-Karp. But due to the overhead with creating threads, as well as contention with threads fighting for progress at locked intersecting nodes in the DFS, we observed some speed-ups with a parallel BFS, but occasional slow-downs when using a parallel depth first search. We plan to look into further optimizations as we attempt to achieve further gains in efficiency, but this provides a solid foundation for

further improvements in the area of parallelized network flow algorithms.

D. Testing & Results

These test results were performed on UCF's Eustis3 server utilizing open-source data sets.
[6]

The testing results show that the parallel DFS slows down the program compared to sequential with a 63.62% decrease in speed when looking at the test case with 10,000 nodes. The parallel DFS+BFS slows down the program compared to sequential with a 48.28% decrease in speed with the same test case. Meanwhile, the parallel BFS experienced a speedup of 13.21% using that same test case (1.132 times faster). By Amdahl's Law, it can be determined that the parallel BFS program implementation of Dinics is 13.32% concurrent.



Nodes	Max Flow	Sequential Time	Parallel DFS Time	Parallel BFS Time	Parallel DFS+BFS Time
50	828	22ms	54ms	26ms	59ms
100	1251	30ms	53ms	33ms	86ms
500	7143	104ms	220ms	144ms	318ms
750	10930	164ms	242ms	242ms	315ms
1000	13476	232ms	495ms	320ms	591ms
2000	25027	650ms	1035ms	698ms	1155ms
3000	39170	1083ms	2379ms	1074ms	2218ms
4000	58517	2098ms	3286ms	1703ms	3576ms
5000	69349	1881ms	5119ms	2142ms	4999ms
10000	142610	10017ms	27535ms	8848ms	19348ms

V. Conclusion

Overall, an important lesson that was gleaned from this project was the potential misapplication of parallel programming. Not every algorithm benefits from concurrency the same way; in the case with some of Dinic's potential applications with different graph traversal algorithms, DFS suffered more of a performance overhead than actual gains in speed. Ford-Fulkerson's parallel implementation never quite sees significant practical usage until test-cases began to show considerable speedups from larger graph sizes. Edmonds-Karp was never able to see any valuable reason for parallelization, but was able to improve upon its initial approach. We gained a significant amount of knowledge in the field of network flow through this research as well as some insight into how locking can affect the run times of algorithms in parallel.

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. Network flow is not a particularly well-explored topic in parallel programming, remaining a more niche field of study. As a result, the majority of research that we were able to find lacked solid implementation examples or were out of scope for what the project team was able to do at this point in time.

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] Zhipeng Jiang, Xiaodong Hu, and Suixiang Gao. "A Parallel Ford-Fulkerson Algorithm For Maximum Flow Problem". In: 2013.

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

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

[6] SumitPadhiyar. *Sumitpadhiyar/PARALLEL$_F$ORD$_F$ULKERSON$_G$ $Parallelizationof FordFulkersonalgorithmonGPU$.* URL: https://github.com/SumitPadhiyar/parallel_ford_fulkerson_gpu.

[7] 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.

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