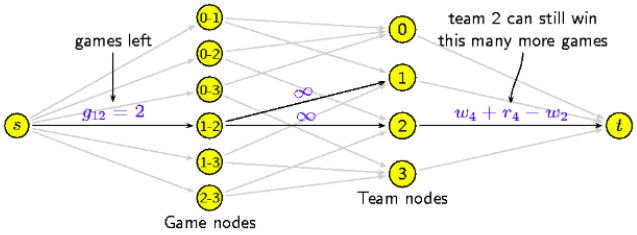
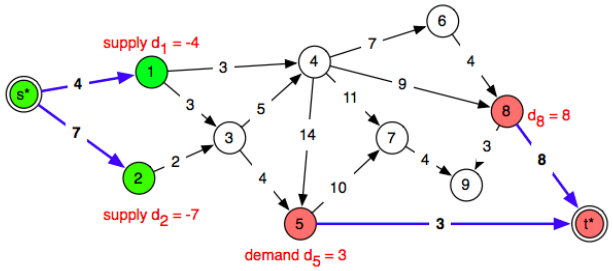
MaxFLO: Parallelizing 2 Different Max-Flow Algorithms





Wynne Yao and Katia Villevald

15-418 Final Project

SUMMARY
Our project focused on parallelizing and evaluating the performance of two commonly used algorithms for solving the maximum flow problem on graphs, Dinic's and Push-relabel. We created sequential and shared-memory (OpenMP) parallel versions of the algorithms. To enhance the performance of the parallel algorithms, we also integrated lock-free updates to certain variables and concurrent data structures (using the thread building block library in C++). Our evaluation of the algorithms involved running the algorithms on realistically-sized graphs and comparing the speedup of the parallel implementation to the sequential.

BACKGROUND

History and Applications of Maxflow Algorithms

Maxflow is an algorithm introduced in 1954, during the Cold War, by T. E. Harris and F. S. Ross as a way of modelling the Soviet railway traffic flow. In 1955, the first algorithm for solving maxflow, Ford-Fulkerson, was presented by Lester R. Ford, Jr. and Delbert R. Fulkerson. Over the years, new algorithms were developed with lower complexities, including push-relabel and Dinic's. Today, these algorithms are used to solve interesting problems like circulation-demand in industrial operations, airline scheduling, image segmentation in computer vision and more.

Maxflow Input and Output Representation

Maxflow algorithms take in an input of a graph with directed edges where each edge has a capacity representing the amount of flow that the edge can support. The input also contains the ids of the source node where the flow originates from and the sink node where the flow terminates at. The output is a mapping of each edge to an amount of flow through that edge where the flow through an edge cannot exceed the capacity of that edge and the total flow out of the source/into the sink is maximized. Note that the capacities and final flows through the edges do not necessarily have to be integers, however in our project we only consider integer capacities and flows.

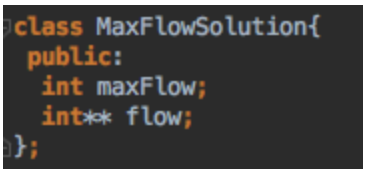
Our project has two types of input instances and two types output instances. Both input instances store the node ids of the source and sink nodes as well as the input graph. Both output instances store the maximum flow possible through the input graph and the final flows through each edge.

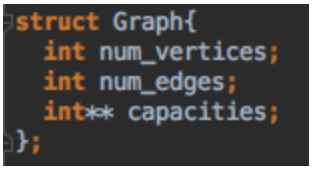
Capacities, Edges and Flows Representation

MaxFlowInstance represents the graph information as an adjacency matrix of capacities so if edge (i,j) with capacity c exists in the input graph, there would be an entry of c at cell (i,j) in the capacities matrix. MaxFlowInstanceSmall represents the edges and capacities of the graphs as adjacency lists (arrays of vectors). So if the edge (i,j) with capacity c exists in the input graph the ith list of edges would contain j and the ith list of capacities would contain the pair (j,c). Similarly, MaxFlowSolution represents the final flows as an adjacency matrix while MaxFlowSolutionSmall represents the finals flows as adjacency lists.

```
class MaxFlowInstance{
 public:
  Graph inputGraph;
  int source;
  int sink;
};
```

```
class MaxFlowInstanceSmall{
 public:
  GraphSmall inputGraph;
  int source;
  int sink;
};
```





Dinic's Algorithm

High Level Algorithm Description

Dinic's algorithm is made up of 2 main steps:
(1) BFS: Labels each vertex of the residual graph with its shortest distance from the source node (i.e. the level of the node).
(2) sendFlow: Pushes as much flow as possible from the source node to a neighboring 1st level node, then from the 1st level node to a neighboring 2nd level node … until the flow reaches the sink node of the residual graph. It repeats this process pushing as much flow from the source as possible.
These two steps are repeated until the BFS returns false because the sink node is unreachable from the source node in the residual graph at which point maximum flow has been pushed from the source to the sink.

Key Data Structures and Operations

There are 3 key data structures in Dinic's:

(1) **start** which is an n-length array used in sendFlow to keep track of what out edges for each node have been tried to push flow through
(2) **flows** which is an n-length array of vectors where the ith vector stores the pair (j,f) if the total flow pushed through edge (i,j) so far is f
(3) **levels** which is an n-length array that represents the distance of a node from the source in the residual graph.
(4) **capacities** and **edges** which are both read-only n-length arrays of vectors that represent the input graph (described in the Capacities, Edges and Flows Representation section).

The **start** structure is only read and updated in sendFlow. Whenever sendFlow makes a recursive call the start structure is passed on as an input. The ith entry of sendFlow is incremented whenever an adjacent edge of the ith node has attempted to have flow sent through it.

The **flows** structure is only updated in sendFlow if recursive calls of sendFlow reach the base case of the current node being the sink and output a positive flow indicating that that much flow can be sent from the source to the sink. The values of flows are read in both BFS and sendFlow to determine whether a residual edge in the residual graph exists (i.e. if flow through the original edge < capacity of the original edge). Since flows is an array of adjacency lists a read query for the flow through edge (i,j) involves scanning through the ith adjacency list to find the tuple (j,f) which can be expensive.

The **levels** structure is only updated in each run of BFS. At the start of the BFS all the entries of level are set to -1 (i.e. are unlabeled) except the source which is labeled with a 0. Then for each node that has a parent node labeled with l and there exists a residual edge between the parent node and itself, the node is labeled l+1 (i.e. the node entry of levels is set to l+1). The levels structure is read in sendFlow to determine whether to send flow through through it.

Potential for Parallelism and Dependencies

Although sendFlow has a for loop that creates recursive calls, we cannot parallelize over it since after one iteration of the for loop the residual graph that the flow would be sent through would be different than the initial residual graph (since the flows structure could be updated in the iteration). So parallelizing the for loop could cause correctness issues. An additional difficulty is that the recursive calls of one iteration of the for loop in sendFlow have to be executed in the order in which they are called since the flow input variable that stores the current flow through a vertex depends on the flow that could get through its parent node. From this analysis, it appears as though we can't take advantage of parallelism in this function.

The BFS function however has a level-based parallelism. For each level from the source node, the elements in the queue for that level can be labeled independently of the other vertices. If a node already has a label they are skipped otherwise level[node] is set to the current level of the BFS. One source of contention is that all newly-labeled vertices would have to be added to the queue for the next iteration which might make the synchronization step more costly.

Initially, when we represented the input graph instance to Dinic's with an adjacency matrix we found that the initialization phase was actually the bottleneck of the sequential algorithm. This function was very parallelizable as it just involved updating independent i or (i,j) entries of structures to their initial values. One source of contention between pairs of threads is that both the (i,j) and (j,i) entry of flows have to be initialized so these two updates would need to be

separated into two parallel sections. With adjacency lists there is still some parallelism over vertices that can be done during the initialization step.

Another potential place for parallelism is when searching for a flow through edge (i,j) in the ith adjacency list, although if the max outdegree of the graph is fairly low, adding parallelism might mean adding unnecessary complexity.

Push-relabel

High Level Algorithm

Sequential Algorithm

The original serial version of push-relabel involves first sending out a preflow, which initially sends out flow equal to the capacities of the edges from the source to the source's neighbors. Vertices in p4ush-relabel have heights. These heights reflect how far away from the sink the vertex is. Each vertex can also have an excess amount of flow. Vertices that have nonzero excess flow are called active vertices. From the active set, a vertex is chosen. It can only push onto its neighbors that have a height 1 less than its own height. We call the edge between the active vertex and such a neighbor an admissible edge. If the active vertex doesn't have any neighbors on which to push, the vertex's height will be relabeled to be the minimum of its neighbors heights plus 1 (so that the next round, it can push somewhere). The active vertex set updates according to the updated flows, and this process continues until there are no more active vertices. The output is the amount of excess flow the sink has.

Parallel Algorithm

We based our parallel implementation off of the paper by Baumstark, et. al. The parallel high level algorithm involves first doing the preflow, same as before. For one iteration, the active vertices are processed in parallel and possible pushes are performed. Each of the active vertices has most if not all of its excess flow pushed out in one iteration. However, the modified excess flows are saved separately and not applied until the end. New labels are computed in parallel but not applied until the end of the iteration. At the end of processing all the active vertices, the new labels and the new excess changes are applied and a new active set is created before the next iteration. After a few iterations, a global relabel is called, which uses a reverse BFS to assign the vertices heights that mirror their distances from the sink again. If the working set is empty, the algorithm stops and outputs the sink's excess flow as the max flow.

Key Data Structures

(1) **flows** is a 2d matrix where flows[i][j] represents the flow from vertex i to vertex j. This is updated during each iteration.
(2) **excessPerVertex** is an n-length array of the excesses each vertex has. This gets updated atomically during each iteration, as well as at the end of the iterations but just for the ones in the active set.
(3) **d** is an n-length array of the labels for each vertex which is only updated at the end of each iteration.
(4) **workingSet** is an unordered set that contains the active vertices for the current iteration. It is only updated at the end of each iteration with the discovered vertices.

(5) **residual** is a 2d matrix where residual[i][j] reflects the residual capacity from vertex i to vertex j. **residual** is only updated at the end of each iteration. residual[i][j] = capacities[i][j] - flows[i][j].
(6) **discoveredVertices** is a vector of concurrent tbb vectors such that discoveredVertices[i] is a vector of all the vertices that active vertex i had discovered on that iteration.
(7) **addedExcess** is an array of length n that contains how much the excess had changed for vertex i during that iteration.
(8) **copyOfLabels** is an array of length n that contains the newly computed labels for the active vertices that need relabeling during that iteration.
(9) **reverseResidual** is an n-length array of vectors where the jth vector has integer i if there exists an edge (i,j) in the residual graph

Key Operations

Most of the data structures (discoveredVertices, d, addedExcess, workingSet, residual, reverseResidual) are updated or reset to their default values at the end of each round of the pushRelabel function. In globalRelabel, before the reverse BFS, the reverseResidual structure is updated using the residual structure. There are some structures that are updated within the push-relabel iteration (during the pushing/relabeling phase). discoveredVertices is updated whenever a vertex has flown pushed to it or has excess flow and addedExcess is updated whenever flow is pushed to a new vertex.

Potential for Parallelism and Dependencies

The most significant potential for parallelism is the fact that active vertices could be processed at the same time as long as they could eventually see the changes to excess flow and labels. Some of the dependencies included having to update residual capacities separately from the flow, since the current round cannot see the effects of the current pushes until the end of the iteration. Other dependencies included having all the threads push onto the same sets or vectors such as the working set or the discovered vertices when creating the new working set or keeping track of the current iteration's discoveries. Because of potential concurrency issues if this was done in parallel, this was also something that initially limited parallelism. Of course, there was also the fact that the algorithm still needs to be somewhat iterative, but each iteration now does more work with its active vertices before moving on to the next working set (one such example is to push out as much flow from the active vertex as possible before needing to synchronize before creating the next working set).

APPROACH

Technologies and Libraries Used

For the parallel version of both Dinic's and Push-relabel our goal was to create versions that could be run on a single multi-core machine. Specifically, we created our algorithm for the 8-core Gates machines used for previous assignments (the specification for which can be found here: https://ark.intel.com/content/www/us/en/ark/products/92985/intel-xeon-processor-e5-1660-v4-20m-cache-3-20-ghz.html).

We used the OpenMP library to parallelize both algorithms. Additionally we used Intel's C++ library called thread building blocks (or TBB) for concurrent data structures like their concurrent_vector and concurrent_unordered_set. These concurrent structures, at the cost of not storing all the elements contiguous in memory, reduce thread contention using fine-grained locking and lock-free techniques.

Parallelizing Dinic's

Sequential Algorithm

The original sequential algorithm that we used for Dinic's was from https://www.geeksforgeeks.org/Dinic's-algorithm-maximum-flow/. It was adjusted to use an adjacency matrix for the input graph representation.

Changes to Sequential Algorithm

Since the most parallelizable function of the Dinic's algorithm is the BFS, we experimented with a slightly different algorithm for the BFS called Parallel-BFS with bags. For each level of the BFS, if the working vector was larger than the cutoff, the vector would be split in 2 and a task for one of the halves would be launched by the master thread. The two halves would then be processed in parallel. Otherwise, if the vector is small enough the vertices would be processed in a for loop, as before.

Mapping Work to Threads

For the Parallel-BFS with bags we created a new task for one of the recursive calls of processLevel (a helper function that labels the vertices passed in with the appropriate level) and mapped it to any available thread (using the untied keyword in OpenMP). For the original BFS, we mapped each vertex in the queue at a level to an available thread (the vertices in the queue are those discovered in the previous level of the search). As to how the work is assigned to cores, OpenMP allows the OS to handle that mapping.

Parallelization Process

Our original sequential implementation of Dinic's represented the capacities and flows as adjacency matrices which is why after running the performance analysis we found that the initialization step was actually the hotspot (with over 75% of the execution time being spent in this step). So the first step was to parallelize this function by separating the updates to the flows in a way that no two threads would try to update the same place in the matrix twice. This improved performance significantly but for graphs with larger sized nodes, the function initialize continued to be the main source of contention. An additional problem we faced is that since Dinic's was relatively fast on the test cases that could be stored in memory (i.e. graphs with up to 2^16 nodes) it was hard to see any interesting results for the parallel version.

To run Dinic's on larger test cases and focus parallelization on the main parts of Dinic's algorithm, like the BFS function, we changed the input (capacities/edges) and output (flows) to Dinic's to be adjacency lists rather than matrices. As a result we could now test the algorithm on much larger test cases, up to 32 times larger, and the newer bottlenecks were now in the sendFlow and BFS functions. Additionally, using adjacency lists improved performance of both

the sequential and parallel algorithms significantly. Adding a parallel BFS with bags, improved performance on larger tests cases, but this was not immediately clear as the parallel BFS seemed to hurt the smaller test cases. We also tried to add a concurrent container for the parallelBFS since the order of the vertices in the vector at a level does not matter, however we had issues getting this idea to work with OpenMP tasks, perhaps because the tasks were generated recursively. Finally, we also tried smaller optimizations like parallelizing the search in the adjacency list for a specific flow in the sendFlow and BFS functions (using a C++ parallel algorithms library). However, this did not help performance either probably because the max out-degrees of the graphs we tested on were fairly small.

Parallelizing Push-Relabel

Sequential Algorithm

The original sequential algorithm that we used for push-relabel was from the 15-451 notes which can be found here: https://www.cs.cmu.edu/~avrim/451f13/lectures/lect1010.pdf

Changes to Sequential Algorithm

The change from the sequential version of the algorithm to the parallel one was mentioned above in the section titled “High Level Algorithm.” We based our parallel version of the algorithm off in the paper authored by Baumstark, Blelloch, and Shun found here: https://arxiv.org/pdf/1507.01926.pdf. However, we ended up tweaking some aspects due to some errors that we found in the pseudocode. We also added additional optimizations such as using tbb concurrent data structures and adjacency lists to get a higher speedup.

Mapping Work to Threads and Tasks

We created a new task for each active node of the working set for each iteration and mapped it to any available thread (using the untied keyword in OpenMP). As to how the work is assigned to cores, OpenMP allows the OS to handle that mapping. We used a vector of concurrent vectors for the discoveredVertices from the tbb library. The way that this concurrent vector works is that as it grows it never moves existing elements. Instead of reallocating and moving its elements, to add more elements, it instead allocates an additional chunk of memory called a “segment.” Thus, multiple threads are able to append new elements concurrently and grow the container.

Parallelization Process

Having finished writing the parallel code based off of the paper, we ran a perf report for hotspots and saw that a significant bottleneck was in PushRelabel (around 35%) because of the slow n^2 time update of the residual capacities before the start of the next iteration. To fix this, we instead only updated the residual capacities of the vertices that had been in the working set and the vertices that those active ones had pushed to during the iteration. We could then combine this step with a preexisting loop that looped through the working set to add the excess changes and relabel the nodes. We also moved the initialization of the residual capacities from its own for loop to one that was already going to be performed in the preflow.

After correcting that bottleneck and running perf again, we saw that the bottleneck was now in the globalRelabel step. Around 93% of the time was spent on traversing the residual array column-wise. The traversal was column-wise because we needed to check the nodes v that were, for instance, going to the sink w that had non zero residual capacities; so we needed to know for all v, if residual[v][w] was non-zero. This column-wise traversal caused a lot of cache misses and made the process slow. Thus, we created a reverseResiduals vector such that reverseResiduals[w] would contain all the residuals from nodes v to w. This significantly drove down the amount of time that was spent on that line.

Next, we noticed that a lot of the time, we're not able to add for instance, a parallel for, when looping over a set or a vector that had threads pushing to it, since, without a lock, this would cause correctness issues. Thus, we tried using tbb concurrent vectors in order to be able to correctly push back onto a frequently-used vector like discoveredVertices and have it be less of a bottleneck. We also originally made discoveredVertices a 2D matrix, but this was quite slow and could have potentially caused a lot of false sharing. Instead, we changed discoveredVertices to be a vector of concurrent vectors, or essentially an adjacency list.

One thing we struggled with was changing the algorithm from the iterative version to the parallel one, since we had to figure out what could be updated during the iterations as opposed to being saved to update at the end of each iteration. For instance, we couldn't check whether there was an edge from vertex i to j in the residual graph within the iteration by checking cap[i][j] - flows[i][j] since flows[i][j] could change. Instead, we realized we needed to save the residual capacities before and after each iteration and use those to check if there was an edge in the residual graph between two vertices. Additionally, after adding OpenMP, we also had to figure out which places required atomic fetch-and-adds or compare-and-swaps (as opposed to having one large lock), such as atomically fetching and adding to the added excess vertex since multiple vertices could be neighbors to that vertex and be pushing flow to it.

Problem of Storing Large Graphs in Memory

As mentioned in previous sections one of the problems we faced with both algorithms is the ability to scale. Adjacency Matrix representations of the graph often resulted in higher speedups in the parallel implementations but suffered the problem of not being able to scale well. Adjacency lists did not result in such good speedup but overall improved performance of both the sequential and parallel algorithms and could be run on larger test cases. As one of our goals was to run the algorithms on realistic sized graphs, our secondary goal (after improving speedup) was reducing storage required for each algorithm. This seemed reasonable since to scale well the algorithms should perform well and fit in memory of the machine.

RESULTS

Performance Metrics

We evaluated our parallel algorithms by comparing their wall-clock times (in seconds) and speedup to their sequential counterparts. Specifically, we ran the command "numactl

--physcpubind=0” with our parallel versions to see their sequential times and used that time in our speedup calculation.

Performance Benchmark Used

Our benchmark consists of primarily delaunay graphs of various sizes, to see interesting trends in scaling our algorithms, and an additional rgg graph, to model our algorithms on different types of graphs. The choice of these types of graphs was inspired by the benchmark of graphs used by Baumstark Blelloch and Shun [2] in their evaluation of their push-relabel parallel algorithm. Both delaunay and rgg graphs also have interesting real-world applications as delaunay graphs are used to construct mesh models of objects and rgg graphs are used in modelling wireless connection networks. The table below shows the complete list of graphs that were used in the benchmark and their properties:

	Num. of Vertices	Num. of Edges
delaunay_n10	1024	3056
delaunay_n11	2048	6127
delaunay_n12	4096	12264
delaunay_n13	8192	24547
delaunay_n14	16384	49122
delaunay_n15	32768	98274
delaunay_n16	65536	196575
delaunay_n17	131072	393176
delaunay_n18	262144	786396
delaunay_n19	524288	1572823
delaunay_n20	1048576	3145686
delaunay_n21	2097152	6291408
rgg_n_2_15_s0	32768	160240

The delaunay graphs that we retrieved were from the 10th DIMACs Implementation Challenge website (https://www.cc.gatech.edu/dimacs10/archive/delaunay.shtml) and the rgg graph was retrieved from the Karlsruhe High Quality Partitioning website (http://algo2.iti.kit.edu/documents/kahip/index.html). These graphs had to be converted to a specific DIMACs instance of the maxflow problem before being read by our parser. We randomly generated capacities for each of the edges in these graphs with maximum capacity being 1000.

Experimental Setup

For Dinic's we performed the following experiments:

1. Parallel vs Sequential Times and Speedup
2. BFS vs parallelBFS
3. Adjacency List vs Adjacency Matrix

For Experiment 1, we ran each test in the benchmark 3 times on the parallel Dinic's algorithm (with adjacency lists) and took the average of the 3 trials. For graphs with the number of nodes less than 2^16, we ran Dinic's with the regular BFS. For graphs with the number of nodes >= 2^16, we ran Dinic's with the parallel BFS, since at this point, the data showed that the parallel BFS resulted in a better runtime than the regular BFS. To determine the speedup for this experiment we compared the runtime of the parallel Dinic's algorithm on a single thread. We also included the times using the original sequential algorithm.

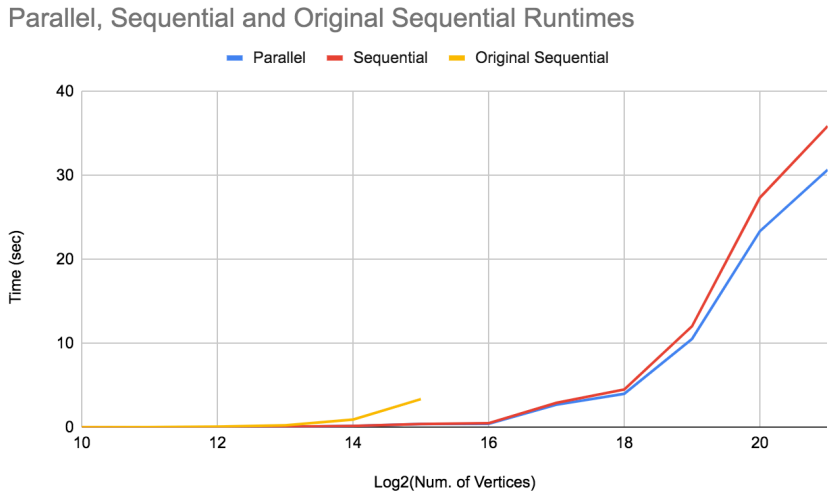
For Experiment 2, we ran each test in the benchmark 3 times on the parallel Dinic's algorithm (with adjacency lists) with the regular BFS and the parallel BFS, recording the average time spent in the BFS function over the 3 trials.

For Experiment 3, we ran each test in the benchmark 3 times on the parallel Dinic's algorithm using adjacency lists, taking the average over the 3 trials. For the parallel Dinic's algorithm using adjacency matrices we ran the tests from the benchmark that could fit in memory. Like in experiment 1, we calculated the speedup of each algorithm by comparing its parallel runtime to the single thread runtime.

For Push-relabel, we only performed the Parallel vs Sequential Times and Speedup experiment. Like for Dinic's, we ran all the tests from the benchmark that fit into memory 3 times and took the average of the 3 trials. To determine the speedup we compared the runtime of the parallel algorithm to the runtime of the algorithm run on a single thread. We also included the times using the original sequential push-relabel algorithm.

Results for Experiment 1: Parallel vs Sequential Times and Speedup

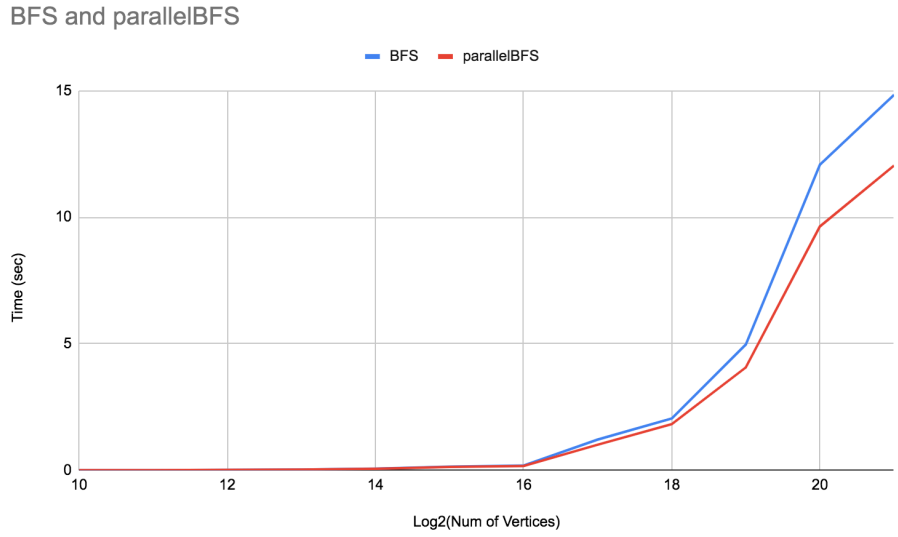
The graph below shows the runtimes of the Parallel Dinic's algorithms, Parallel Dinic's algorithm ran on a single core (i.e. Sequential) and the Original Sequential algorithm. Although the parallel version's runtime does appear to be smaller than the sequential's, especially on graphs with more than 2^17 nodes, the speedup table indicates that the speedup isn't significant given the number of cores.



Test Case	Speedup
delaunay_n10	0.998
delaunay_n11	1.008
delaunay_n12	1.012
delaunay_n13	1.004
delaunay_n14	0.984
delaunay_n15	0.984
delaunay_n16	1.149
delaunay_n17	1.084
delaunay_n18	1.129
delaunay_n19	1.143
delaunay_n20	1.171
delaunay_n21	1.169
rgg_n_2_15_s0	1.000

Results Experiment 2: BFS vs parallel BFS

The main function of Dinic's that we tried parallelizing is the BFS function. Below is a graph of the runtime of the parallel Dinic's algorithm (where the graph is represented with adjacency lists) when ran with the regular BFS and the parallel BFS with bags. It appears as though BFS and parallelBFS have around the same performance for a smaller number of nodes. In fact, the regular BFS performs better when the number of nodes is smaller perhaps due to the added complexity of recursion in the parallelBFS. However, when the number of nodes hits 2^18 we can see that the parallelBFS begins to outperform the regular BFS. Note that we used a cutoff of 128 to determine when to split the vector of nodes into halves.



Results Experiment 3: Adjacency List vs Adjacency Matrix

The table below demonstrates that although adjacency matrices achieved reasonable speedup on graphs where the number of nodes was 2^16, the machine could only fit graphs with up to that many nodes in memory. From the other table, it appears as though adding adjacency lists improved the runtime of Dinic's on both single and multiple threads which as a result reduced the speedup. Adjacency lists also allowed parallel Dinic's to be run on all the test graphs in the benchmark.

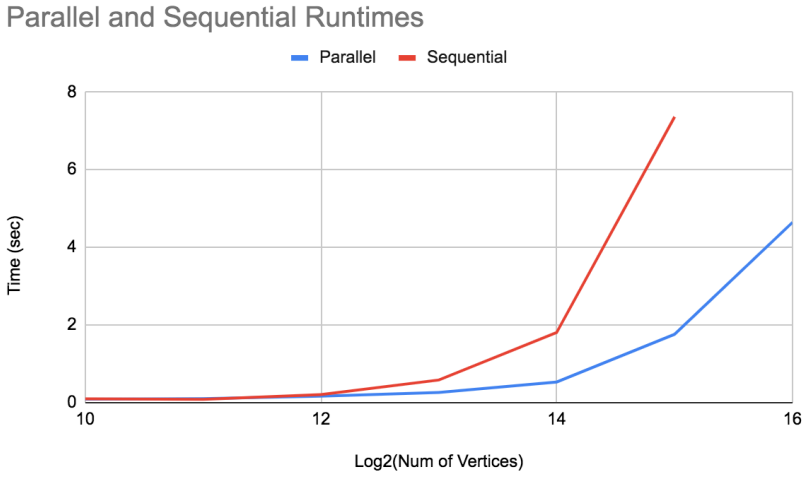
Log(Num. of Vertices)	Speed Up Adjacency List	Speed Up Adjacency Matrix
10	0.998	0.883
11	1.008	1.030
12	1.012	1.210
13	1.004	1.605
14	0.984	2.049
15	0.984	1.901
16	1.149	4.675
17	1.084	NA
18	1.129	NA
19	1.143	NA
20	1.171	NA
21	1.169	NA

	Runtime of Adjacency List (sec)	Runtime of Adjacency Matrix (sec)
delaunay_n10	0.01499	0.01804
delaunay_n11	0.01380	0.02743
delaunay_n12	0.04138	0.06326
delaunay_n13	0.08326	0.12468
delaunay_n14	0.17940	0.30127
delaunay_n15	0.41507	0.59118
delaunay_n16	0.43637	1.48935
delaunay_n17	2.70458	Killed (DFM)
delaunay_n18	3.99820	
delaunay_n19	10.53835	
delaunay_n20	23.36808	
delaunay_n21	30.70295	
rgg_n_2_15_s0	0.26925	0.43557

Analysis of Dinic's Results

From experiment 1 we see that the speedup of the final parallel Dinic's is not very significant. Most likely this is due to the fact that a significant portion of the algorithm, the sendFlow function, is inherently parallel and recursive. The sendFlow bottleneck became more apparent after running performance reports on larger test cases, starting from graphs with 2^16 nodes. For these test cases, no matter which BFS algorithm was used, sendFlow consistently comprised between 18 to 25% of the execution time.

Speedup and Times for Push-relabel



	Speedup
delaunay_n10	1.136
delaunay_n11	0.802
delaunay_n12	1.257
delaunay_n13	2.271
delaunay_n14	3.443
delaunay_n15	4.200
delaunay_n16	NA
rgg_n_2_15_s0	4.670

	Original Sequential Runtime (sec)
delaunay_n10	2.569932
delaunay_n11	17.7736115
delaunay_n12	159.02847

Analysis of Overall Speedup and Times for Push-relabel

Here were some of the percentages of time spent in the two major parts of the push-relabel algorithm.

Test case	% time globalRelabel	% time pushRelabel
delaunay_n10	.78	2
delaunay_n11	1.66	2.86
delaunay_n14	8	15
delaunay_n15	14	17

In the perf report of the delaunay_n11 test case, the majority of the time was spent in neither globalRelabel nor pushRelabel, but rather in a library called libgomp, which is something that OpenMP uses to manage threads. Perhaps to get a better speedup on smaller test cases, we might need to switch to using a different parallel framework to avoid this overhead.

We speculated that one thing that is limiting speedup is the fact that the algorithm is still inherently iterative - each iteration concatenates together the new working set before the next iteration, thus the iterations have dependencies between each other. Another speculated limiter is false sharing because of the large 2D matrices we use. Instead, they could be replaced with adjacency lists. Furthermore, the working set could be made into a concurrent unordered set as well. We also saw that 49% of the execution time was spent in globalRelabel to check that residual[v][w] > 0, and if so, pushing v onto reverseResiduals[w]. This had to be done in a critical section, as multiple threads could be pushing onto w's vector. Lastly, we know that there

is also synchronization overhead from mfence - in the perf report for delaunay_n14, we see that mfence is taking up 32% of the execution time. As we did not explicitly put in an mfence, this mfence is probably due to the synchronization used in the OpenMP commands. It is possible that this indicates that the section before the mfence took a long time, not the mfence itself. However, after looking more closely, the sections before the mfence involve simple loads and additions, so this is probably not the case.

Future Steps

Push-relabel appeared to get better speedup on larger test cases so for the future it would be interesting to represent the input graph instance as adjacency lists to evaluate push-relabel on larger test cases. Additionally, adding adjacency lists to Dinic's improved the performance of the parallel version so we might observe a similar result for push-relabel. As real-world graph algorithms are typically too large to fit on a single machine's memory it would be worthwhile to investigate an MPI implementation of push-relabel and Dinic's.

Our choice of machine target was sound, since there is still a lot of synchronization of threads required in both maxflow algorithms which could prove to be costly for a CUDA implementation. Additionally, mapping a single thread to a unit of work could result in bad load-balancing. With a CPU however, idle threads that finish their work early can work on other tasks.

REFERENCES

- Niklas Baumstark, Guy Blelloch, and Julian Shun. 2015. Efficient implementation of a synchronous parallel push-relabel algorithm. In Proc. ESA.
- Udacity. "Finishing the Parallel BFS with Bags" YouTube, 23 Feb. 2013, https://www.youtube.com/watch?v=M4HSekx-8XA
- Wikipedia contributors. "Maximum flow problem." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 29 Nov. 2019. Web. 10 Dec. 2019.
- Nishant Singh. "Dinic's Algorithm for Maximum Flow " *Geeks for Geeks.* https://www.geeksforgeeks.org/Dinic's-algorithm-maximum-flow/
- 2013 Lecture 14: 15-451/651: Design & Analysis of Algorithms, lecture notes,School of Computer Science 15-451, Carnegie Mellon University, 10 Oct 2013. https://www.cs.cmu.edu/~avrim/451f13/lectures/lect1010.pdf
- Wikihow, WikihowLearn, "Course:Concurrent Programming in CPP" Wikihow, https://en.wikitolearn.org/Course:Concurrent_Programming_in_CPP/TBB_Containers/TBB_Containers
- 2016 Lecture 12: 15-451/651: Network Flows I, lecture notes,School of Computer Science 15-451, Carnegie Mellon University, 22 Feb 2016. http://www.cs.cmu.edu/afs/cs/academic/class/15451-s16/www/lectures/lec12-flow1.pdf
- M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. 24th IEEE International Parallel and Distributed Processing Symposium, 2010.

WORK DISTRIBUTION

Wynne Yao - 50%

Katia Villevald - 50%