## 0 Project information

- Project name: parallel maximum flow
- Team members: Siyuan Chen, Xinyue Yang
- URL: `https://xinyue-yang.github.io/parallel-maxflow/`

## 1 Summary

We parallelized two maximum flow algorithms (Edmonds-Karp and Dinic's) under the shared address space model using OpenMP. We evaluated their performance on GHC and PSC machines against different network types. We demonstrated that (1) across the two algorithms, the former is more parallelizable but overall the latter is more performant; and (2) within each algorithm, the top-down and bottom-up parallelism approaches are more suitable for sparse and dense networks, respectively.

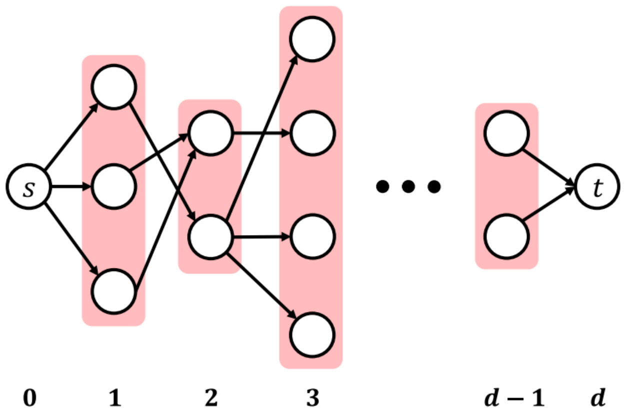


Diagram is from 15-451 lectures notes.

## 2 Background

### 2.1 Maximum flow

Flow is widely used to model constrained resource distribution problems. The input to the flow problem consists of a directed graph $G$; among the vertices are the source $s$ and the sink $t$; each edge also has an associated nonnegative capacity $c(u, v)$. The output to the problem is an assignment of flow to each edge such that the following constraints are satisfied:

- capacity constraint: the flow for each edge is nonnegative and at most the capacity of the edge; and
- flow conservation: for every vertex other than the source and the sink, the incoming flow equals the outgoing flow.

Several categories of flow problems exist: maximum flow, minimum cost maximum flow, etc. We choose to focus on maximum flow as it is widely applicable in many real life problems, such as airline scheduling, preference matching, and image segmentation. Specifically, we choose to focus on two augmenting-path-based maximum flow algorithms: Edmonds-Karp and Dinics.

### 2.2 Augmenting-path-based maximum flow algorithms

We first present the high-level idea of augmenting-path-based maximum flow algorithms.

Fix a network and a flow. For each edge, consider the flow currently assigned: given this flow's presence, we can only push less flow in the same direction, but more flow in the opposite direction (think of this as flows cancelling). In other words, we can construct an equivalent network (the residual network) where the capacity in the same direction of the edge is decreased by the flow amount and the capacity of the reverse edge is increased by the flow amount. Note that if we can push additional flow through in this residual network, we can add this flow to the original flow in the base network and result in a larger flow.

Thus, in general, to find a maximum flow, we can do the following:

- start with the zero flow;
- repeat until there isn't an augmenting path (a source-sink path of positive residual capacity) in the residual network:
  - push additional flow through along the augmenting path;
  - update the residual network

## 2.3 Edmonds-Karp

As outlined in the previous section, Edmonds-Karp finds augmenting paths and pushes flow through them. Specifically, Edmonds-Karp performs the following two steps repeatedly until the residual network disconnects the source and the sink vertex:

- building the layer graph: the layer graph organizes the graph's vertices into layers based on their distances to the source vertex; for each vertex, the parent vertex in the previous layer is identified, and the corresponding maximum possible amount of incoming flow is recorded
- pushing the flow along the *shortest* augmenting path: with the help of the layer graph, Edmonds-Karp can backtrack and find the shortest source-sink path with positive residual capacity, which it then augments the maximum possible amount of flow along.

This choice of augmenting the flow along the shortest augmenting path in the residual network affords Edmonds-Karp some nice theoretical run time guarantees (which translates to great practical performance as well).

## 2.4 Dinic's

Similarly, Dinic's also relies on constructing the layer graph and pushing flow through. However, whereas Edmonds-Karp only augments flow alongside one shortest source-sink path, Dinic's instead pushes flow through all possible shortest source-sink paths present in the layer graph. In practice, this usually means that Dinic's needs to run fewer outer iterations of build layers and push flow, which improves the performance against Edmonds-Karp.

# 3 Approach

## 3.1 Overview

We started the codebase from scratch and wrote everything in C++.

We first implemented the sequential version of both algorithms. We then started experimenting with different parallelism approaches along with constructing the benchmarking framework, including the test case parsing and generation code. For each parallelism approaches, we first implemented and evaluated them for Edmonds-Karp, and later ported them to Dinic's. This makes sense since the two algorithms share a similar main parallelizable component (build layers). Throughout this process, we gradually added different types of networks to evaluate against; the resulting performance characteristics also allowed us to continuously optimize our implementations.

Our final code deliverables include:

- five different test case parsing/generation mechanisms;
- sequential implementations of Edmonds-Karp and Dinic's; and
- (for both Edmonds-Karp and Dinic's) implementations of multiple versions of two different parallelism strategies (four versions for the first and two for the second).

## 3.2 Network parsing and generation

Our code supports five different test case parsing/generation mechanisms. Note that a test case is equivalent to a network which our algorithms are tasked to find the maximum flow of.

- Parsing from a file: we implemented serializing/deserializing a network (alongside its current flow) by storing it in the edge list format; the first line of the file is the number of vertices, the source vertex, the sink vertex, and the number of edges. The subsequent lines are the from vertex, to vertex, capacity, and current flow along the edge.

  This mechanism is mostly used for debugging purposes.
- Generating a uniformly random graph: this method generates a random graph with the specified number of vertices and edges. It does so by iteratively generating edges between uniformly randomly chosen vertices with a uniformly randomly chosen capacity. Note that our network can have self loops and multiple edges between the same pair of vertices. Also note that it is not guaranteed that the generated network

will have a connected source and sink vertex pair (as the edges are random); this is likely to happen when the number of edges is significantly lower than the number of vertices squared.

We primarily used this method to generate test cases to guide development in the early stages of experimenting with different parallelism strategies. We can quickly generate test cases of various size and dense/sparse-ness and get an intuitive, albeit somewhat unreliable, feel of whether our minor changes are helpful or counterproductive. Later on, however, we realized that we needed test cases that guarantee connectivity and have clear cut dense/sparse-ness characteristics.

- Generating a clique: this method generates a clique with size the specified number of vertices. Each pair of distinct vertices are connected by two antiparallel directed edges, each with a uniformly randomly chosen capacity.

  We used this method to generate a dense graph (that's also guaranteed to be connected) to evaluate our parallel algorithms against. Throughout most of the iterations, most of the vertices in the residual graph are the same distance (or at least similar) away from the source, which means that we are more likely to have large frontiers for the build layers process and thus benefit more from parallelism.

- Parsing from a file describing a Delaunay triangulation of uniformly randomly chosen points in the unit square. This methods converts the triangulation (which is stored in the adjacency list format) into a network by taking each undirected edge and replacing it with two antiparallal directed edges with uniformly randomly chosen capacities.

  Networks derived from Delaunay triangulations are great candidates for sparse graphs to evaluate our algorithms on. Not only are they guaranteed to be connected, the indegree and outdegree of individual vertices are reliably upperbounded, which means that we expect to see limited frontier sizes and thus impaired performance of our parallel algorithms.

- Generating a random grid network: this method takes in the number of rows and columns that describe a grid of vertices as well as a separate source vertex and a separate sink vertex. Within the grid, between consecutive rows, there is an edge (with uniformly randomly chosen) from every single vertex in the top row to every single vertex in the bottom row. Additionally, the source vertex has an outgoing edge to each vertex in the first row and every vertex in the last row has an outgoing edge to the sink vertex. Essentially, the network mimics a neural network with several fully connected layers of the same size.

  By varying the number of rows and columns of the grid network, we can essentially control its density—tall grid networks are sparse (lots of small layers) whereas wide grid networks are dense (a few large layers). We expect poor parallel performance of our algorithm for the former and better performance for the latter.

## 3.3 Parallelism strategies

Both Edmonds-Karp and Dinic's first construct the layer network; they then push flow through this layer network in different ways.

After some initial brainstorming and experimentation, we realized that it is quite challenging to parallelize the second step (push flow) of the two algorithms.

Edmonds-Karp's push flow relies on iteratively backtracking and looking up a vertex's parent in the layer graph to push flow through the shortest augmenting path—a process that is inherently sequential (each step depends on the result of the previous step's memory access).

For Dinic's, each inner run of push flow modifies the layer graph, and hence later runs unavoidably depend on the results of previous ones. We did attempt to parallelize this by employing fine-grained locking on the nodes and having each thread individually exploring the edges at randomly—essentially probabilistically making the augmenting paths non-overlapping, but overall parallelism in this direction seemed to be too complex and unrealistic for us to proceed.

We thus focus on parallelizing the shared first step between Edmonds-Karp and Dinics—constructing the layer network. This first step is very similar to constructing the BFS tree of the network and additionally keeping track of the maximum incoming flow of a vertex. Whereas sequentially it suffices to perform BFS using a queue (and simply process the tasks off the queue until the queue is empty), in parallel, we cannot simply use the central task queue to store all vertices to process and have each thread fetch vertices off the front of the queue: sequentially, this approach guarantees that vertices within the queue are ordered from closer to the source to farther away from the source; in parallel, however, it is possible for threads to process vertices at different speeds and thus add vertices to the queue out of order. Thus, to parallelize building the layer graph, we have to abide by the constraint that we finish process all vertices the same distance from the source before moving on to farther vertices. Adopting standard graph terminology, we have to finish processing the current frontier of vertices entirely before moving on to the next frontier. Note that this concept of frontier is precisely the layers of the layer graph.

### 3.3.1 Top-down strategy

On a high level, our first approach to parallelizing the layer graph construction step of Edmonds-Karp and Dinic's is straightforward: we parallelize over vertices within the current frontier; for each vertex in the current frontier, we (now sequentially) go through each outgoing edge and check if its neighbor is eligible to be added to the new frontier.

**Synchronization** Although the parallelization is straightforward, there are several subtle points where concurrent reads and writes need to be carefully considered.

- Linking child vertices: as different threads process different parent vertices in the frontier, it could very likely be possible that multiple parent vertices have outgoing edges to the same eligible child vertex. In this case, it's crucial that the parent vertices not attempt to concurrently "claim" the child vertex and potentially corrupt the graph data structure and/or pollute the new frontier.
- Inserting into the new frontier: although we only perform concurrent reads to the current frontier, we obviously need to perform writes to the new frontier; thus, it is crucial that our implementation handles multiple threads attempting to insert into the new frontier different child vertices at the same time.

**Overhead** In terms of overhead beyond those inherent to the OpenMP abstraction (setting up threads, etc.), this strategy does require two additional int vectors with lengths the number of vertices (frontier and new frontier; we need to preallocate the number of vertices since we obviously cannot dynamically resize; we instead keep track of the actual lengths of the vectors externally). However, this effect is mitigated by the fact that in a typical run of Edmonds-Karp (and Dinic's, although the latter typically requires fewer overall iterations as it augments in batches), multiple iterations of build layers and push flow are necessary, and that our frontiers and new frontier vectors can be reused between iterations.

**Version 1: coarse-grained locking** In this initial implementation of the top down strategy for build layers, we solve the first synchronization issue (linking child vertices) by wrapping the corresponding logic within an OpenMP critical section.

```
for (const auto j: adj[u])
    if (const auto& [from, to, cap, flow]{edges[j]};
        flow < cap and edge_in[to] == NONE) {
#pragma omp critical
        edge_in[to] = edge_in[to] == NONE ? j : edge_in[to];
        if (edge_in[to] == j) {
            /* PERFORM CHILD LINKING LOGIC */
        }
    }
```

Note that this is essentially a form of coarse-grained locking around the edge_in vector (which stores the parent/incoming edge for each vertex), since the critical sections are mutually exclusive. As a minor optimization, we perform a read to edge_in[to] to check if it is even necessary to acquire the lock (similar to the philosophy of test-test-and-set); note that once we've acquired the lock, we would still need to double check that we can actually link the child vertex, since it's possible for another parent vertex to have linked the child vertex between the test and the acquiring of the lock.

Regarding the second synchronization issue (inserting into the new frontier), we realized we could just use OpenMP's atomic capture pragma.

```
int index{};
#pragma omp atomic capture
index = new_frontier_size++;
new_frontier[index] = to;
```

**Version 2: fine-grained locking** Once we realized that omp critical is essentially coarse-grained locking, we of course attempted applying fine-grained locking to the first synchronization issue. We created a vector of OpenMP locks that persisted across the multiple runs of build layers and push flow (this reduces overhead as opposed to creating and destroying the locks for each iteration of build layers).

```
for (const auto j: adj[u])
    if (const auto& [from, to, cap, flow]{edges[j]};
        flow < cap
        and edge_in[to] == NONE
        and omp_test_lock(&lock[to])) {
        if (edge_in[to] == NONE) {
            /* PERFORM CHILD LINKING LOGIC */
        }
        omp_unset_lock(&lock[to]);
    }
```

Note that we can use omp_test_lock as opposed to waiting to acquire the lock since if another thread is using the lock that we desire, it must mean that the corresponding child vertex is already being claimed. Also, we still read edge_in[to] before we attempt to test the lock, and we still double check even if we successfully acquire the lock.

This version's approach to the second synchronization issue is the same as its predecessor's.

**Version 3: atomic compare-and-swap** At this point, we are essentially implementing the atomic compare-and-swap operation using OpenMP pragmas: we are attempting to check if the incoming edge of the child vertex is currently NONE, and if so we want to swap it atomically with the current candidate edge. Given this, our third version directly uses std::atomic's compare_exchange functionalities. Note that we choose the strong as opposed to the weak version since we are not spinning to wait for the value to become zero (which is the use case for the weak alternative); in that case the CAS should just fail.

```
for (const auto j: adj[u])
    if (const auto& [from, to, cap, flow]{edges[j]};
        flow < cap and edge_in[to] == NONE)
        if (auto none{NONE};
            edge_in[to].compare_exchange_strong(none, j)) {
            /* PERFORM CHILD LINKING LOGIC */
        }
```

The second synchronization issue is also simplied a lot. We can simply use an atomic int to represent the size of the new frontier and use atomic post increment to get the index that we should be inserting our new vertex into.

```
new_frontier[new_frontier_size++] = to;
```

**Version 4: atomic compare-and-swap with relaxed memory ordering** We did some more research into std::atomic and noticed that most of its operations take in optional arguments specifying the memory order, and that the default value seems to be the overly strict std::memory_order_seq_cst (sequential consistency). Thus, the final optimization we executed for this top-down strategy is to instead use the weakest appropriate memory order: std::memory_order_relaxed (seems to be slightly stronger than the weak consistency introduced in class; apparently this still preserves same-thread read and write orders).

```
for (const auto j: adj[u])
    if (const auto& [from, to, cap, flow]{edges[j]};
        flow < cap and edge_in[to].load(RELAXED) == NONE)
        if (auto none{NONE};
            edge_in[to].compare_exchange_strong(
                none, j, RELAXED, RELAXED
            )) {
            /* PERFORM CHILD LINKING LOGIC */
        }
```

**Evaluation** Consider the following run time table for different versions of the top-down strategy across ten runs on the GHC machine with 8 cores, on the clique test case with 800/8000 vertices for Edmonds-Karp/Dinic's.

| version (Edmonds-Karp) | coarse-grained | fine-grained | atomic CAS | atomic CAS, relaxed |
|---|---|---|---|---|
| average run time (ms) | 1733 (100%) | 832 (48%) | 767 (44%) | 753 (43%) |

| version (Dinic's) | coarse-grained | fine-grained | atomic CAS | atomic CAS, relaxed |
|---|---|---|---|---|
| average run time (ms) | 1632 (100%) | 1461 (90%) | 1408 (86%) | 1401 (86%) |

Despite the differing scale of the performance of the two algorithms, it is clear that with the four versions of the same top-down strategies, there is a huge drop in runtime from coarse-grained to fine-grained and a small drop between fine-grained and atomic CAS. This is fairly expected as synchronization through the OpenMP abstract is expected to incur greater overhead than directly implementation through std::atomic.

In the source code, parallel1a through parallel1d correspond to the four versions detailed above. In the results section, we mainly focus on the performance of the fourth version of the top-down strategy (atomic CAS with relaxed memory consistency), as we believe it is the most optimized version.

### 3.3.2 Bottom-up strategy

Besides the top-down strategy for constructing the layer network, we also experimented with deriving the new frontier from the previous frontier from the bottom up: instead of checking the outgoing edges of the parent vertices in the new frontier, we can instead check whether each individual child vertex is eligible to be added to the new frontier based on whether or not it has an incoming edge from a parent vertex in the current frontier. Note that this child vertex checking can be fairly straightforwardly parallelized if we represent the frontier as **boolean vectors** (this actually ended up being the biggest bug/footgun we encountered for this project) where each element indicates whether the corresponding vertex is present in the frontier/new frontier.

**Synchronization** As discussed above, this bottom up approach relies on the fact that we only perform concurrent reads to the current frontier, and each thread will by design write to data corresponding to **disjoint** set of vertices. Therefore, no explicit synchronization is actually needed.

**Overhead** In terms of additional data structures, we do need two flag vectors to represent the frontier and the new frontier; however similar to the top-down approach they can be reused across the different build layer runs.

However, another aspect of this approach involving overhead is the artifactual computation we have to perform as a result of parallelizing over child vertices: in cases where the current frontier is small, we would still need to perform essentially the same amount of computation, whereas the top-down approach can plausibly run faster. Conversely, in cases where the graph is dense and each frontier contains sufficiently many vertices, this top-down approach can benefit from the greater, “flatter”, parallelism its design affords.

**Implementation and bug encountered** Implementing the bottom-up strategy was mostly straightforward...

```
#pragma omp parallel for default(none) shared(num_verts, edges, adj, edge_in, \
flow_in, empty_frontier, frontier, new_frontier)
for (int v = 0; v < num_verts; ++v)
    /* CHECK IF CHILD VERTEX V IS ELIGIBLE FOR THE NEW FRONTIER */
```

...except for one single bug that we encountered. After implementing the frontiers as bit vectors, we ran the correctness tests multiple times, and occasionally it seems that the bottom-up strategy will result in incorrect flow (it sometimes hangs as well, presumably in one of the while loops asserting the frontier is not empty). This of course indicates the presence of a data race—even though no explicit synchronization is required for this approach: as the different threads assess different child vertices and determine that some of them are eligible to be added to the new frontier, they will write to the new frontier flag vector concurrently, a process that should be safe since the threads are by design writing to locations corresponding to different flags. However, it turns out that std::vector<bool> internally uses **bit-packing**, i.e., groups of adjacent booleans are stored within the same word, and writing to a single location involves read-modify-write-ing the entire word in a non-atomic process. As a result, as the different threads access nearby locations on the flag vector, flag updates get corrupted and lost.

Once we figured out the root cause of the data race, we realized we essentially just needed to pad the boolean flags, or equivalently using std::vector<int> as opposed to std::vector<bool> to represent frontier membership.

**Optimization and evaluation** We did think of ways we can potentially improve this rather straightforward bottom up approach. Our original implementation uses a fixed, static assignment of the vertices to the different threads. However, since not all vertices have the same in degree, the algorithm can potentially benefit from dynamic scheduling: if one thread is stuck evaluating a few vertices with lots of incident edges, other threads can potentially evaluate some of its other vertices. However, it turns out that for the test cases that we ran against, the approach doesn't really benefit from dynamic scheduling, i.e., load balance is more or less flat rather than spiky.

In the source code, parallel2a corresponds to the static schedule implementation, and parallel2b corresponds to the dynamic schedule implementation, which is very rarely slightly better and often worse than its static counterpart. As a result, we elect to use the static schedule version for our main results section below.

# 4 Results

## 4.1 Experimental Setup

For each of the Edmonds-Karp and Dinic's, we perform the following experiments:

- Runtime and speedup analysis of top-down and bottom-up approaches on GHC machines
- Explore change in runtime under different problem sizes
- Runtime and speedup analysis on PSC machines with more cores

## 4.2 Dense graph performance

For dense graph performance analysis, we choose cliques as our test cases. Cliques are graphs where every pair of nodes is connected by an edge.

As the graph density increases, the advantage of Dinic's algorithm over Edmonds-Karp becomes more pronounced. In cliques, the time complexity of Dinic's BFS phase becomes $O(V^3)$, while Edmonds-Karp's BFS phase has a time complexity of $O(V^4)$.

Additionally, Dinic's algorithm can take advantage of the dense graph structure to efficiently compute the maximum flow by sending multiple flows along shortest paths in a single phase, further improving its performance compared to Edmonds-Karp, which sends one flow at a time.

Thus, we run Edmonds-Karp on cliques of size 800 and run Dinic's on cliques of size 8000 to explore their speedups.

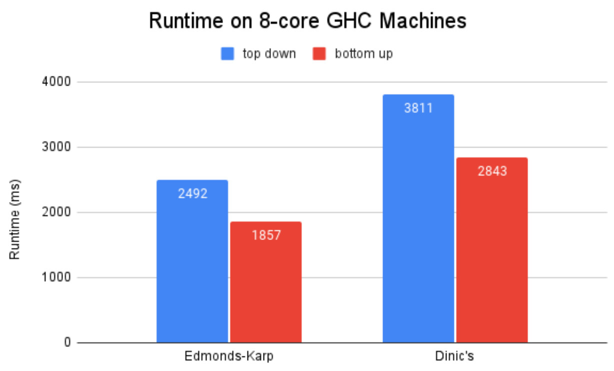
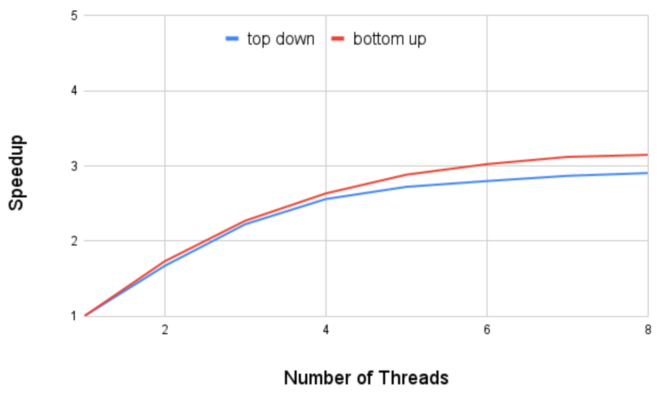
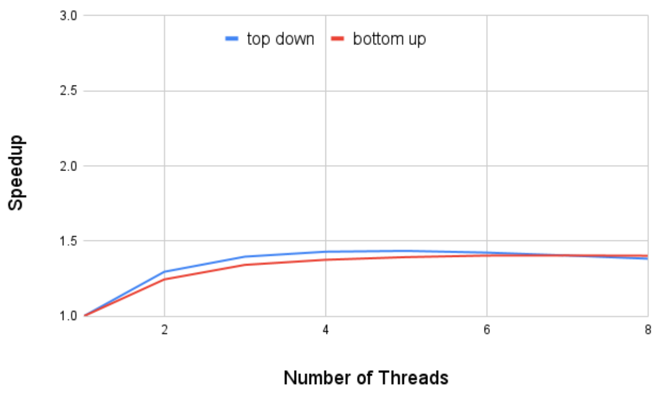


Figure 1: Runtime Comparison between Top-Down and Bottom-Up on Dense Graph

Top-down and bottom-up approaches achieve similar runtime, with bottom-up being slightly better. Recall that top down approach loops through all the vertices and check whether it is a neighbor of current frontier, while bottom up only loops through vertices in the current frontier and add them to the new frontier. In clique, every vertex should be added to the new frontier, so in theory, both approaches should have the same performance. However, in practice, top-down faces the redundancy of finding frontier neighbors for every vertex, but in bottom-up, all vertices are added to the new frontier upon first check, which improves the runtime.



(a) Edmonds-Karp



(b) Dinic's

Figure 2: Speedup vs number of threads

From the graph, we can see that Edmonds-Karp has better speedup than Dinic's. This is likely due to Edmonds-Karp having a higher proportion of the parallelizable BFS phase, allowing it to benefit more from the parallel execution, as predicted by Amdahl's Law.

In Edmonds-Karp, we reach a speedup of 3x when running on 8 cores. This is likely due to contention for shared resources. With more threads executing concurrently, there is increased contention for shared resources such as memory and CPU pipelines, leading to performance degradation. In section 4.5, we run the same experiments on PSC machines, which have more cache sized and memory bandwidth. On PSC machines, Edmonds-Karp scales better (up to 30x), so it supports our conjecture that the speedup on GHC machines is limited by shared resources.

## 4.3 Sparse graph performance

For sparse graph performance analysis, we choose delaunay graphs as our test cases. Delaunay graphs are consisted of triangles, so there are $O(V)$ edges and suitable for testing sparse graphs. They are widely used for testing max flows and also have interesting real-world applications, such as to construct mesh models.

We retrieved delaunay graphs of size $2^{10} \sim 2^{25}$ from DIMACs website, and randomly assigned capacities for each edge with maximum capacity 1000. We ran Edmonds-Karp on delaunay network of size $2^{16}$, and ran Dinic's on delaunay network of size $2^{18}$.

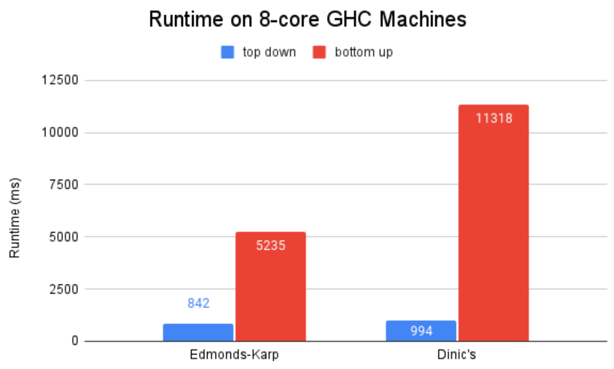
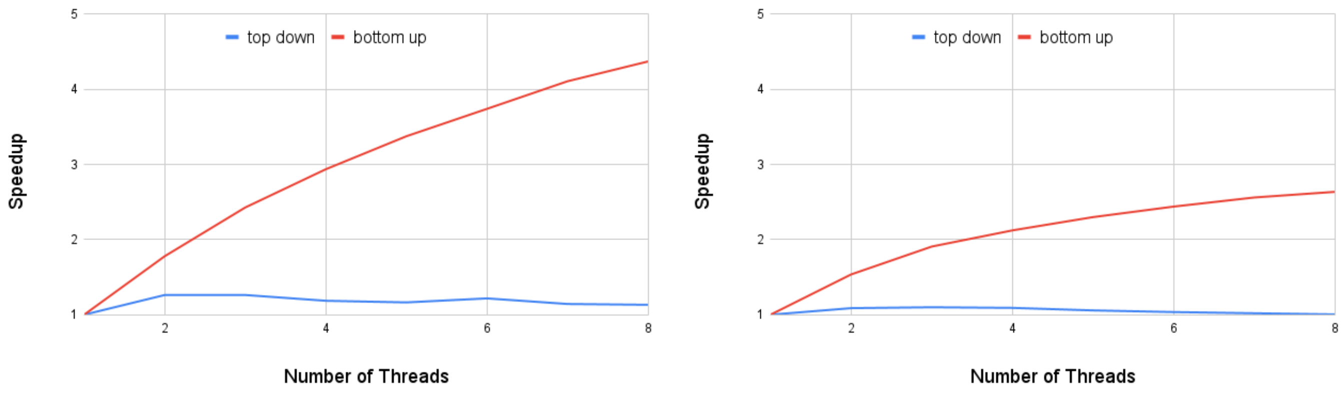


Figure 3: Runtime Comparison between Top-Down and Bottom-Up on Sparse Graph

Unlike dense graphs where top down and bottom up approaches have similar performances, in sparse graph, speedup of bottom up approach beats top down significantly. The nature of sparse graphs makes it extremely inefficient for top down approach; it loops through too many non-neighbors.



(a) Edmonds-Karp (b) Dinic's

Figure 4: Speedup vs number of threads

From the graph, we can see that bottom-up has higher speedup than top-down although its absolute runtime is worse. This is because the inefficient part of bottom-up, checking adjacency, is parallelized. On the other hand, in top-down approach, the frontier size is small (3 in this case), so parallelizing on finding neighbors of frontier is not useful. For 8 cores, its overhead even beats the parallelization, causing speedup to be less than 1x.

## 4.4 Problem size sensitivity

In the previous sections, we ran different algorithms on one single graph of fixed size. Another interesting behavior is how runtime scales up to problem size. We ran both Edmonds-Karp and Dinic's on cliques of size 200 $\sim$ 1200 with 1 core and 8 cores, respectively.

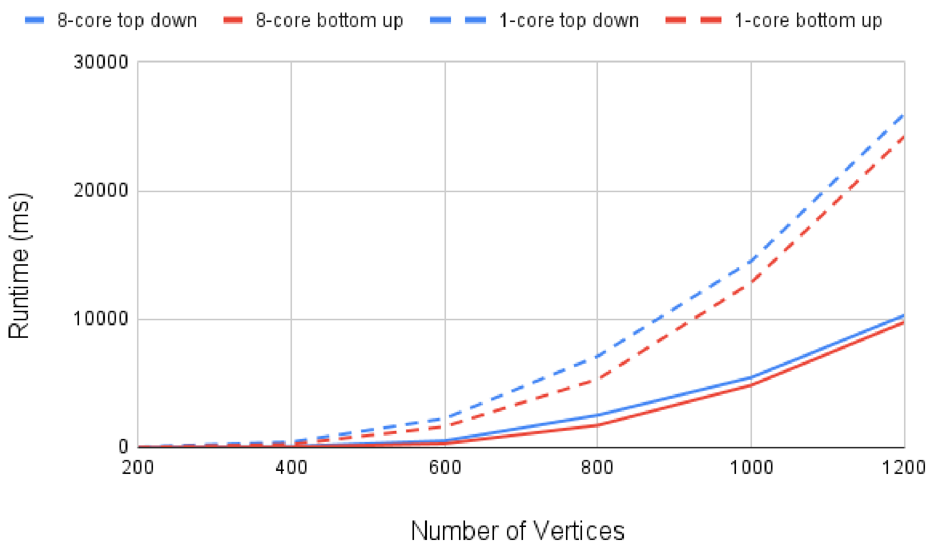


Figure 5: Edmonds-Karp Runtime vs Problem Size on Dense Graph

From the graph, we can see that top-down approach has slightly higher runtime than bottom-up. This is consistent with our previous experiments and analysis. Running with 8-core largely improves the runtime.

We then further calculated the speedup for 8 cores with respect to 1 core:

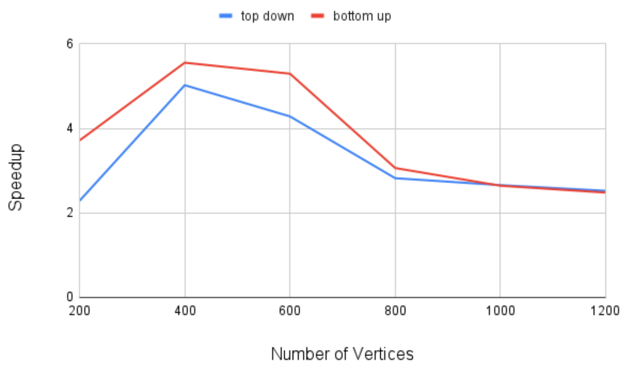


Figure 6: Edmonds-Karp Speedup vs Problem Size on Dense Graph

The speedup trend observed in the graph, where it initially increases and then decreases as the number of vertices grows, can be explained by Amdahl's Law and the interplay between the parallelizable and sequential portions of the Edmonds-Karp algorithm with parallel BFS.

Initially, with a smaller number of vertices, the speedup increases because the parallel BFS portion, which can effectively utilize multiple cores, dominates the overall computation time. As the number of cores increases (in this case, from 1 to 8), the parallel portion of the algorithm can be executed more efficiently, resulting in a higher speedup.

However, as the number of vertices continues to grow, the sequential portion of the algorithm, which includes tasks such as augmenting the flow along the found paths, becomes more significant relative to the parallel BFS portion. According to Amdahl's Law, the maximum achievable speedup is limited by the sequential portion of the algorithm that cannot be parallelized.

The decreasing trend in speedup can be attributed to memory contention and cache effects. With larger problem sizes, the amount of data required by each thread increases, leading to potential memory contention and cache thrashing. These effects can degrade the overall performance.

## 4.5 Speedup on PSC machines

On PSC machines, we ran Edmonds-Karp on clique of size 1,200, and ran Dinic's on clique of size 10,000. Their speedups are as follows:

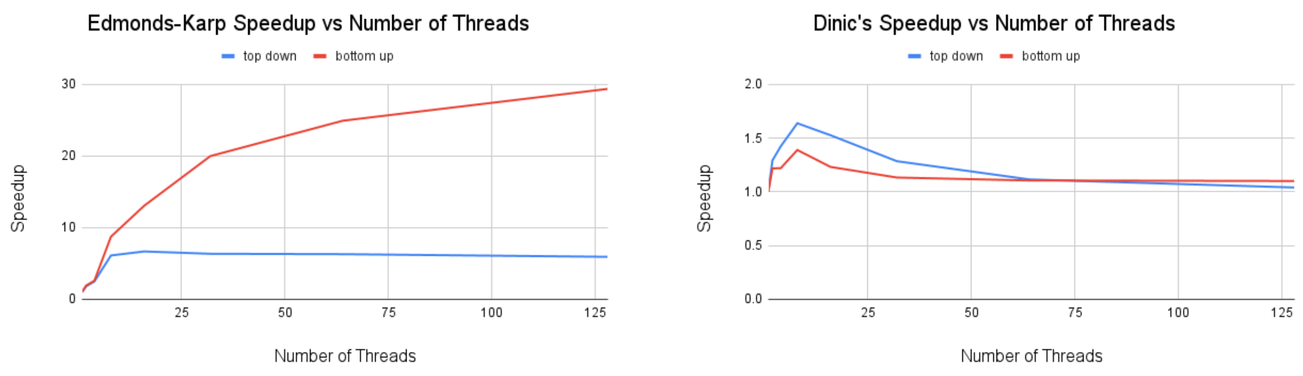


Figure 7: Speedup vs number of threads on PSC machines

In the Dinic's algorithm, the speedup initially increases as the number of threads increases from 1 to 8 threads. This initial increase in speedup is expected as adding more threads allows better parallelization of the computationally intensive BFS phase of the algorithm. However, after reaching the peak, the speedup starts to decrease gradually as more threads are added. This decrease in speedup can be attributed to several factors:

- Limited scalability: As mentioned before, Dinic's algorithm has a large sequential part, which prevents further speedup beyond a certain number of threads.
- Contention for shared resources: With more threads executing concurrently, there is likely increased contention for shared resources such as memory and CPU pipelines, leading to performance degradation.
- Inefficient cache: As the number of threads increases, the memory access patterns become more complex due to multiple threads accessing and modifying the level graph data structure concurrently.

In the case of Edmonds-Karp, the speedup exhibits a different behavior compared to Dinic's algorithm. The speedup of bottom-up approach increases steadily as the number of threads increases, which the top-down approach plateaus and stops scaling earlier.

Also, the absolute runtime for bottom-up outperforms top-down, which we haven't encountered on GHC machines. Here is a detailed breakdown:

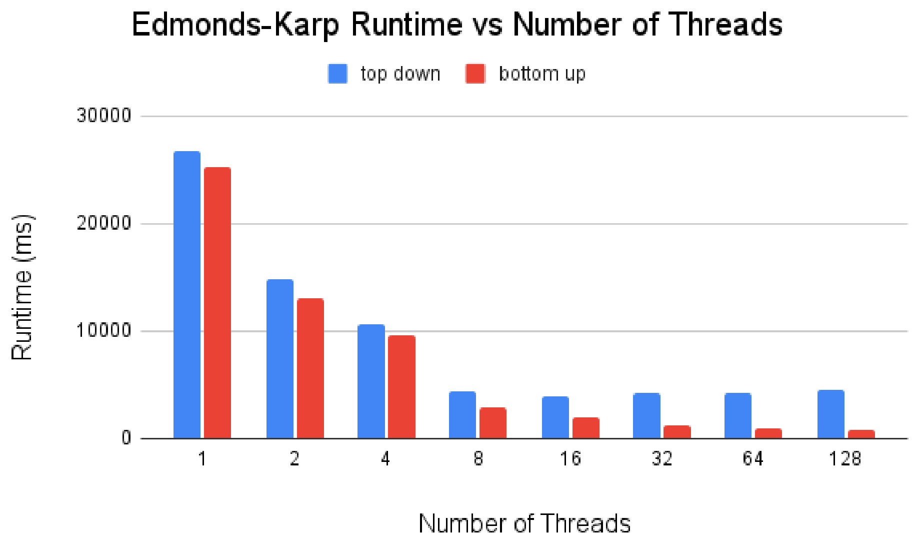


Figure 8: Edmonds-Karp Speedup vs Number of Threads on Dense Graph

The reason lies in the different memory access patterns and the potential for better cache utilization in the bottom-up approach.

In the bottom-up approach, when looping through every vertex in the frontier and adding their neighbors to the new frontier, the memory accesses are more localized and exhibit better spatial locality. Specifically, in our clique test case, every vertex is connected to every other vertex, so when traversing the neighbors of a vertex in the frontier, the memory accesses will be concentrated around the adjacency lists of those vertices. This localized memory access pattern can better leverage the cache, reducing the number of cache misses and subsequent main memory accesses.

On the other hand, in the top-down approach, when looping through every vertex and checking if it is a neighbor of a vertex in the frontier, the memory access patterns are more scattered and exhibit poorer spatial locality. Each thread may need to access the adjacency lists of multiple vertices in different parts of the graph, leading to more cache misses and potentially higher memory access latency.

## 5 References

- Lecture 12: 15-451/651: Network Flow II, lecture notes, School of Computer Science 15-451, Carnegie Mellon University, Fall 2023.
  https://www.cs.cmu.edu/~15451-f23/lectures/lecture13-flow-ii.pdf
- 15-210 Lecture Notes, Fall 2022, Parallel Graph Algorithms
- Delaunay Graphs. 10th DIMACS Implementation Challenge,
  https://sites.cc.gatech.edu/dimacs10/archive/delaunay.shtml

## 6 Work Distribution

Both teammates contributed equally in terms of determining the overall trajectory of the project, performing literature review, brainstorming test cases, and writing the project proposal, milestone report, and final report. Throughout the iterations of the parallel algorithms, Xinyue primarily focused on running the experiments and visualization whereas Siyuan primarily focused on implementing the different optimizations.

Overall, we feel that we had comparable contribution to the project with respect to our strengths, and would like to elect for a 50/50 credit distribution.