EC504 Final Report: Parallel Pre-Flow Push with MPI

Joshua Stern
Fall 2018

I. Problem Description

On of the most important problems in the field of computer science is the max-flow problem, which involves finding the maximum possible flow within a graph from a starting source to an ending sink node. By using a graph to represent a physical network of routers, we can use max-flow algorithms to model the maximum amount of network traffic that can traverse a network between any two points, as well as provide directions on how to route this traffic so that no network link is forced to carry flow that exceeds its capacity. Without max-flow algorithms, networks would either underperform by wasting unused capacity or become congested by overloading network links. The most popular and simple max flow algorithm is Ford Fulkerson, which calculates maximum flow by finding global augmentations of the network and pushing flow across these paths. It keeps repeating this process until no further augmentations can be found, and results in a time complexity of O(E*f), where E is the number of edges and f is the maximum flow. A faster version of Ford Fulkerson, the Edmonds-Karp algorithm, is able to achieve a time complexity of $O(VE^2)$ by using Breadth-First Search to find these augmentations. However, other algorithms are able to achieve even better complexities.

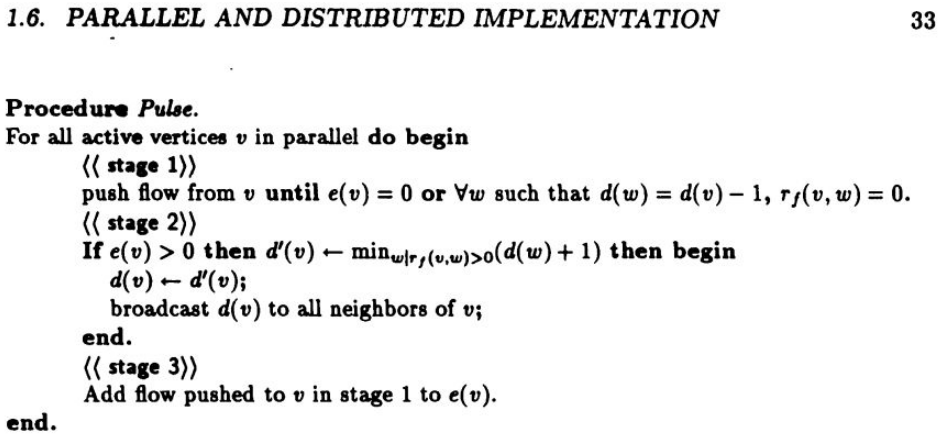
As networks becoming increasingly large, nodes cannot store global information due to the large amount of memory it would require. This creates the motivation for an algorithm such as the pre-flow push algorithm in which each node can just maintain local information about itself and its neighbors. In this algorithm, flow is pushed from the source node regardless of whether or not it will make it to the sink. Once this flow reaches a point where all outgoing edges have no more available capacity, the flow is sent back and looks for another way to reach the sink. If no possible path exists, it returns to the source. The pre-flow push algorithm runs in $O(V^2E)$ time, which is faster than both Ford-Fulkerson and Edmonds-Karp. One of the primary goals of this project is to implement the pre-flow push algorithm to find the max-flow of a network, while only allowing each node to store the information about itself and its adjacent nodes.

As Moore's Law begins to fail and the acceleration of processor frequencies continues to slow down, computer scientists are seeking alternative methods to accelerating applications such as better algorithms and more parallelism. The pre-flow push algorithm is seemingly the best algorithm for solving the max-flow problem. However, we can extract parallelism from the algorithm to achieve further speedups. Due to each node only needing to hold information on itself, its outgoing and incoming edges, and its adjacent nodes, we can have all nodes perform local computations at the same time. My goal is to use MPI in order to have every node perform computations in parallel, as well as send messages to each neighbor that contain information about pushed flow and changes in height labels. I also wanted to ensure that each node was only using local information by simulating my algorithm on an actual parallel cluster. For this, I use the BU Shared Computing Cluster (SCC) and have each node run in parallel. By basing my algorithm off the Pulse algorithm developed by Andrew Vladislav, I am able to achieve a time complexity of $O(V^3)$.

II. Relevant References and Background Materials

a. Pre-Flow Push Algorithm

The version of the pre-flow push algorithm I chose to implement is based on the Pulse Algorithm developed by Andrew Vladislav. Vladislav published his algorithm, which is used to implement pre-flow push in a parallel environment, in his 1987 Ph.D. dissertation *Efficient Graph Algorithms for Sequential and Parallel Computers by Andrew Vladislav Goldberg* (Pg 33-34). The full text can be found at http://hdl.handle.net/1721.1/14912. Figure 1 below shows an image taken from his dissertation outlining the algorithm. I will further describe this algorithm and provide specifics on how I implemented it in Section III of this report.



1.6. PARALLEL AND DISTRIBUTED IMPLEMENTATION 33

Procedure ***Pulse.***
For all active vertices v **in parallel do begin**
⟨⟨ **stage 1**⟩⟩
push flow from v **until** $e(v) = 0$ **or** $\forall w$ **such that** $d(w) = d(v) - 1$, $r_f(v, w) = 0$.
⟨⟨ **stage 2**⟩⟩
If $e(v) > 0$ **then** $d'(v) \leftarrow \min_{w|r_f(v,w)>0}(d(w) + 1)$ **then begin**
$d(v) \leftarrow d'(v)$;
broadcast $d(v)$ to all neighbors of v;
end.
⟨⟨ **stage 3**⟩⟩
Add flow pushed to v in stage 1 to $e(v)$.
end.

Figure 1: Vladislav's Pulse Algorithm for Pre-Flow Push

Goldberg, Andrew Vladislav. *Efficient graph algorithms for sequential and parallel computers*. Diss. Massachusetts Institute of Technology, 1987.

b. MPI

The Message Passing Interface (MPI) is the de facto standard for message passing in distributed and parallel networks. MPI allows programmers to specify the sending and receiving of messages between specific processes, which I take advantage of in my implementation by using MPI to send flow and label information between nodes and their neighbors. In developing my code on my personal computer, I use MPICH-3.2, a popular version of MPI. The BU SCC has OpenMPI already installed, but the code runs properly on both systems. In Section IV, I will provide instructions for installing MPICH-3.2, which should be used if you are testing the code on your own computer. If testing the code on a BU lab computer, SCC, or any other computer where MPI is already installed, then no further installations are necessary.

III. Implementation

a. Vladislav's Parallel Pre-Flow Push Algorithm

As previously mentioned, my algorithm for computing max-flow is based on a parallel pre-flow push algorithm proposed by Vladislav in his dissertation on parallel algorithms. Vladislav's algorithm takes advantage of parallelism by having each node perform local computations in parallel. Each node then sends the results of its local computation to its neighbors, and then the process repeats. Vladislav's algorithm can be divided into 5 stages (or pulses, as Vladislav refers to them as):

1. Each node pushes its excess flow to its neighbors
2. Each node recalculates its height
3. Each node receives the flow from step 1, and adds any received flow to its own excess
4. Each node sends its new height to all of its neighbors
5. If there is still excess, repeat

Analysis: Because all nodes can perform these operations in parallel, Vladislav's algorithm can run faster than a sequential pre-flow push algorithm. In his analysis, he states that the maximum number of times these steps can be performed in $4V^2$. Since in each stage, a node can send flow to a maximum V other nodes, the total number of non-saturating pushes is limited to $4V^3$. This makes the time complexity of his algorithm $O(V^3)$.

b. My Implementation with MPI

<u>Step 1: Have each node get its local data</u>
After reading in the graph information and adjacency matrix, each node gets two arrays, each of size O(V). One array is used to hold all of the incoming edge capacities and adjacent vertices, while the other array contains all of the outgoing edge capacities and adjacent vertices. Once each node has filled these arrays using the adjacency matrix, it can use them to complete the remainder of the algorithm. The arrays are essentially taken from the rows and columns from the adjacency matrix. The adjacency matrix will not be used again, meaning that each node will only have its local information.

Time Complexity = O(V) Space Complexity = O(V)
Data Structures: Two arrays of size O(|V|) that contain the capacities of all connected edges, as well as the ranks of all adjacent vertices

<u>Step 2: Perform BFS for each node to get the height of itself and its neighbors</u>
Each node then computes its height using a form of breadth-first search. I implemented this by having each node receive the heights of all adjacent nodes. Each node then adds 1 to the minimum height value that it received, and then forwards this value to all its adjacent nodes. This process is started at the sink node, which is given a height of 0. At the end, the height of the source node is set to |V|. Each node ends up with the value of its own height, as well as an array containing the heights of all adjacent nodes. All message passing is done using MPI.

Time Complexity = O(|V| + |E|) Space Complexity = O(n)
Data Structure: An array of size O(|V) that contains the heights of all adjacent vertices

Step 3: Perform initial push from source node
The source node fills its outgoing edges with flow matching their capacities. All vertices on the other sides of these edges add this received flow to their excess. All message passing is done with MPI.

Time Complexity = O(V)

Step 4: Implement Vladislav's algorithm with MPI

- Each node pushes its excess flow to its neighbors
    - Each node uses MPI_Send to send flow to its adjacent nodes until it no longer has any excess, or it has filled up the edge capacities. While doing so, nodes sending out flow update their arrays containing data on their connected edge capacities.
- Each node recalculates its height
    - Each node uses its height array to relabel itself if it still has excess flow. No message passing is done in this step.
- Each node receives the flow from step 1, and adds any received flow to its own excess
    - Each node uses MPI_Recv to receive the flow sent in step 1. If flow is received from an edge, then the arrays containing data on their connected edge capacities are updated.
- Each node sends its new height to all of its neighbors
    - Each node uses MPI_Send and MPI_Recv to send a receive height labels to and from all adjacent nodes
- If there is still excess, repeat
    - Perform an MPI_Allreduce to see if any excess still exists. If yes, repeat step 4. Else, the algorithm is completed and the sink node can output its excess.

Time Complexity = $O(V^3)$

Total Algorithmic Time Complexity = $O(V^3)$ Total Space Complexity = O(V)

c. Assessment of goals met from project proposal

Features listed in project proposal:

- Must implement a parallel pre-flow push max flow algorithm on a multi-node cluster.
    - This feature was implemented with my code. The algorithms used included the steps outlined above on top of Vladislav's algorithm for a parallel pre-flow push. The data structures used were edge capacity arrays and a heights array. The total complexity of this algorithm was $O(V^3)$.
    - The requirement of implementing the code on a multi-node cluster was also met, as I was able to run my code on the BU SCC, with instructions for replicating this provided in Section IV.

- Each node knows the the capacity of its neighboring edges and heights of adjacent nodes.
    - In my code, each node only has arrays about the capacity of its connected edges, the vertices that it is connected to, and the heights of these vertices. Although the program requires an adjacency matrix to run, this matrix is only used to fill these arrays and this discarded.
- Takes as input an adjacency matrix with the edge capacities replacing the '1's in the matrix.
    - I designed the code to accept an adjacency matrix, as this was simpler than requiring separate input for each vertex. As previously mentioned, this adjacency matrix is only used to build the local data arrays.
- Must return the maximum flow of the network.
    - I tested the code on flow graphs with known solutions and was able to achieve the same solutions.
- Use the Message Passing Interface (MPI) for inter-node communication.
    - All message passing between vertices is done through MPI. For the exchange of flow and height data, point to point messaging is done with MPI_Send and MPI_Recv. To calculate how much excess flow remains in the network, MPI_Allreduce is used.

d. Experiments and Results

I started by testing my algorithm on very small graphs between 2-4 vertices just to make sure that it was working for small cases. Then towards the end of my debugging stage, I tested by algorithm on a flow graph provided from homework 6 (shown below), which consisted of 8 vertices and 11 edges. In all my tests, I was successfully able to calculate the correct maximum flow from any given source node to any other sink.

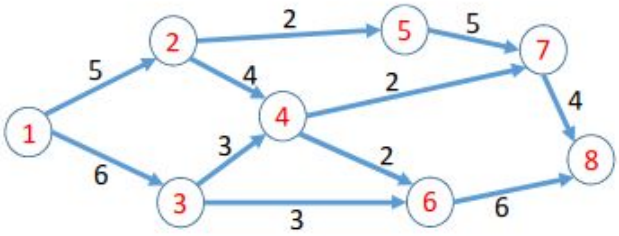


Figure 2: Example flow graph used

In a max-flow algorithm, the user just wants to know what the max flow is between any two nodes. This makes the output format simple. My code takes in a text file containing graph metadata and an adjacency list (further outlined in Section IV). Sample output is as follows:

time = 0.092960s
capcity = 9

IV. Supporting Files, Compilation and Execution, Examples

The code and files for this project can be found at https://github.com/joshstern1/504Project
On this github is the code (parallel_preflowpush.c), an example graph input file (graph.txt) and a readme that just explains the same concepts and instructions found in this section.

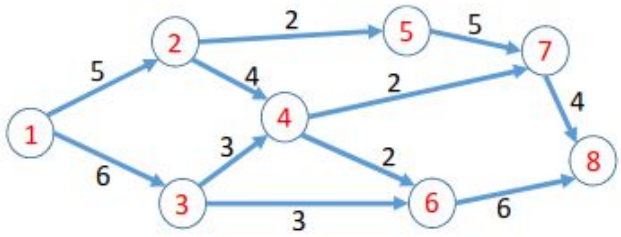
a. Supporting Files

- The main code file is *parallel_preflowpush.c*. This program, when given an argument of a text file containing information on a graph's vertices, edges, capacities, source, and sink, outputs the max-flow.

- Input graph file (ex: graph.txt)
  - The program expects one argument: a text file containing an adjacency matrix for the input graph. The format of the text file is as follows:

    #Vertices
    #Edges
    Rank of source
    Rank of sink
    Adjacency matrix
    The number 0 (used to indicate the end)

  - Each entry in the adjacency matrix contains either the capacity of an edge, or a 0 if the edge does not exist




```
8
11
0
7
0 5 6 0 0 0 0 0
0 0 0 4 2 0 0 0
0 0 0 3 0 3 0 0
0 0 0 0 0 2 2 0
0 0 0 0 0 0 5 0
0 0 0 0 0 0 0 6
0 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0
0
```

Figure 3: Example flow graph and its associated input graph.txt file (Note: In MPI, rank numbers start at 0, so in converting the input graph into the text file, I had to decrease the rank of each node in the graph by 1. This graph.txt file is included in my submitted zip file and can be found at https://github.com/joshstern1/504Project).

- MPI installation
    - If you are running the code on a system with MPI already installed, you may skip this section. If you are running this code on your PC, and you do not have MPI installed, download MPICH-3.2 from:

        http://www.mpich.org/static/downloads/3.2.1/mpich-3.2.1.tar.gz

    - Installation instructions:

1. Unpack the tar file:

    tar xfz mpich.tar.gz

2. Create an installation directory:

    mkdir /home/you/mpich-install

3. Create a build directory:

    mkdir /tmp/you/mpich-3.0.2

4. Configure MPICH:

    *cd /tmp/you/mpich-3.0.2 /home/you/libraries/mpich-3.0.2/configure *
    -prefix=/home/you/mpich-install |& tee c.txt

5. Build MPICH:

    make 2>&1 | tee m.txt

6. Install MPICH commands:

    make install 2>&1 | tee mi.txt

7. Add mpich-install/bin directory to your path

    export PATH=/home/you/mpich-install/bin:$PATH

- The complete list of instructions can be found at:

https://www.mpich.org/static/downloads/3.2/mpich-3.2-installguide.pdf

b. Compilation and Execution

- This code was written in C. However, you must be careful to adhere to the following compilation instructions, or else the code will not run properly.

- To compile: *mpicc parallel_preflowpush.c*

- To run on a PC: *mpiexec -n #numprocs a.out graph.txt*
    - Replace #numprocs with the number of vertices in the graph
    - **Ex:** To run with 8 vertices:

      mpiexec -n 8 a.out graph.txt

      (if mpiexec does not work, replace with mpirun)

- To run on the SCC, you must specify the total number of nodes you want, and the amount of cores that you need on each node.
    - Let V = Total number of cores
    - Let N = Number of cores on each node
    - Note that you can allocate a max of 16 nodes, with 28 cores on each node, for a total of 448 cores

  qsub -pe mpi_28_tasks_per_node V -b y "mpirun -npernode N ./a.out graph.txt"

    - **Ex:** to run with 448 total cores (16 nodes), with 28 cores on each node:

  qsub -pe mpi_28_tasks_per_node 448 -b y "mpirun -npernode 28 ./a.out graph.txt"

- more details can be found at :

  http://www.bu.edu/tech/support/research/system-usage/running-jobs/parallel-batch/

V. Work Breakdown

I worked on this project independently. I wrote all of the code from scratch, performed all experiments, and wrote this report.

Signature: Joshua Stern Date: 12/12/28