15-418 Final Project Report

Michelle Zhang, Michelle Ling
mwzhang, mling2

Parallelizing Network Flow Algorithms

1 Link to Presentation Video

https://bit.ly/3vO4HTD

2 Summary

(a) Project Overview

For our final project, we wrote parallel versions of two common network flow algorithms: Dinic's Algorithm and the Ford-Fulkerson Algorithm. These algorithms are used to solve maximum flow problems on graphs, and when run on large networks, they may not always have optimal performance and would thus benefit from parallelism. We used Open MP to parallelize the two algorithms. To evaluate the performance of our algorithms, we ran our parallelized algorithms on various types of graphs and calculated both total speedup and computational speedup (taking only parallel computation time into account). We also performed analysis on which types of graphs (sparse, average, or dense) performed the best on our algorithms.

3 Background

(a) Introduction to Max Flow Algorithms

Network flows can be used to model a variety of graph problems and are used heavily in operations research and related fields. Max flow has many applications, including image segmentation, modeling traffic, and network connectivity. Flows can be constructed on directed graphs, where each graph has a source node and a sink node. Edges have weights indicating their capacity, or how much flow can be carried from one node to another. In addition to capacities, edges can also have costs, and a network flow problem often involves calculating what the maximum amount of flow that can be carried from the source node to the sink node is while using the minimum cost.

Many problems can be reduced to a network flow problem. One popular problem is bipartite matching, which is itself very applicable to scenarios like matching up applicants and positions, customers and suppliers, etc. It can also be easily translated into a network flow problem by adding dummy source and sink nodes and adjusting the capacities.

(b) Overview of Dinic's Algorithm

(i) Algorithm description

We can eliminate the dependence on the maximum flow entirely with Dinic's algorithm. Dinic's employs a search algorithm (BFS) as a subroutine, which offers some room for parallelization.

(ii) Runtime complexity

Dinic's algorithm has a runtime of $O(mn^2)$, where n is the number of nodes in the graph and m is the number of edges in the graph.

(iii) References for sequential and parallel implementations

For the sequential implementation of Dinic's Algorithm, we used our old Dinic's implementation code from 15-451 and modified it to fit our purposes. For the parallel implementation, we used resources from this class including Open MP guides, lecture and recitation notes, and past homework assignments to guide our approach.

(c) Overview of Ford-Fulkerson Algorithm

(i) Algorithm description

Ford-Fulkerson is a general approach where for a network flow, while there is an augmenting path, we push the maximum possible flow along that augmenting path, and look for another. An augmenting path is a path that has positive capacity, meaning we can push more flow along it. We used the Edmond-Karp implementation, which involves finding the shortest augmenting path. Therefore, we used a breadth-first search subroutine to implementing this shortest path search.

(ii) Runtime complexity

Ford-Fulkerson has a sequential runtime of $O(mF)$, where F is the value of the maximum flow and m is the number of edges in the graph. With larger graph inputs, F could be a very large value, making Ford-Fulkerson pretty inefficient. When the capacities are large, we could potentially be pushing a small amount of flow (and decreasing the capacities by a little bit) each time, which would take many, many iterations.

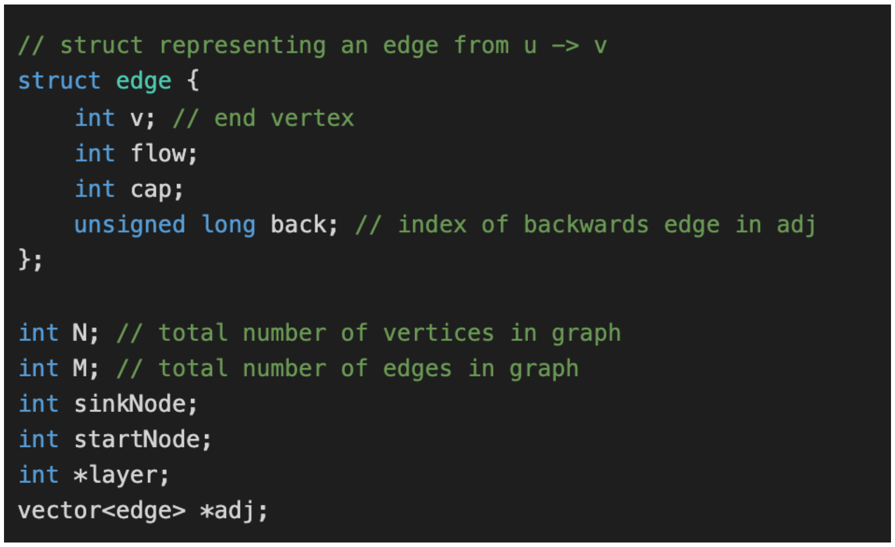
(iii) References for sequential and parallel implementations

Our Ford-Fulkerson algorithm implementation was based on a description from the 15-451 lectures notes (linked in references). We examined a number of papers to learn about different approaches we could take to parallelize it. Most of them said that BFS is the main parallelizable component, with different frameworks to implement it. We also used the OpenMP guide to assist with understanding the framework's directives, as well as course lectures and notes.

4 Approach

(a) Dinic's Algorithm

(i) Data structures



```
// struct representing an edge from u -> v
struct edge {
    int v; // end vertex
    int flow;
    int cap;
    unsigned long back; // index of backwards edge in adj
};

int N; // total number of vertices in graph
int M; // total number of edges in graph
int sinkNode;
int startNode;
int *layer;
vector<edge> *adj;
```

In the Dinic's implementation, we use an adjacency list graph representation. Vertices are represented as ints. The adjacency list graph consists of edges, represented by structs, which contain information about the end vertex, the flow through that edge, the capacity of that edge, as well as the index of the backwards edge in the adjacency list. As we proceed through the algorithm, edges' capacities and flows are updated accordingly.

Global variables that we keep track of are the total number of vertices in the graph (N), the total number of edges (M), as well as the sink node, start node, and a size-N list containing information about which layer a node is located in after residual graphs are updated. This layer list is used in BFS; at the beginning, all nodes are at level -1 besides the source node (at level 0). Then for each node n with parent node at layer with l, if there exists a residual edge between the parent node and n, the node is labeled with layer $l + 1$. The layers list is then read in the `addFlow` function determine which subsequent nodes we can send flow through.

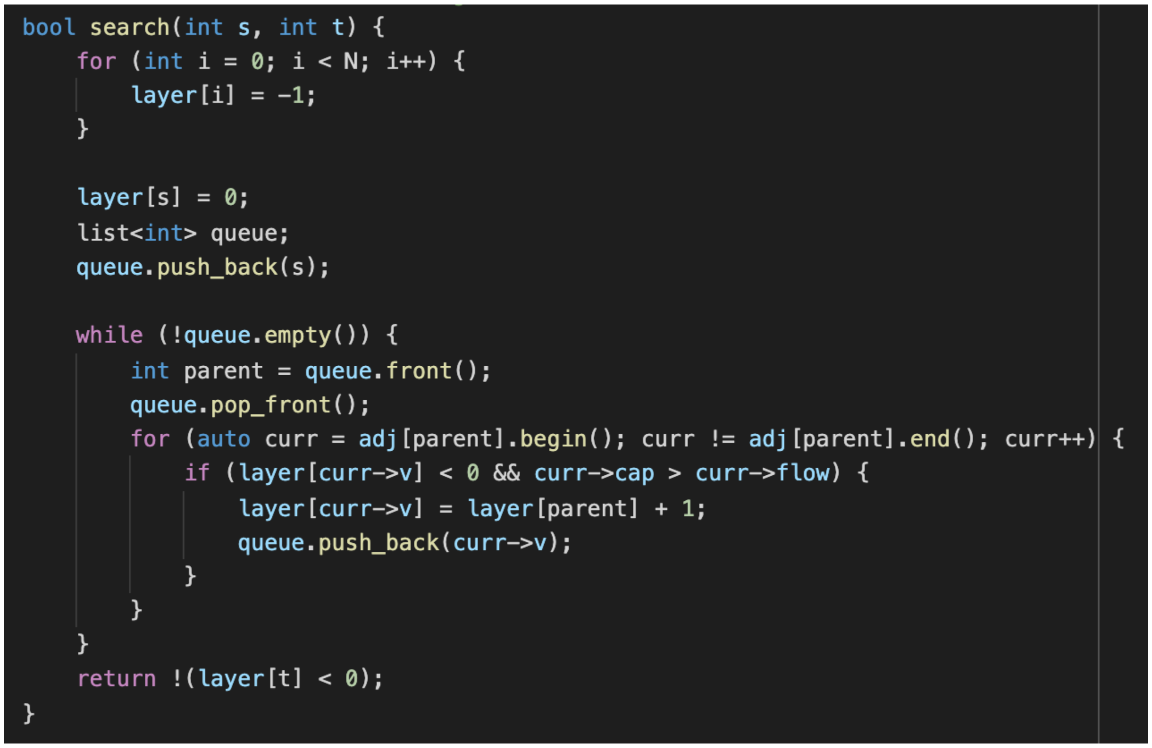
Having an adjacency list graph representation allows us to easily find the neighbors of a node when iterating through the nodes in BFS. Keeping track of the index of the backwards edge is important since the edges are directed, so as Dinic's is run, the information in the forward and backwards edges begins to diverge.

(ii) Sequential implementation notes

In this section, we will break down our Dinic's sequential implementation by describing the major functions and their purposes.

First, we have a function called `createEdge` which takes in two vertices defining an edge (the start and end vertex) as well as the edge's capacity. This function is called during the initialization step to add backwards and forward edges to the graph adjacency list.

Secondly, we have the BFS function `search`. As described above, for each iteration, the BFS function determines if additional flow can be sent from source to sink. Also, this function assigns levels to the vertices when residual graphs are updated. We use a queue data structure in this function. The neighbors of the popped node are iterated through sequentially. Below is a screenshot of the BFS implementation, for easy comparison to the parallel version, which we will discuss in the next section.



```
bool search(int s, int t) {
    for (int i = 0; i < N; i++) {
        layer[i] = -1;
    }

    layer[s] = 0;
    list<int> queue;
    queue.push_back(s);

    while (!queue.empty()) {
        int parent = queue.front();
        queue.pop_front();
        for (auto curr = adj[parent].begin(); curr != adj[parent].end(); curr++) {
            if (layer[curr->v] < 0 && curr->cap > curr->flow) {
                layer[curr->v] = layer[parent] + 1;
                queue.push_back(curr->v);
            }
        }
    }
    return !(layer[t] < 0);
}
```

Next, we have a function called `addFlow`, which updates edge flows after BFS returns that flow is still possible.

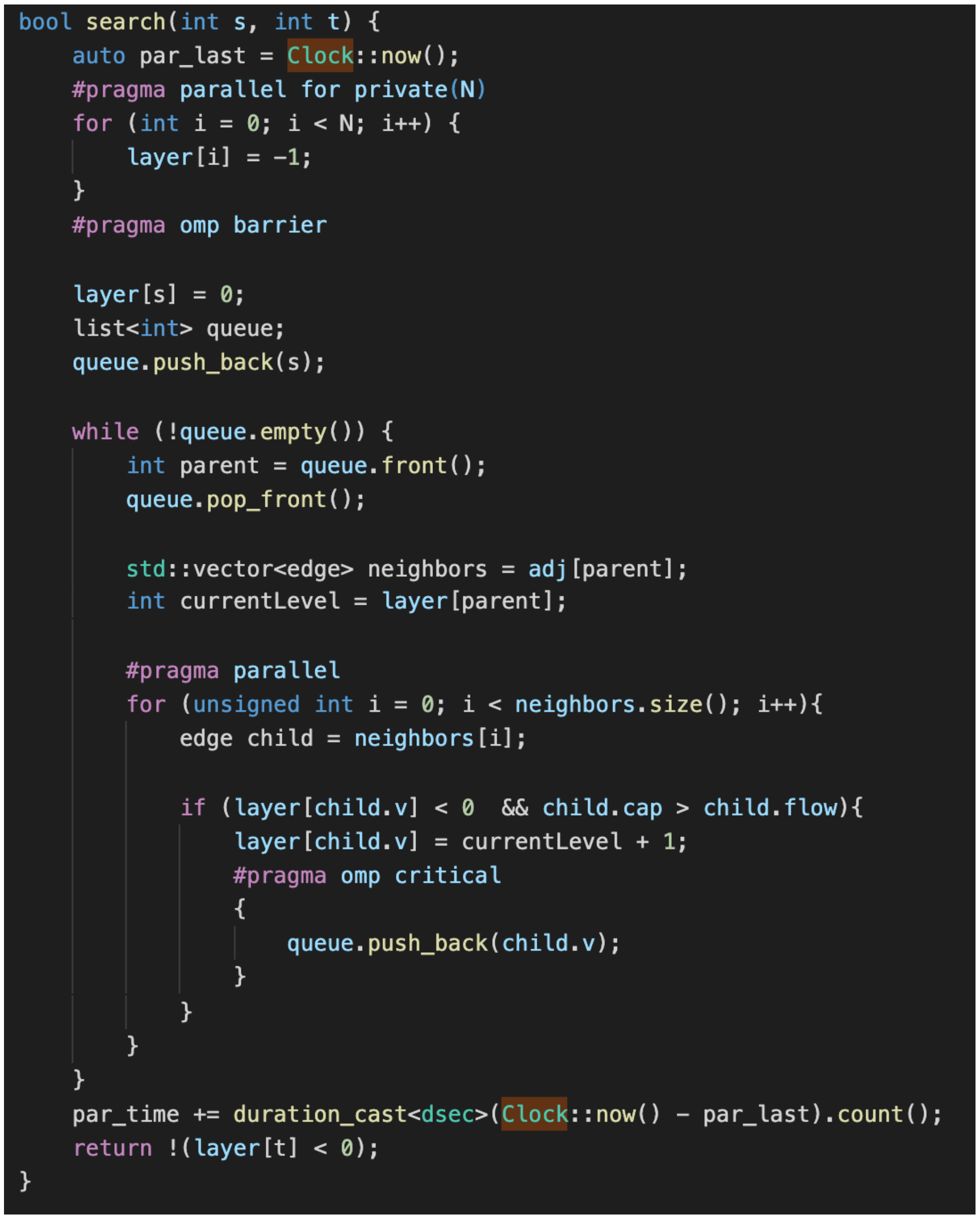
Our main function is `maxFlow`. This function is called by `main`, and it calls the BFS and `addFlow` functions to return the max flow possible in the graph. The resulting max flow is printed.

(iii) Parallel implementation notes

We now move into a discussion of the parallel implementation of Dinic's Algorithm. All the major data structures, including our adjacency list graph implementation, edge struct, and layer list, remain the same as in the sequential implementation.

Parallelizing the search function:

The main changes occur in the `search` function, which is the BFS implementation. Below is the parallel BFS code:



```
bool search(int s, int t) {
    auto par_last = Clock::now();
    #pragma parallel for private(N)
    for (int i = 0; i < N; i++) {
        layer[i] = -1;
    }
    #pragma omp barrier

    layer[s] = 0;
    list<int> queue;
    queue.push_back(s);

    while (!queue.empty()) {
        int parent = queue.front();
        queue.pop_front();

        std::vector<edge> neighbors = adj[parent];
        int currentLevel = layer[parent];

        #pragma parallel
        for (unsigned int i = 0; i < neighbors.size(); i++){
            edge child = neighbors[i];

            if (layer[child.v] < 0  && child.cap > child.flow){
                layer[child.v] = currentLevel + 1;
                #pragma omp critical
                {
                    queue.push_back(child.v);
                }
            }
        }
    }
    par_time += duration_cast<dsec>(Clock::now() - par_last).count();
    return !(layer[t] < 0);
}
```

We used Open MP to parallelize our search code. Instead of iterating through the neighbors sequentially as in the serial implementation, we now look at the neighbors of the popped node in parallel. Note that we have a critical section when we add the child to the search frontier, as having synchronization issues when adding to the queue would lead to incorrect future iterations of BFS.

In addition to parallelizing the search through neighbors, we also parallelize the clearing of the layer list (i.e. when we set every element to -1 at the beginning of the function).

Parallelizing other sections?

We realized that BFS is the main part of Dinic's that can be parallelized. Although the `addFlow` function has a for loop, we cannot parallelize over it because one iteration of the for loop depends on the residual graph of the previous iteration. Thus parallelizing over the for loop could cause correctness issues. In addition, these iterations must be sequential by nature since they have dependencies; sending flow in one iteration depends on previous paths of flows found.

However, the BFS function offered opportunity to parallelize because for each level from the source node, the elements in the queue for that level can be processed independently of the other vertices. We require a synchronization step when adding children to the queue, which makes this part of the code more costly.

We will discuss this challenge more in-depth in our Results section of the report, but since the only part of Dinic's algorithm that could be parallelized was the BFS function, we did not see very interesting results. Additionally, the sequential version is already an optimized algorithm for solving max flow problems, further contributing to the lack of much speedup in our parallel version.

Mapping work to threads:

In the BFS function, Open MP allows the OS to handle mapping the work to threads. Each node in the queue at a level is mapped to an available thread (the nodes currently in the queue are those discovered in the previous search iteration).

(b) Ford-Fulkerson Algorithm

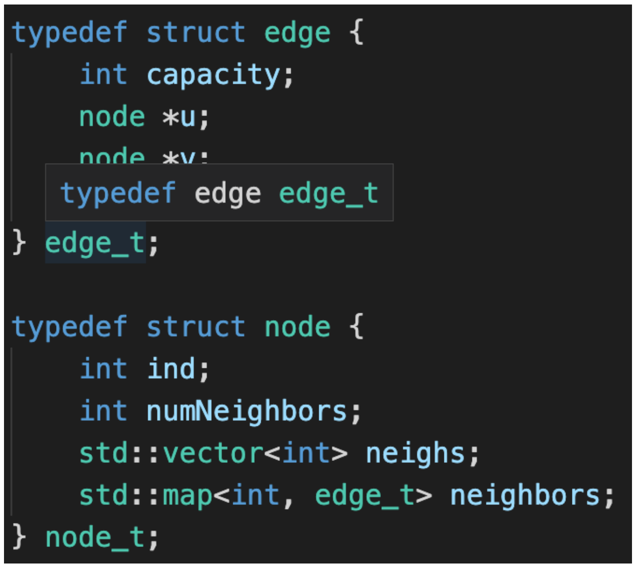
(i) Data structures

i. Adjacency List Graph - We similarly used an adjacency list graph representation for Ford-Fulkerson. Node ids are integers, and each node contains a mapping of a neighbor to the edge between them. This representation was geared towards making it very efficient to gather a node's neighbors. We would expect that the efficiency may be worse on an adjacency matrix implementation, because depending on how we're accessing neighbors, it may go across cache lines with more dense graphs, which would affect performance and speedup.
ii. nodes - We have a variable that keeps track of all of the nodes in the graph in a list, and the nodes keep track of all the information they individually need.
iii. `visited`, `parents` - In the BFS subroutine, these keep track of the nodes that have already been visited, and `parents` maps each node to its parent, so that the augmenting path can later be easily recreated.

(ii) Sequential implementation notes

The implementation is composed three main parts:

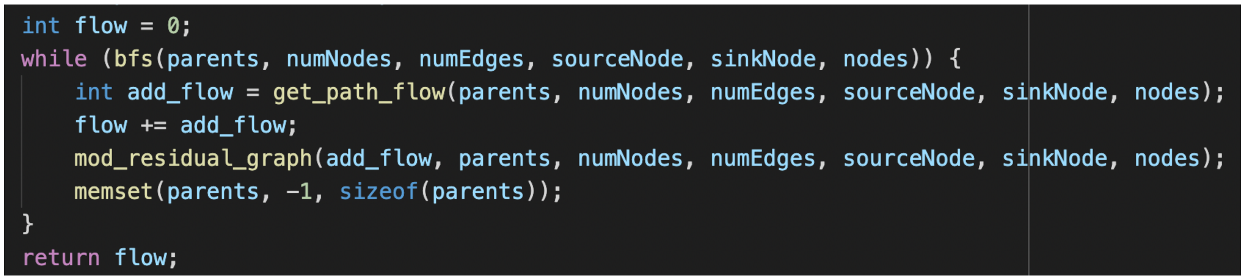
i. `bfs` - This subroutine searches for whether or not there is an augmenting path in the graph. We run this in a `while` loop.
ii. `get_path_flow` - This gets the maximum amount of flow we can push through the augmenting path by tracing through the parents from the bfs implementation and getting the maximum capacity left on the path.



```
typedef struct edge {
    int capacity;
    node *u;
    node *v;
  typedef edge edge_t
} edge_t;

typedef struct node {
    int ind;
    int numNeighbors;
    std::vector<int> neighs;
    std::map<int, edge_t> neighbors;
} node_t;
```

Figure 1: Graph Data Structures



```
int flow = 0;
while (bfs(parents, numNodes, numEdges, sourceNode, sinkNode, nodes)) {
    int add_flow = get_path_flow(parents, numNodes, numEdges, sourceNode, sinkNode, nodes);
    flow += add_flow;
    mod_residual_graph(add_flow, parents, numNodes, numEdges, sourceNode, sinkNode, nodes);
    memset(parents, -1, sizeof(parents));
}
return flow;
```

Figure 2: Main Ford-Fulkerson Loop

iii. `mod_residual_graph` - This modifies the graph to be the residual graph. When we push flow in one direction along an edge, the remaining capacity of that edge is decreased, but the twin edge (the one going in the opposite direction) has its capacity increased by the amount of flow that went through. This function goes through the augmenting path that we found and subtracts on the forward directed edges and adds on the twin edges.

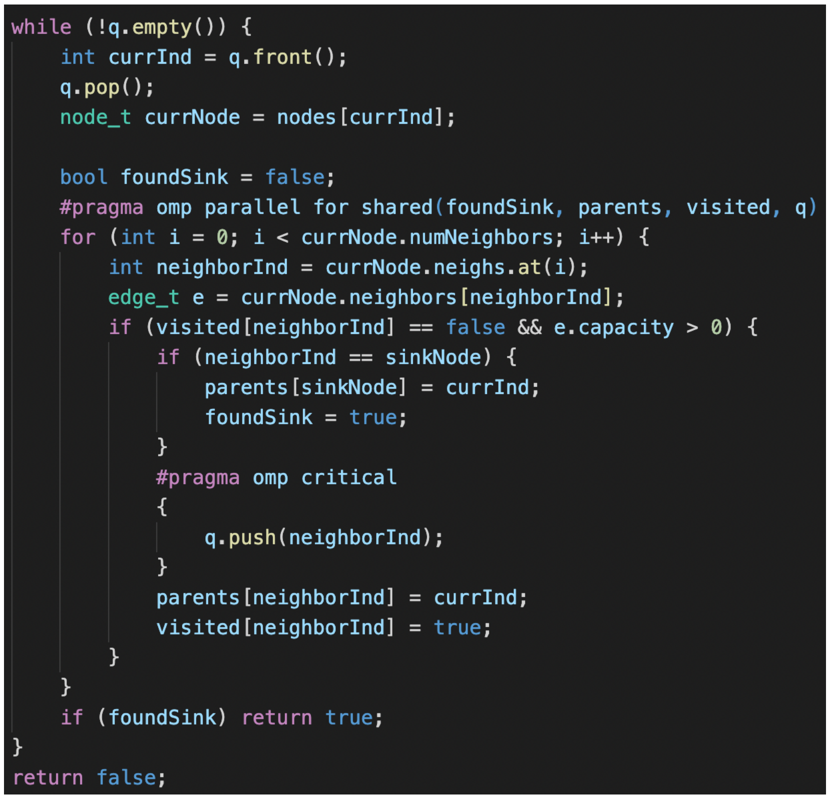
After getting a new residual graph, we will search for another augmenting path. The main loop for Ford-Fulkerson is shown in Figure 2.

(iii) Parallel implementation notes

We now move into a discussion of the parallel implementation of the Ford-Fulkerson algorithm. Examining the entire Ford-Fulkerson algorithm, we saw that we could mainly only parallelize the BFS subroutine.

Parallelizing the search function: The main changes occur in the `bfs` function. The parallel code for the subroutine can be found in Figure 3.

We used OpenMP to parallelize our search code. While processing each element of the queue, we iterate through all neighbors of that element, check whether they've already



```
while (!q.empty()) {
    int currInd = q.front();
    q.pop();
    node_t currNode = nodes[currInd];

    bool foundSink = false;
    #pragma omp parallel for shared(foundSink, parents, visited, q)
    for (int i = 0; i < currNode.numNeighbors; i++) {
        int neighborInd = currNode.neighs.at(i);
        edge_t e = currNode.neighbors[neighborInd];
        if (visited[neighborInd] == false && e.capacity > 0) {
            if (neighborInd == sinkNode) {
                parents[sinkNode] = currInd;
                foundSink = true;
            }
            #pragma omp critical
            {
                q.push(neighborInd);
            }
            parents[neighborInd] = currInd;
            visited[neighborInd] = true;
        }
    }
    if (foundSink) return true;
}
return false;
```

Figure 3: Parallelized BFS Subroutine

been visited, and make some updates and possibly add them into the queue again. We were able to look at different neighbors in parallel by assigning a thread to a neighbor to process. We used a dynamic scheduler, which means that once a thead finishes processing one neighbor, we can assign it to a new one. We tried a variety of different schedulings such as block and interleave static assignments, but these did not fare as well (we talk more about this in the Other Approaches section).

Parallelizing other sections

We figured out that we could not parallelize the other portion of BFS, which is processing elements in the queue. Since we are looking for the shortest path, we could not process the queue elements out of FIFO order. Modifying the residual graph in parallel would not save much computation time as we had to iterate through it to find the path anyways, and doing another pass through in parallel to update the path may add unnecessary overhead to a computationally light portion of the algorithm anyways.

We also made an MPI implementation for Ford-Fulkerson, but, as we will explain further in the Other Approaches section, this algorithm inherently involved a lot of sharing data, and so our preliminary MPI results didn't look very promising, and our code involved many rounds of message passing as well (e.g. new neighbors that were found had to all still go into one centralized queue).

Therefore, we were limited by Amdahl's law, stating that the fraction of time on parallelizable work limits the amount of speedup we can get.

5 Results & Analysis

5.1 Experimental Setup & Measurements

5.1.1 Inputs/Outputs

Graphs were passed in with the following format:

```
    <num of nodes> <num of edges> <source node id> <sink node id>
    <out-node id> <in-node id> <edge capacity>
    ...
```

The number of edges would tell us how many edges to read in. The graph would be located in a text file, and the command line argument would be the path to the text file.

We could also optionally pass in the number of threads that we want the program to run with as a command line argument, or we directly modified a global constant `NUM_THREADS`.

The output, the maximum flow value, would be returned at the end. In addition, we outputted various timing results to help us do speedup calculations.

5.1.2 Measurements Procedure

To measure performance, we used the Clock in the standard chrono library. We took two types of measurements: total time and computation time. The total time consisted of how long the program took in its entirety, from as soon as the main function is called to after the result is returned. The computation time is the portion that was parallelizable, namely the breadth-first search functions. Since the BFS function is called several times, we made sure to add to the computation time in increments. A measurement would start right before BFS is called, and then we took an interval to right after it finishes executing. The rest of the computation, such as creating the residual graph in Ford-Fulkerson, were excluded from the computation time measurement.

We ran the parallel OpenMP algorithm using 1 thread for the baseline measurements, and also 2, 4, and 8 threads on the 8-core GHC machines. We did not compare performance with our sequential implementations as they were implemented with slight modifications and ended up being faster than the 1-thread parallel implementation. We used the GHC machines to perform all of the experiments. We attempted to run the tests with 16 and 32 threads as well but we found that in all inputs, the performance got worse at these high thread counts. We will discuss this behavior in the Discussion subsection, but we decided the reasonable speedup results could be best captured with at most 8 threads.

We ran tests using a variety of types of graphs to capture many different use cases of the Ford-Fulkerson and Dinic's algorithms and how efficient each case would be. We wrote and ran a graph generating script to generate all of our graphs.

Graph File Name	# Vertices	# Edges	Density
small_a	5	5	1
small_b	6	10	1.67
med_a	100	400	4
med_b	1000	100000	100
avg_a	24375	51718	2.12
avg_b	27692	61354	2.22
avg_c	54318	91160	1.67
dense_a	5253	94080	17.9
dense_b	1000	97234	97.2
dense_c	4000	1000000	250
dense_d	4000	2000000	500
sparser_a	20000	1000	0.05
sparser_b	20000	10000	0.5
sparser_c	8000	50000	6.25
denser_a	1000	400000	400
denser_b	5000	1000000	200
denser_c	2000	1950000	975

Table 1: Input Graphs

We measured the graph density with the average number of edges coming out of or into each node. We started with some sparse graphs with less than 100 edges per node (this is also dependent on graph size), to hundreds and up to 1000 edges per node.

The problem size was all varied across different inputs, with small graphs that had less than 50 nodes and edges, to larger graphs that hundreds to a few thousand nodes, and anywhere from thousands to millions of edges.

Table 1 shows is a list of properties for the graphs that we used for the Dinic's and Ford-Fulkerson algorithms.

5.2 Dinic's Algorithm

5.2.1 Results

The graphs showing the relationship between performance speedup and times versus the number of threads as well the graph density can be see in Figures 4, 5, and 6.

The data tables can be seen in Figure 7.

5.2.2 Discussion

Problem Size: For the parallel Dinic's Algorithm, in general, problem size was not correlated with speedup performance. Looking at Figure 4, we can see that the small graph tests had about the same speedup as (if not better speedup than) some of the larger graphs, such as *sparser_a*.

Problem size was somewhat correlated with computation time, though, as can be seen in Figure 5. The *dense_d* graph test took significantly longer than the other graphs, since it has much more

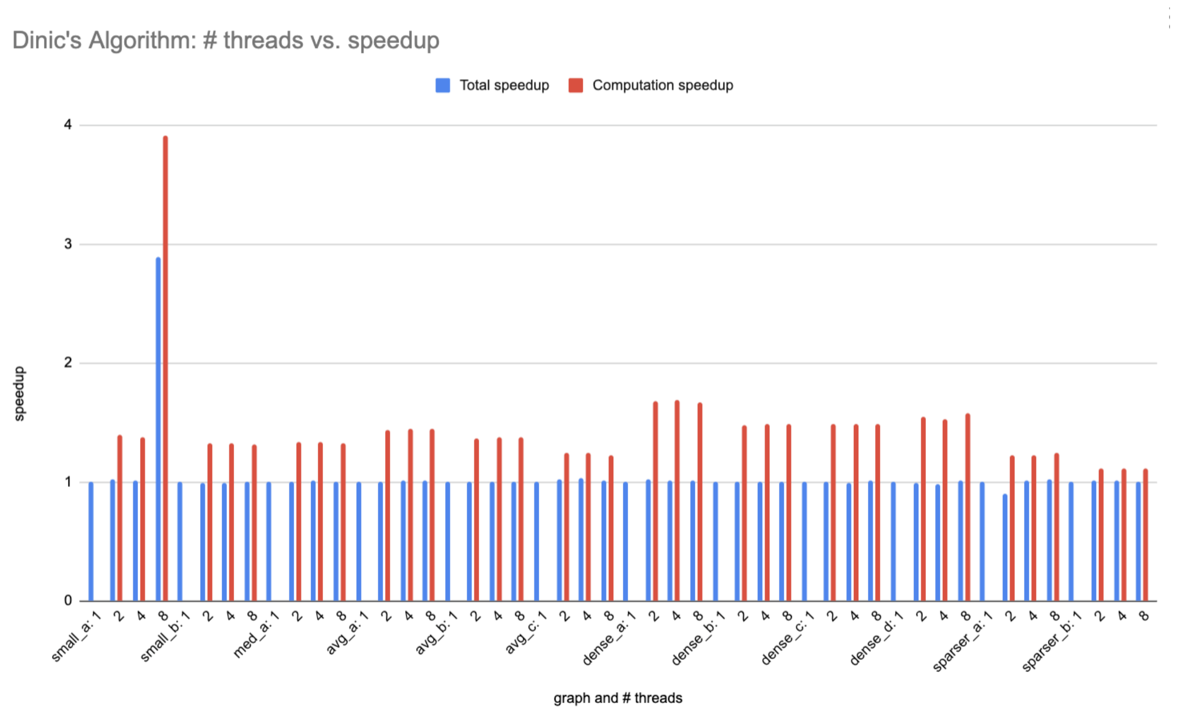


Figure 4: Number of Threads vs. Total and Computation Speedup

edges than the others. However, although there was a rough relationship between problem size and computation time, the correlation is not exact, as can be seen by the fact that *avg_a* took longer to compute than *avg_c*, even though *avg_c* is a bigger graph.

Graph Density: Looking at Figure 6, we can see that denser graphs lead to much better speedup than average or sparse graphs. This result is consistent no matter how many threads we are using. As density increases, speedup increases because there are on average more neighbors visited per node, so there is more parallelizable work available. This overshadows the overhead required to assign threads to nodes. In addition, since parallel computation time is a significant portion of the total time, having BFS be parallelized is efficient, as it allows this increase in work to be distributed among threads, thus improving speedup.

5.3 Ford-Fulkerson Algorithm

5.3.1 Results

The graphs showing the relationship between performance speedup and times versus the number of threads as well the graph density can be see in Figures 8 and 9.

The data tables can be seen in Figure 10.

5.3.2 Discussion

Problem Size: We saw that the problem size didn't impact the speedup too much. Although smaller graphs with less vertices didn't have as much capability to more dense, since their maximum number of neighbors wasn't that much anyways, so in general, the smaller graphs all had pretty mediocre speedup, where the multi-threaded implementations actually performed worse than if

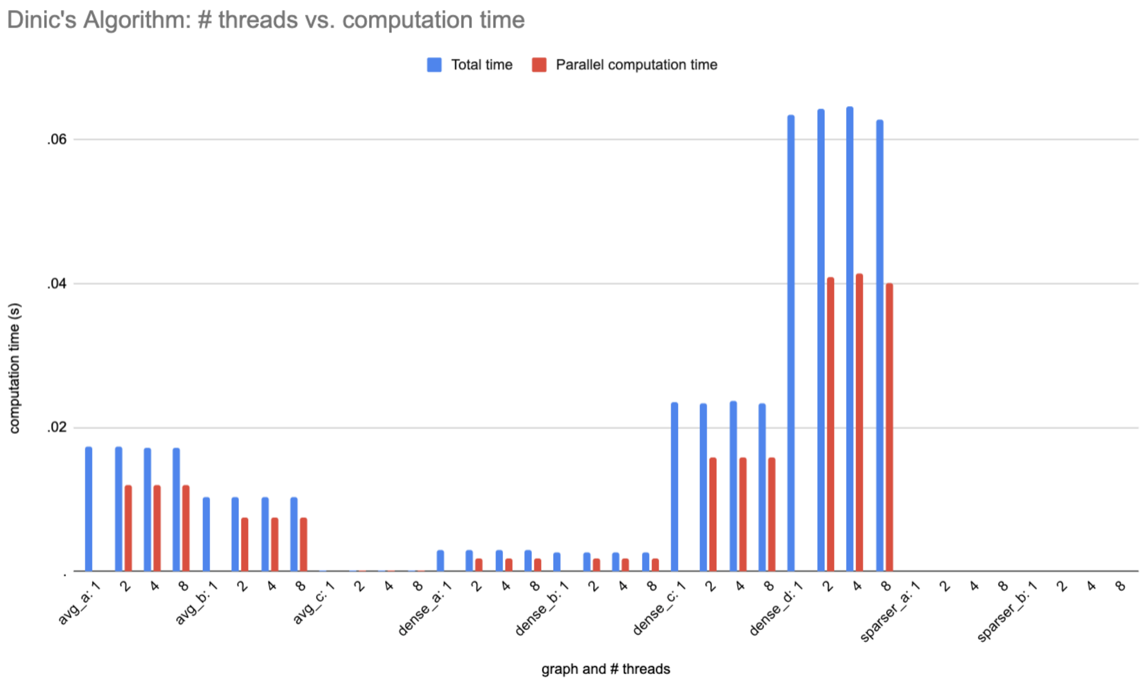


Figure 5: Number of Threads vs. Total and Computation Speedup
(Note that for the *avg_c*, *sparser_a*, and *sparcer_b* test cases, the total and parallel computation time was very small.)

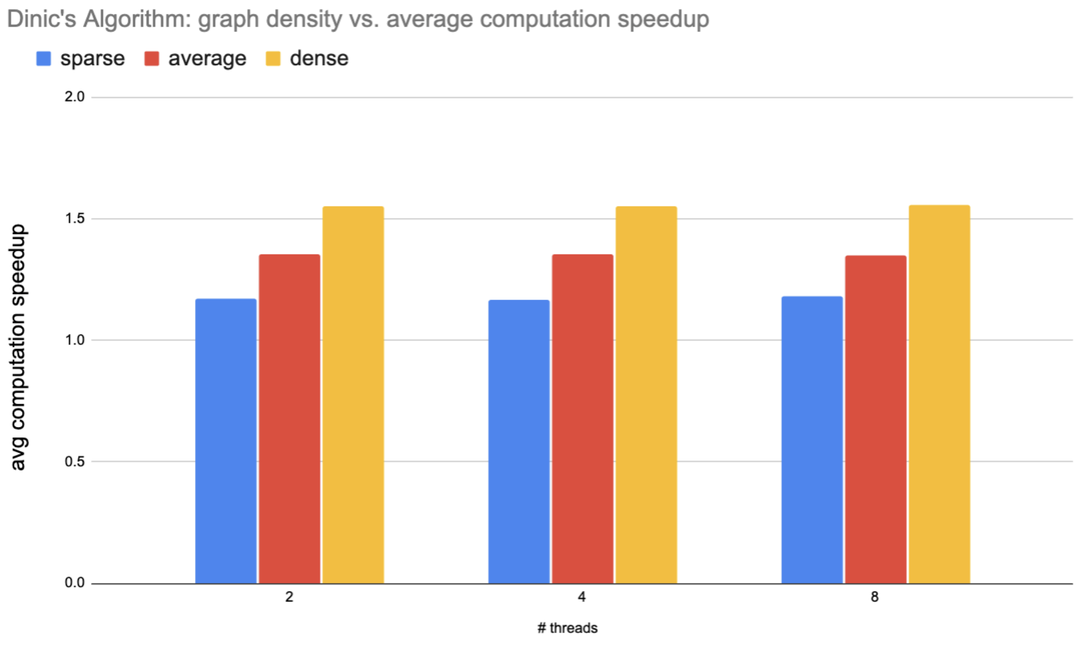
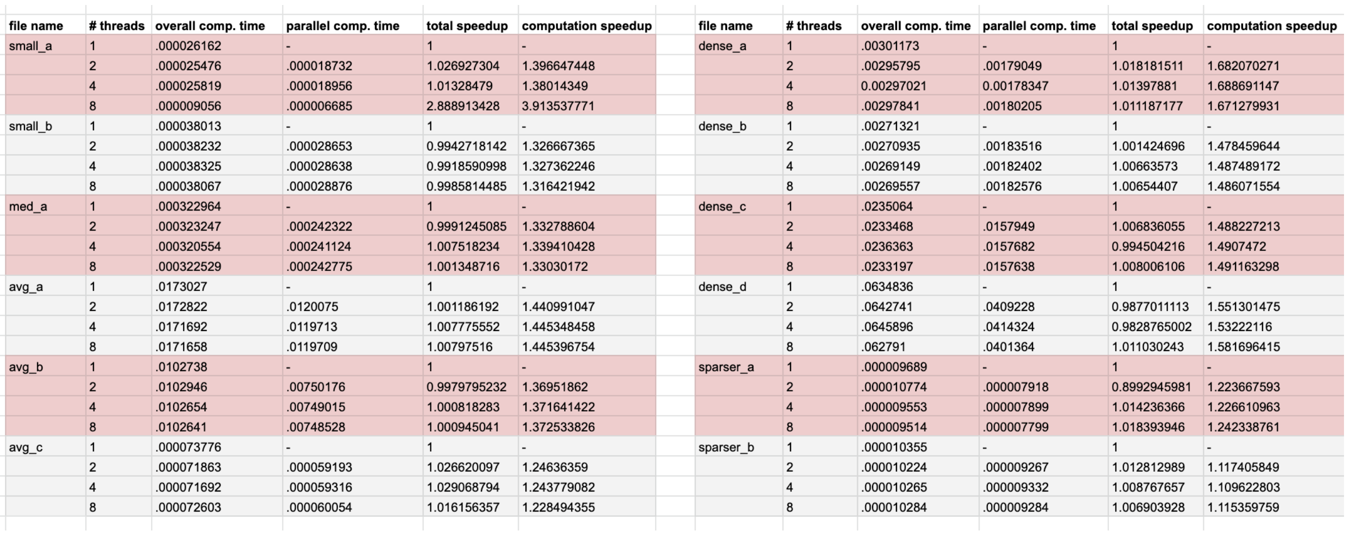


Figure 6: Graph Density vs. Average Computation Speedup



file name	# threads	overall comp. time	parallel comp. time	total speedup	computation speedup
small_a	1	.000026162	-	1	-
	2	.000025476	.000018732	1.026927304	1.396647448
	4	.000025819	.000018956	1.01328479	1.38014349
	8	.000009056	.000006685	2.888913428	3.913537771
small_b	1	.000038013	-	1	-
	2	.000038232	.000028653	0.9942718142	1.326667365
	4	.000038325	.000028638	0.9918590998	1.327362246
	8	.000038067	.000028876	0.9985814485	1.316421942
med_a	1	.000322964	-	1	-
	2	.000323247	.000242322	0.9991245085	1.332788604
	4	.000320554	.000241124	1.007518234	1.339410428
	8	.000322529	.000242775	1.001348716	1.33030172
avg_a	1	.0173027	-	1	-
	2	.0172822	.0120075	1.001186192	1.440991047
	4	.0171692	.0119713	1.007775552	1.445348458
	8	.0171658	.0119709	1.00797516	1.445396754
avg_b	1	.0102738	-	1	-
	2	.0102946	.00750176	0.9979795232	1.36951862
	4	.0102654	.00749015	1.000818283	1.371641422
	8	.0102641	.00748528	1.000945041	1.372533826
avg_c	1	.000073776	-	1	-
	2	.000071863	.000059193	1.026620097	1.24636359
	4	.000071692	.000059316	1.029068794	1.243779082
	8	.000072603	.000060054	1.016156357	1.228494355

file name	# threads	overall comp. time	parallel comp. time	total speedup	computation speedup
dense_a	1	.00301173	-	1	-
	2	.00295795	.00179049	1.018181511	1.682070271
	4	0.00297021	0.00178347	1.01397881	1.688691147
	8	.00297841	.00180205	1.011187177	1.671279931
dense_b	1	.00271321	-	1	-
	2	.00270935	.00183516	1.001424696	1.478459644
	4	.00269149	.00182402	1.00663573	1.487489172
	8	.00269557	.00182576	1.00654407	1.486071554
dense_c	1	.0235064	-	1	-
	2	.0233468	.0157949	1.006836055	1.488227213
	4	.0236363	.0157682	0.994504216	1.4907472
	8	.0233197	.0157638	1.008006106	1.491163298
dense_d	1	.0634836	-	1	-
	2	.0642741	.0409228	0.9877011113	1.551301475
	4	.0645896	.0414324	0.9828765002	1.53222116
	8	.062791	.0401364	1.011030243	1.581696415
sparser_a	1	.000009689	-	1	-
	2	.000010774	.000007918	0.8992945981	1.223667593
	4	.000009553	.000007899	1.014236366	1.226610963
	8	.000009514	.000007799	1.018393946	1.242338761
sparser_b	1	.000010355	-	1	-
	2	.000010224	.000009267	1.012812989	1.117405849
	4	.000010265	.000009332	1.008767657	1.109622803
	8	.000010284	.000009284	1.006903928	1.115359759

Figure 7: Data for Dinic's Algorithm

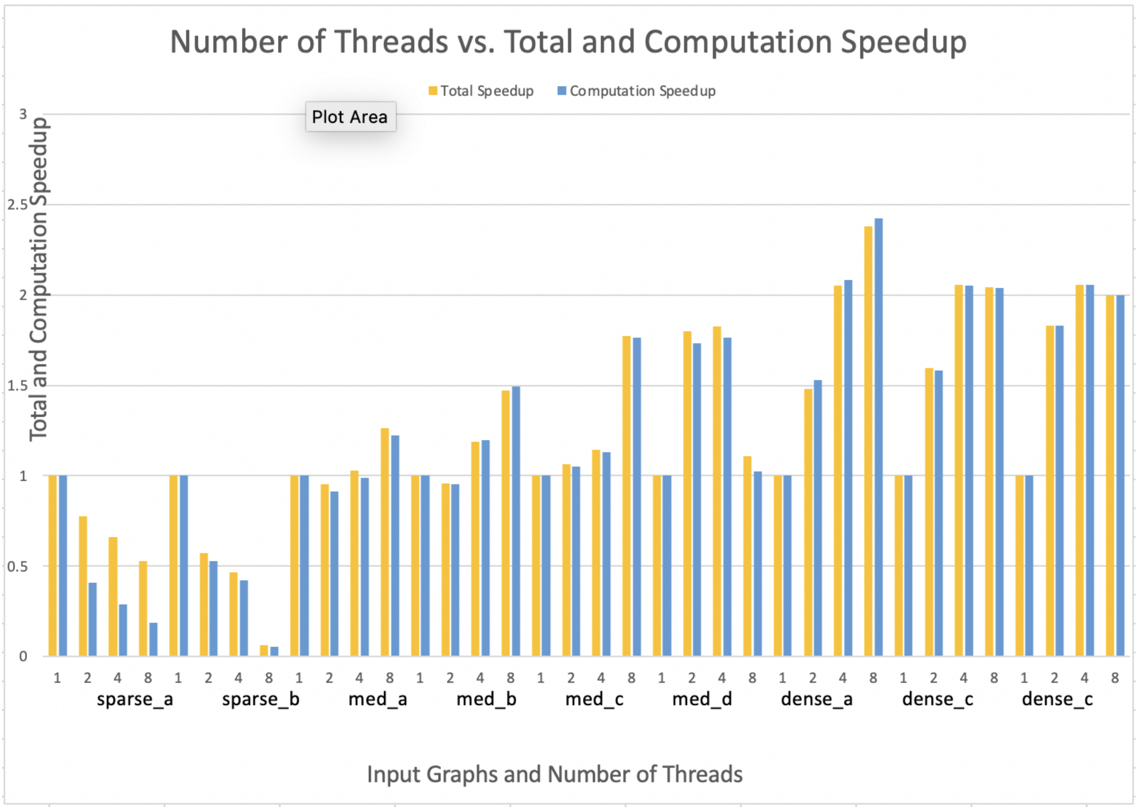


Figure 8: Number of Threads vs. Total and Computation Speedup

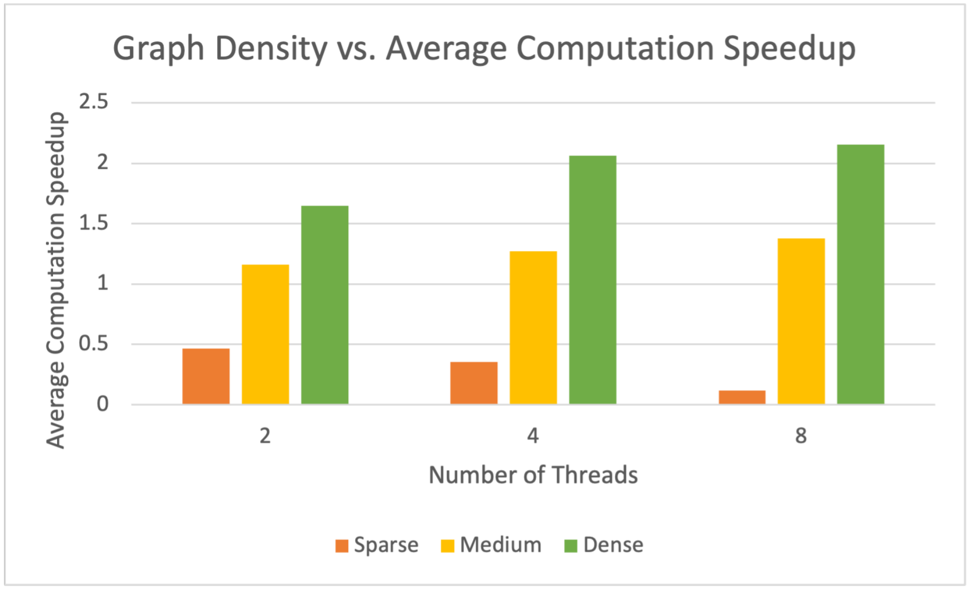


Figure 9: Graph Density vs. Average Computation Speedup

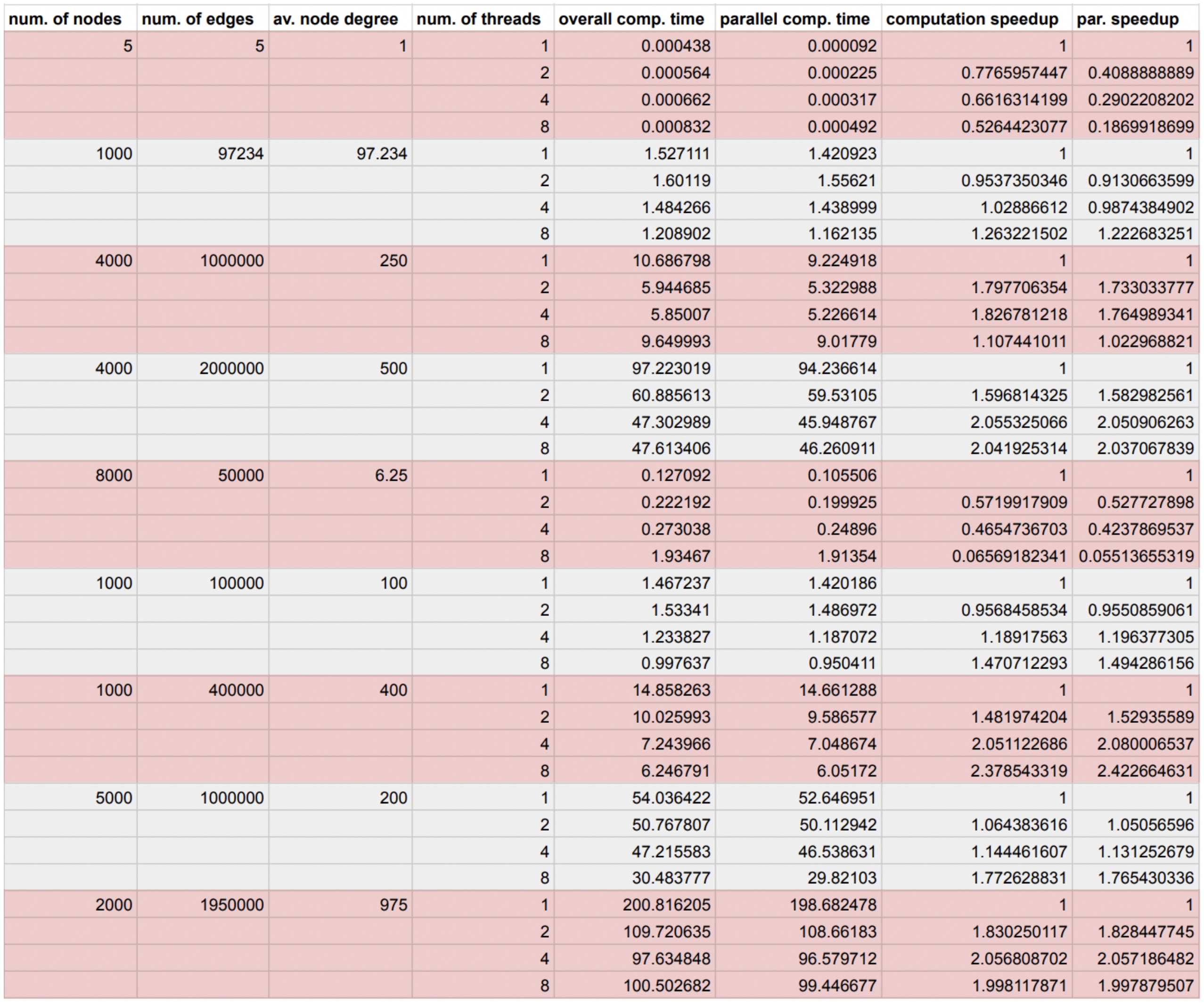
we just used one thread. This is likely due to the much larger role that overhead for scheduling threads plays in smaller graphs. In Figure 8, we see that the first two graphs of smaller size have speedup of less than 1 for the total and computation speedup. Since we are parallelizing by assigning threads to handle different neighbors, the overhead of assigning a thread to a neighbor will be proportionally larger with a smaller graph.

Additionally, we need to sequentially update the queue with the neighbors that we're looking through, which means that we need to lock the queue resource before and unlock it after each thread is adding its neighbor. Transferring control across each thread involves overhead that can take longer than just a sequential implementation with one thread.

Graph Density: We saw that dense graphs led to much higher speedups than sparse graphs. For instance, on our most dense graph (denser_c with a density of 975), we were able to get slightly more than a 2x speedup with 4 threads. With the 400 density graph, we saw that speedup on 8 threads got up to 2.5x speedup. We can explain this increase in speedup as density increases because there is on average more neighbors per visited node. This means that there is more parallelizable work, so the overhead of assigning threads to neighbors to process will be proportionally less. Additionally, we can see that the computation time is a large majority of the total time, so, especially with dense graphs, the parallelizable portion, the BFS traversal, is taking the most time in the program.

5.3.3 Between Threads

We generally saw the best speedup to thread ratio around 4 threads. We also did not go past 8 threads due to the massive amounts of overhead decreasing the amount of speedup that we got. We were able to get above 2x in some cases for dense graphs with 4 threads. Between 4 and 8 threads, the speedup generally did not improve that much, which suggests that overhead is already playing a large part and not as worth it to use 8 threads. Additionally, in smaller graphs, there may not even be 8 neighbors for one node, so that would explain the 8 thread speedups not doing much better on average. The 2-thread experiments did alright for dense graphs, and we saw a bigger



num. of nodes	num. of edges	av. node degree	num. of threads	overall comp. time	parallel comp. time	computation speedup	par. speedup
5	5	1	1	0.000438	0.000092	1	1
			2	0.000564	0.000225	0.7765957447	0.4088888889
			4	0.000662	0.000317	0.6616314199	0.2902208202
			8	0.000832	0.000492	0.5264423077	0.1869918699
1000	97234	97.234	1	1.527111	1.420923	1	1
			2	1.60119	1.55621	0.9537350346	0.9130663599
			4	1.484266	1.438999	1.02886612	0.9874384902
			8	1.208902	1.162135	1.263221502	1.222683251
4000	1000000	250	1	10.686798	9.224918	1	1
			2	5.944685	5.322988	1.797706354	1.733033777
			4	5.85007	5.226614	1.826781218	1.764989341
			8	9.649993	9.01779	1.107441011	1.022968821
4000	2000000	500	1	97.223019	94.236614	1	1
			2	60.885613	59.53105	1.596814325	1.582982561
			4	47.302989	45.948767	2.055325066	2.050906263
			8	47.613406	46.260911	2.041925314	2.037067839
8000	50000	6.25	1	0.127092	0.105506	1	1
			2	0.222192	0.199925	0.5719917909	0.527727898
			4	0.273038	0.24896	0.4654736703	0.4237869537
			8	1.93467	1.91354	0.06569182341	0.05513655319
1000	100000	100	1	1.467237	1.420186	1	1
			2	1.53341	1.486972	0.9568458534	0.9550859061
			4	1.233827	1.187072	1.18917563	1.196377305
			8	0.997637	0.950411	1.470712293	1.494286156
1000	400000	400	1	14.858263	14.661288	1	1
			2	10.025993	9.586577	1.481974204	1.52935589
			4	7.243966	7.048674	2.051122686	2.080006537
			8	6.246791	6.05172	2.378543319	2.422664631
5000	1000000	200	1	54.036422	52.646951	1	1
			2	50.767807	50.112942	1.064383616	1.05056596
			4	47.215583	46.538631	1.144461607	1.131252679
			8	30.483777	29.82103	1.772628831	1.765430336
2000	1950000	975	1	200.816205	198.682478	1	1
			2	109.720635	108.66183	1.830250117	1.828447745
			4	97.634848	96.579712	2.056808702	2.057186482
			8	100.502682	99.446677	1.998117871	1.997879507

Figure 10: Data for Ford-Fulkerson's Algorithm

jump from 2 to 4 threads in dense graphs.

5.4 Discussion

5.4.1 Challenges

Dinic's Algorithm: As discussed before, parallelizing Dinic's Algorithm was challenging due to the fact that a large part of the algorithm, the `addFlow` function that updates possible flows after each BFS iteration, is inherently iterative. Thus, we could not parallelize this section and could only parallelize the BFS function.

Ford-Fulkerson Algorithm: Similarly, most of Ford-Fulkerson is iterative as well, so it was difficult to find parts of the algorithm that would result in massive speedups. The speedup results that we have are pretty expected. Comparing our sequential and 1-thread parallel implementations for Ford-Fulkerson, we also see an increase in total and computation time for the 1-thread version, showing that OpenMP does have an amount of overhead even on one thread. However, it isn't as involved as some of the other frameworks (e.g. MPI) so that was one of the reasons we decided to use it. Nevertheless, overhead on higher thread counts proved to be a challenge for our speedup values.

5.4.2 Algorithms Comparison

Dinic's tradeoffs: Based on our analysis, we found that sequential Dinic's runs quickly on sparse graphs and much slower on larger, denser graphs (i.e. computation time of Dinic's is faster for sparser graphs). However, the speedup achieved from parallelizing Dinic's seems to benefit denser graphs more than sparser graphs. This is an interesting conclusion that shows that different algorithms may run better on different types of graphs, depending on implementation approach.

Ford-Fulkerson: We found that Ford-Fulkerson is much faster on sparser graphs and slower on larger, denser graphs. The speedup achieved through parallelizing FF benefits denser graphs over sparser graphs as well. This is likely due to the fact that the basis of parallelizing both functions is parallelizing the BFS subroutine.

Looking at the speedup on the same test cases for both functions, we see that there is slightly more variability with the Ford-Fulkerson speedups, while the Dinic's speedups are fairly consistent. This is likely due to the more involved non-BFS computations in Dinic's, as we can also see that the proportion of time dedicated to the BFS computation is higher in the FF (almost all of it) algorithm than the Dinic's algorithm. Therefore, with a proportionally higher parallelizable component, it allows the overall speedup to vary more.

5.4.3 Speedup Analysis

Dinic's Speedup Analysis:
There are many likely factors that contributed to the less-than-ideal speedup results with our parallel Dinic algorithm. Some possible explanations include:

- With smaller test graphs, there is more overhead that occurs, thus overshadowing any possible benefit that parallelism presents. When there is not enough useful work to distribute, scheduling work to threads is not an efficient use of resources. This leads to worse performance and therefore less-than-ideal speedup.

- We did not implement a granularity limit. That is, for all problem sizes, we ran the parallel algorithm on the graph even if it was not the most efficient approach. This means that small graphs again incur a lot of overhead, which worsens overall performance. Implementing a granularity limit to control for smaller graphs would likely improve speedup.

- For sparse graphs, parallelism does not present much benefit. This is because most parallelism occurs in the BFS function, which helps alleviate the amount of sequential iteration through neighbors. However, if nodes do not have many neighbors (as in the case of sparse graphs), then parallel BFS has little effect on performance improvements. Thus this explains less-than-ideal speedup for sparse graphs.

- Our Open MP approach involves having many shared structures, which can be costly and reduce performance. Some of these shared data structures include the the layer list, the BFS queue, and lists of parents/children.

- Since we could only implement the BFS subroutine of Dinic's, Amdahl's law says that the resulting parallel algorthm will only be as fast as the sequential portion of the algorithm. Thus, for graphs where paralellizing BFS has little to no value and where the `addFlow` function takes up a majority of computation, we are limited by the fact that the `addFlow` function must be sequential.

Ford-Fulkerson Speedup Analysis:
There are many likely factors that contributed to the less-than-ideal speedup results for our parallel Ford-Fulkerson algorithm. Some possible explanations include:

- We made an effort to balance the workload between threads by using OpenMP's dynamic scheduling directive. The amount of work that each thread is assigned to is basically the same. However, due to Amdahl's law, the amount of parallelizable work is highly limited, which limits the amount of speedup we can achieve. Amdahl's law mentions that the amount by which a program can speed up overall is limited by the fraction of time that is actually able to be parallelizble. Even though the BFS component is a large part of the overall duration of the algorithm, we could not fully parallelize the entire BFS function, and so some of this computation is not parallelizable. For instance, accessing the queue to add new neighbors had to be done sequentially. Processing the queue had to be done sequentially as well, so even the computation time measurement includes some inherently sequential parts, limiting the computation speedup numbers as well.

- We did not have a granularity limit, so even when the number of neighbors on a node was less than the number of threads, we still assigned threads to neighbors. This resulted in a proportionally larger amount of overhead in the smaller and sparser graphs, and was seen especially with the 8 thread experiments.

- Graph density played a large role in gaining more speedup on larger graphs. Sparse graphs have nodes that simply do not have many neighbors. This proportionally decreases the amount of parallelizable work we are able to do, so we are limited by Amdahl's law even further.

- We need to share the `visited` and `parents` and centralized queue data structures, making it difficult to parallelize even across neighbors. Notably, we have a `critical` directive for the section that adds neighbors into the queue. This makes it difficult to let threads do a lot of work on their own because a graph structure is inherently tightly coupled.

6 Alternative Approaches

(a) MPI - Another approach we were considering was the parallelizing the BFS subroutine using MPI. Since most graph algorithms are typically too large to fit on a single machine's memory, if we had more time and guidance, it would be useful to get an efficient MPI implementation of both algorithms. Our current MPI code can be found in the `ford_fulkerson.cpp` file. However, as we saw in the OpenMP implementations, the threads that are mapped to neighbors share a lot of data, so our implementation required a lot of message passing between a master thread and the rest subservient threads. Some preliminary experiments did not show much speedup, so we didn't continue with this approach.

(b) Granularity - In addition, to improve speedup, we could work on experimenting with different granularity levels in our implementations – i.e. if problem size is smaller than a certain threshold, run the sequential algorithm instead of the parallel algorithm. This would reduce the amount of overhead that occurs when running the parallel algorithms on small graphs that do not have enough useful work to distribute to multiple threads.

(c) Random Graph Assignments - The input graphs that we used were all randomly generated, so the number of neighbors and locations of neighbors for each node was also randomly distributed. We tried to exploit blocking or interleaving assignments to gain a better speedup, but this random structure preventing us from getting any meaningful results. It would be interesting to try some input graphs with a lot of locality and make some static block/interleave assignments rather than dynamic scheduling, like we did here.

(d) Push-Relabel - We also made an implementation of Push-Relabel (in the `pr` directory). However, we wanted to compare the speedup across more similar algorithms, so we decided not to continue with this one. We did see that there may be more room for parallelization in Push-Relabel, as nodes can operate more independently, so it would also be interesting to try this out.

7 References

- https://www.cs.cmu.edu/ 15451-s22/lectures/lec10-flow1.pdf (FF Description)
- https://www.cs.cmu.edu/ 15451-s22/lectures/lec11-flow2.pdf (Dinic's)
- https://pdfs.semanticscholar.org/c4fb/713ed6b41672dc51782513062cd470d979c3.pdf (Edmonds-Karp) http://worldcomp-proceedings.com/proc/p2013/PDP3767.pdf (Ford-Fulkerson)
- https://iq.opengenus.org/push-relabel-algorithm/ (Push-Relabel)
- http://web.mit.edu/ neboat/www/presentations/spaa2010.pdf (BFS)
- https://thesai.org/Publications/ViewPaper?Volume=8&Issue=6&Code=IJACSA&SerialNo=20

8 Work Distribution

Michelle, Michelle - 50%, 50%

Michelle Ling - Ford-Fulkerson sequential and parallel implementations, ran FF experiments

Michelle Zhang - Dinic's sequential and parallel implementations, ran Dinic's experiments

Both - Discussed parallelization approaches for both algorithms, discussed and worked on alternative approaches (MPI, PR)