15-418 Project Report

Gary Gao, Mingxuan Li
wgao2@andrew.cmu.edu, mingxua3@andrew.cmu.edu

Basic Project Information

- **Project Title**: Parallelization and Analysis of Algorithm for Maximum Flow Problems
- **Team Members**: Gary Gao, Mingxuan Li
- **Website URL**: https://garygao33.github.io/ParallelFlow/
- **Github**: https://github.com/garygao33/ParallelFlow

1 Summary

We implemented and parallelized Dinic's algorithm for the maximum flow program using the shared-address-space framework OpenMP. We evaluated the performance of our two parallel versions on various inputs using the CPU-based GHC and PSC machines and analyzed their trends of scalability and sensitivity to inputs. We demonstrated the overall desirabliity of the edge-disjoint approach and showed the input graph characteristics such as density and size have noticeable impact on speedups.

2 Background

In this project, we aim to parallelize Dinic's algorithm using OpenMP on shared-memory multicore CPU systems. Our focus will be on exploiting the abundant parallelism in level graph construction and augmenting path searches, while carefully designing synchronization strategies for ensuring correct blocking flow computation. We intend to analyze the speedup performance of the parallelizations, with additional studies of its sensitivity to various characteristics of inputs. By doing so, we hope to gain valuable insights into optimizing the highly irregular graph algorithms on modern CPUs.

2.1 Maximum Flow Problem

The **maximum flow problem** is a fundamental problem in graphs. Given a directed graph where each edge has a capacity, the objective is to compute the maximum amount of flow that can be sent from a source node to a sink node, subject to capacity constraints and flow conservation.

The problem is formally stated as follows: given a directed graph $G = (V, E)$ where each edge $e \in E$ has a non-negative capacity $c(e)$, find the maximum feasible flow from a designated source node s to a sink node t. The flow on each edge must satisfy two primary constraints: the *capacity constraint*, ensuring that for every edge e, the flow $f(e)$ satisfies $0 \leq f(e) \leq c(e)$; and the *flow* *conservation constraint*, which requires that, for any node v except s and t, the total inflow equals the total outflow.

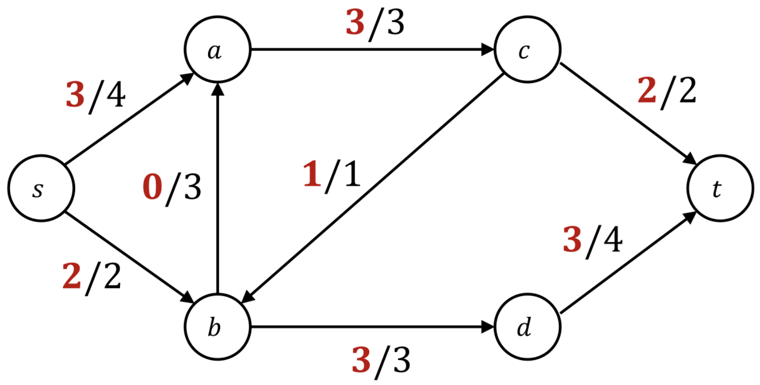


Figure 1: A very simple example of maximum flow on graphs, with flow and capacity marked on directed edges. The image is taken from 15-451 Spring 2025 Lecture 12 notes.

Computing maximum flow is not only a theoretical interest but also a practical necessity across various real-world domains, such as routing in computer networks, optimizing transportation systems, solving matching problems in bipartite graphs, and performing image segmentation in computer vision, which makes efficient algorithms for this problem particularly valuable. Several algorithms exist for solving the maximum flow problem, including the Ford-Fulkerson algorithm and its refinements such as Edmonds-Karp. However, one of the most efficient and structurally suitable algorithms for parallelization is Dinic's algorithm.

2.2 Dinic's Algorithm

Dinic's algorithm is a powerful and efficient method for solving the maximum flow problem, notable for its structured approach that enhances opportunities for parallel execution. Unlike earlier methods such as the basic Ford-Fulkerson algorithm, which augments along one path per iteration, Dinic's algorithm operates through a series of well-organized *phases*.

```
Dinic(G, s, t):
    initialize flow f(e) = 0 for all edges e in G and build residual graph G_f
    while build_level_graph(G_f, s, t) builds a valid level graph:
        while there exists a path p in level graph from s to t:
            send flow along path p
            update residual capacities in G_f
    return total flow from s to t
```

Figure 2: A high-level pseudocode of Dinic's algorithm.

Each phase consists of two distinct parts. First, the algorithm constructs a **level graph** by assigning levels to nodes based on their distance from the source in the current residual graph. Only edges that advance nodes from one level to the next are retained in this graph. Second, the algorithm computes a **blocking flow** within the level graph by finding and augmenting flow along multiple edge-disjoint paths from the source to the sink until no further such paths exist. By iteratively performing these two steps, the algorithm ensures significant progress toward the maximum flow in each phase.

The structure of Dinic's algorithm allows for a greater degree of parallelism compared to sequential augmenting path methods. Because multiple paths can be processed independently, especially within the blocking flow computation, Dinic's algorithm is a promising candidate for parallel implementation.

2.3 Augmenting Path and Blocking Flows

To provide the readers a better understanding of Dinic's algorithm, we explain a few concepts mentioned.

An **augmenting path** in the residual graph is a path from the source to the sink along which additional flow can be pushed. Dinic's innovation lies in the idea of using multiple augmenting paths within a single phase to achieve a **blocking flow**. A blocking flow is a flow where every path from the source to the sink in the level graph contains at least one saturated edge, meaning no further flow can be augmented without reconstructing the level graph.

These concepts are crucial because they guarantee that each phase makes nontrivial progress toward the final maximum flow and that the number of phases required is bounded by the number of nodes, leading to strong theoretical performance guarantees. Furthermore, they provide the basic structures on which the algorithm can be parallelized. Utilizing augmenting paths and blocking flows is essential for comprehending the core mechanics of Dinic's algorithm and its parallelization challenges.

2.4 Key Data Structures and Operations

Dinic's algorithm relies on several important data structures. The graph is generally stored as **Adjacency lists**, which efficiently represent the graph's topology and supporting rapid exploration of neighboring nodes. The **residual graph** maintains the current residual flow of edges and their twins after each augmentation, enabling the identification of available paths. The **level array** records the level of each node during the level graph construction phase, essentially storing an implicit level graph by restricting traversal to forward edges.

Another important data structure is the list of **next-edge pointers**. In the implementation, such an array is used to keep track of which outgoing edge of each vertex v should be explored explore next during the blocking flow computation phase.

Operations on these data structures make up the work in Dinic's algorithm. Building the level graph involves visiting each node's neighbors and assigning levels based on their distance from the source. During blocking flow computation, paths from the source to the sink are searched and flow is pushed along them, requiring updates to the residual capacities and careful tracking of visited nodes and edges to prevent repeated useless visits to nodes and maintain integrity and consistency of the flows in the graph.

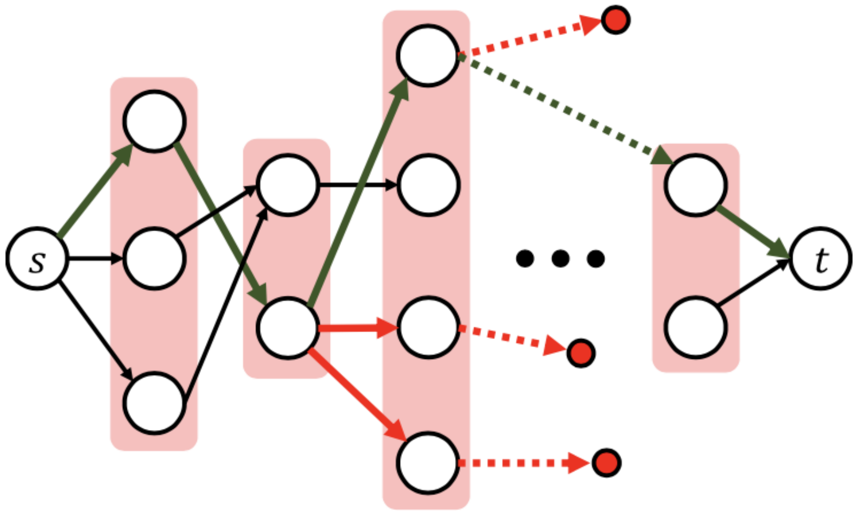


Figure 3: An illustration of the blocking flow finding phase. The red paths illustrates already-explored edges, which are either dead-ends or limited by capacity, making them useless for later searches in the same phase. This shows the importance of maintaining an array of next-edge pointers. The image is taken from 15-451 Spring 2023 Lecture 12 notes.

2.5 Inputs and Outputs

The inputs to Dinic's algorithm are a directed graph $G = (V, E)$ with specified capacities $c(e)$ for each edge, a source node s, and a sink node t. The output is the value of the maximum flow from s to t, along with the flow assignments $f(e)$ for each edge, which collectively satisfy the capacity and conservation constraints.

For our program, the input format is an edge list text file. The source is fixed to be vertex 0 and the sink is fixed to be vertex 1. The first line contains the total number of vertices and the total number of edges. Each line after the first specifies an directed edge by listing its starting vertex, ending vertex, and capacity. This is a natural choice since our implementation uses adjacency lists as the internal representation of graphs. Furthermore, this approach decreases file size of sparse-graph inputs, since empty edges no longer need to be stored, when compared to storing graphs as adjacency matrices. For outputs, we use a single integer maximum flow value, as intended by the problem.

2.6 Preliminary Analysis of Parallelization and Challenges

Naturally, the computationally intensive components of Dinic's algorithm are the level graph construction phase and the blocking flow computation phase. Parallelizing Dinic's algorithm is non-trivial due to the dynamic nature of the residual graph and the tight dependencies between operations. After each flow is pushed along a path, the graph changes immediately, potentially invalidating other concurrent paths being explored. Ensuring correctness in this context requires careful coordination, such as using locks or conflict detection, both of which introduce overhead and complexity.

Another challenge is the inherent irregularity of graph structure. In both synthetic and real-world graphs, node degrees can vary significantly, leading to uneven workloads across threads. This is especially problematic during the blocking flow computation, where some threads may complete quickly while others follow long or useless paths. Such divergence causes load imbalance and makes it difficult to efficiently utilize parallel resources. Furthermore, the algorithm's control flow is highly data-dependent. Whether a node or edge is visited or used in an augmenting path depends on capacities that change at runtime. This dynamic behavior complicates parallel scheduling and synchronization, particularly when trying to extract disjoint augmenting paths or perform simultaneous residual updates safely.

Overall, the level graph construction phase has a more structured, level-synchronous nature and exhibits better spatial and temporal locality, making it well-suited to parallelization. Nodes at the same level can be explored concurrently, and memory access patterns during this phase are relatively predictable. In contrast, the blocking flow computation phase is irregular, with each thread maintaining its own path stack. Access patterns here are less predictable and often pointer-chasing through adjacency lists. It is therefore expected that data parallelism and SIMD instructions are less helpful in paralleliazation. There's low locality and high contention when threads update residual capacities or shared metadata like visited flags or flow values. Augmenting flow requires writing to both forward and reverse edges, which may be accessed by other threads, raising the potential for conflicts. On multicore CPUs, the key challenges include synchronization overhead, especially with fine-grained locks on shared graph structures. Cache locality is expected to be poor due to scattered memory access, and contention on frequently accessed data structures (like adjacency lists or level arrays) could potentially reduce performance.

3 Approach

With the considerations on the challenges of paralleism of Dinic's algorithm presented above, we decide to implement the parallelizations in C++, with the shared-address-space framework OpenMP on CPUs. The choice of C++ allows us to write fast code with frameworks that we already have experience with, such as OpenMP. The high divergence and irregularity of graph element accesses motivates us to choose CPUs as hardware, because they have excellent context switching abilities. Also, OpenMP's shared address space allows for more facilitated implementations of parallelization, considering that data related to flows on edges must be constantly updated and communicated to all threads.

We produced our code base from scratch, mainly with prior knowledge and lecture material from 15-451. We began with sequential Dinic's algorithm, and gradually moved to implementing and improving parallelized versions. We eventually produced two parallelization approaches, which we will introduce below. At the same time, we gradually developed graph generation algorithms that allow us to produce inputs of desired characteristics. To better control the outcome of generated graphs, and ensure their correctness, we implemented graph generation code with Python, which allows for versatile code structures.

3.1 Input Graph Generation

We have developed two types of inputs and their generation code. Graphs can be generally categorized into sparse and dense graphs, both of which are highly influential to an algorithm's performance.

- Sparse Graphs: Sparse graphs are graphs whose vertices have degrees much lower than the maximum possible number. Our method is able to generate randomized sparse graphs with a specified number of vertices and specified average degrees. This is achieved by iteratively adding uniformly random, non-repeating (and no self-looping) edges to all vertices until they reach a certain degrees. We do not allow self-looping edges that create 1-cycles and 2-cycles due to Dinic's algorithm's requirement. To ensure that generated graphs do not have unused subgraphs, we add some extra edges in the end of edge-adding to ensure that connectedness.

  These randomized sparse graphs are important robustness tests for our implementations. Furthermore, the ability to control the number of vertices and their degrees allows us to produce input suites that can be used in related input sensitivity studies.

- Dense Graphs: In contrast, dense graphs are graphs whose vertices have large degrees, in the sense that they are a non-tiny fraction of the total number of vertices. Our method are able to generate layered dense graphs with a specified number of vertices per layer and depth. We have taken inspiration from deep neural networks, and developed code that can build already layered graphs have large layers that are fully connected. Moreover, such structure allows us to control the depth of the graph (the distance from source to sink), which are also crucial for sensitivity studies.

3.2 Parallelization of Level Graph Construction Phase

Dinic's algorithm have two phases of level graph construction and blocking flow computation alternating. That is before the algorithm can start searching for augmenting paths and blocking flows, the level graph construction must be complete, and before the level graph construction begins, the previous round of blocking-flow-finding should have already ended. Given this obvious exclusiveness of two phases, we conclude it is virtually impossible to have a solution that can parallelize both phases simultaneously. Therefore, the two phase are parallelized separately, with OpenMP launching threads inside both phases.

Before we present the final version of this approach, we first introduce the exploration journey.

- Initial approach
  - In the sequential approach, we used the standard algorithm maintaining a queue of reached vertices. However, the level graph constructing phase is readily data parallel, in the sense that the neighborhood of different vertices in the same level can be processed separately, and the next level simply consists of the union of their neighborhood. Hence, we began by modifying the main loop to be looping over the newly constructed levels, and using the `#pragma omp parallel for` directive to parallelize within each level intuitively. This essentially maps blocks of vertices in the newly constructed level to a

unit of work, with the entire level of work distributed to the threads, which correspond to cores in our experimental machines.

  - For the collection of vertices in the next level, a critical section (as `#pragma omp critical`) must protect the code pieces that insert individual neighborhoods to the array storing the new frontier, so that the array is not corrupted by the data races. The implicit barrier at the end of directive would satisfy the synchronization requirement, ensuring that all neighborhoods of current-level vertices are aggregated in an array.
  - Although this naive approach created no correctness issues, the code would run for several seconds even on small test inputs. The issue is that multiple vertice may have outgoing edges to the same vertex, so repeated vertices in the stored level vertex list creates significant extra work, since their neighborhoods are also repeeated during the aggregation.

- Failed improvement: locked acquisition of neighboring vertices.
  - We attempted to prevent repeated vertices by requiring a test-and-set lock when a thread is attempting to add an actual neighbor of its current vertex to its recorded neighborhood. One lock is kept for each vertex in the current level. Unlike normal locks, such locks would not be unlocked, in order to prevent other threads from acquiring the lock again (they would skip when detecting that the lock is active).
  - Tests on tiny inputs proved that this indeed eliminated repeated vertices, but the runtime remained prohibitively long, likely due to the extra work of maintaining locks equal to the number of vertices and testing for locks.

After the failed improvement, we realized the need to have fine-grained atomicity of individual vertices rather than expensive locks. However, there was no naive solution due to the need of branching operations (whether to add the vertex again to the neighborhood or not) based on the detected state of vertices. We therefore did extra reading on documentation and concluded that an atomic compare and exchange primitive was needed. This allowed us to reach a solution with performance comparable to sequential versions.

Our approach is presented below as pseudocode with some details:

```
#pragma omp parallel for
for (vertex u in this level) {
    for (edge e starting at u) {
        v = endpoint of e;
        if (level[v] != -1) continue;
        if (__sync_bool_compare_and_swap(&level[v], -1, current level))
            add v to neighborhood of u;
    }
    #pragma omp critical
        insert recorded neighborhood of u to new frontier
}
```

In our approach, instead of maintaining an array of flags, we use an implicit flag of a temporarily recorded level of -1 to indicate that the vertex has not been claimed. The primitive compare and swap here will return true only if the implicit flag of level -1 is detected, which also swaps the actual level into `level[v]`, marking the vertex as claimed.

The use of an atomic compare and exchange primitive minimizes synchronization overhead and overhead of maintaining and accessing locks, with no extra locations of synchronization beyond the ones that are required by the correctness of data structures.

3.3 Blocking Flow Computation Phase Approach 1: Edge Flow Reservation

After seeing acceptable results on the first phase, we moved to the parallelization of the blocking flow computation phase. As we recall, this phase is more challenging to parallelize than the level graph construction phase. This is mainly because the blocking flow computation phase is irregular, with each thread maintaining its own record of explored paths. This creates serious problem if synchronization is not handled correctly, because simultaneous updates to flow information would produce invalid flows and corrupt the residual graph.

We began by implementing the intuitive approach of having threads record the fact that it is attempting to use the some amount of flow in this edge, in a way that can be communicated to other threads. We describe our approach as edge flow reservation. The idea is that, when a thread explores an edge, it will try to reserve the amount of flow it can push through the edge by updating the flow variable stored in the edge as if this amount of flow is being consumed. This prevents the issue of multiple threads simultaneously including an edge in its augmenting paths and pushing flow through that edge (which easily cause the flow to exceed the capacity).

We present the design process below:

- One important operation necessary for this approach is the rollback, which returns reserved flows on already-explored edges when a thread finishes one round of path exploration. If it fails to find a valid augmenting path, it must refund all the reserved flow to previous edges. Even if it succeeds in finding a valid path, because the flows are reserved without knowledge of the capacities of later edges, the actual flow pushed may be smaller than the reserved amount in earlier edges (if the later edges have a smaller residual capacity), prompting a partial refund of reserved flow.
- Since multiple edges may attempt to update stored flow information of the same edge simultaneously, the atomicity of such operations (including read the flow for passing into recursions). This can be done with multiple `#pragma omp atomic` directives.
- With no limit on overlapping edges in augmenting paths during exploration, each thread is free to explore in any direction. They each maintain their individual next-edge pointer array to determine the next edge to explore on each vertex.
- As a result, instead of explicitly dividing workload and assign them to physical threads, we design each thread attempt to explore all edges on their own, until no further augmenting paths can be found. Although this may appear to produce little parallelism, the shared updates on the flow information in the residual graph would communicate effectively when an edge has been fully consumed and become available, ensuring that unexplorable edges are no longer explored. In a sense, the workload assignment here is implicitly dynamic. This can be achieved with a simple `#pragma omp parallel redution` diretive, which launches the threads and eventually aggregates the total pushed flow via reduction.

We also present the pseudocode of this approach here:

```
#pragma omp parallel reduction(+ local flow):
    declares local next-edge pointer array;
    local flow = 0;
    do {
        find and try to push flow along an augmenting path
            /* reserves flow along the way and refund after if necessary */
            /* all accesses to the residual graph are made atomic */
        local flow += flow pushed
    } while (flow pushed is nonzero)
```

We omitted the repeated atomic directives here. In the implementation, we essentially wrapped the atomic accesses into helper functions for better code structure.

With this approach, we experienced no correctness issues on the inputs and observed nontrivial speedup on GHC when using multiple threads. However, we did note that the speedup virtually plateaued at 4 threads, suggesting that the scalability of this approach are not fully desirable, which we will analyze later. Meanwhile, we tried to reduce synchronization, but ultimately unsucessful because all atomic directives were indeed absolutely necessary for correctness.

3.4 Blocking Flow Computation Phase Approach 2: Disjoint Paths

We noticed that the approach of reserving edges requires careful management of atomicity, as well as expensive rollback (removing or refunding reserved flow from earlier edges) when a path exploration is done. Furthermore, as we mentioned earlier, the preliminary runtime results were suboptimal for that approach. We therefore decided to develop another parallelization approach that does the opposite, which is disallowing any competing edges. In other words, different threads can only simultaneously explore edge-disjoint augmenting paths.

The key advantage of this approach is that there is no need to do any rollback on an already-explored edge since disjoint valid augmenting paths are guaranteed to have no contention for flow capacities. Moreover, in this way, it is guaranteed that an edge is only explored once within the same blocking flow computation phase, maintaining the cost heuristic of computing a blocking flow in the same way as a sequential implementation. We believed that this have a great potential in reducing the extra work and overhead that were present in the first approach. Also, with the blocking flow computation only happening on an implicitly layered directed graph, we no longer need to force atomicity of edge updates, including the residual flow update for its twin, because no two threads would attempt to update the flow of the same edge.

Again, we present the debugging journey first before presenting the final version.

- Initial approach
  - Even though the theoretical mapping of problem data structure to machine threads appear straightforward: each thread explores one augmenting paths, it was difficult to assign threads efficiently when using the `#pragma omp parallel for` directive during implementation. Because of the irregular nature of graphs, it is difficult to predict

the possible edges to explore assigned to each direction. We therefore started with a complex implementation, which launches threads with conditional parallel for directive within the recursive flow-finding function, and keeps track of a global counter of the number of threads being used to avoid unnecessarily launching threads.

- Another important issue is to ensure that different threads indeed explore edge-disjoint paths. The use of an array of next-edge pointer helped us form a relatively simple solution of retrieving the next edge to explore:

```
#pragma omp atomic read
    i = next[u];
#pragma omp atomic update
    next[u]++
```

  The intention of this piece of code is that the thread will retrieve the next explorable edge of the current vertex and increment the indicator, so that other threads will go to explore edges further down the adjacency list. We chose to use `#pragma omp atomic` directives because trial-and-error in the previous sections had proven that fine-grained atomicity was necessary for desirable scaling.
- Before facing any performance issues, we discovered that this initial implementation suffered from occasional segmentation faults on test inputs, as well as flows higher than capacity on certain edges. This hinted at a synchronization issue.

• Resolving correctness issues

- After some careful reasoning on the problematic code, we believed that issue lied within the atomic reads and updates of `next[u]`. The two lines belong two separate atomic clauses, even though they are next to each other. In other words, the atomic directive protects each access individually, but does not guarantee that no disruption would occur between these two lines. This led to multiple threads reading the same value and exploring the same edges, and next edge pointers being incremented mistakenly.
- After discovering mistakes, we initially sought inefficient solutions such as keep restructuring next edge pointer array, which caused even more correctness issues. After examining OpenMP documentations more carefully, we discovered that the two accesses to `next[u]` can be combined as one capture action, which can be made atomic by:

```
#pragma omp atomic capture
    i = nextu[u]++;
```

- This elegant change resolves the correctness issues. However, when running on timed inputs, the program's runtime has large fluctuations. We then ran the TAU profiler, and discovered that synchronization made up a significant portion of runtime.

The remaining problem is creating balance workloads. The current approach of nested parallel for directives not only disregard work balance (since threads attempting to launch threads earlier will obtain the resource), but also creates large overhead of maintaining the threads, including the necessary atomicity for updating the thread count tracker. To resolve this issue, we take inspiration from semi-static work assignment methods. Essentially, in each iteration of the outermost loop, the leveled graph will be recomputed, changing the structures of the initial layers. To utilize this, we can simply go thorugh the first couple of layers of the leveled graph and determine the number of possible exploration directions going out of those layers. This would then form the major loop of the blocking flow construction phase, with the `#pragma omp parallel for` directive launching threads and mapping work (which are organized as blocks of vertices corresponding to the outgoing directions) to physical threads.

Our approach is presented below as pseudocode (of the loop within the flow computation function) with some details:

```
#pragma omp parallel for schedule(dynamic, block size)
for (vertices u aggregated from initial layers) {
    while(true) {
        #pragma omp atomic capture
            i = next[u]++;
        if (i goes beyond limit) break;
        Edge e = Adj[u][i];
        if (edge is valid);
            recursion and explore next vertices;
            update flow;
            if (flow out == flow in) return;
    }
}
```

Note that although the work assignment method takes inspiration from semi-static methods, we still use dynamic assignment for the directive here. This is necessary for workload balance because there is still no information about the actual workload for each block after diving up these vertices form initial layers, due to irregular structures of graphs.

As mentioned before, this approach of enforcing disjoint augmenting paths being explored eliminates the locations that need synchronization as long as disjointness is guaranteed since no edges are in conflict. This minimizes synchronization beyond implicit barriers to only one location that requires atomic capturing of next edge pointers.

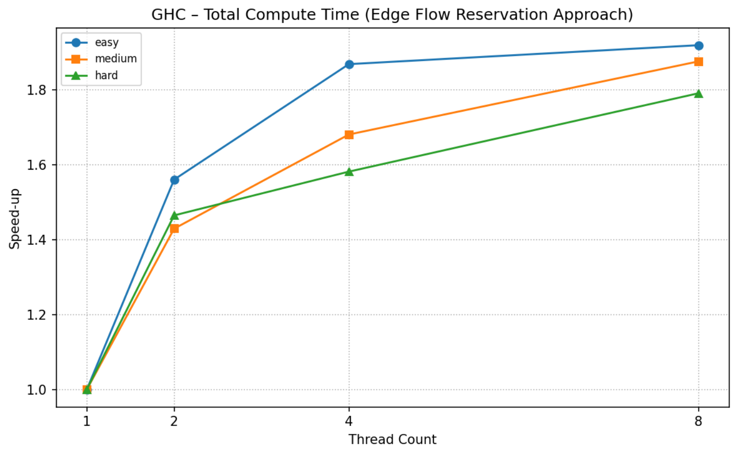
4 Results

In this section, we present the experimental results and detailed analysis for both of our parallel implementations. The experiments include speedup scaling on GHC and PSC machines, as well as cache miss and various sensitivity studies on GHC machines. We also provide a deeper analysis on the individual phases and their influence on the total speedup.

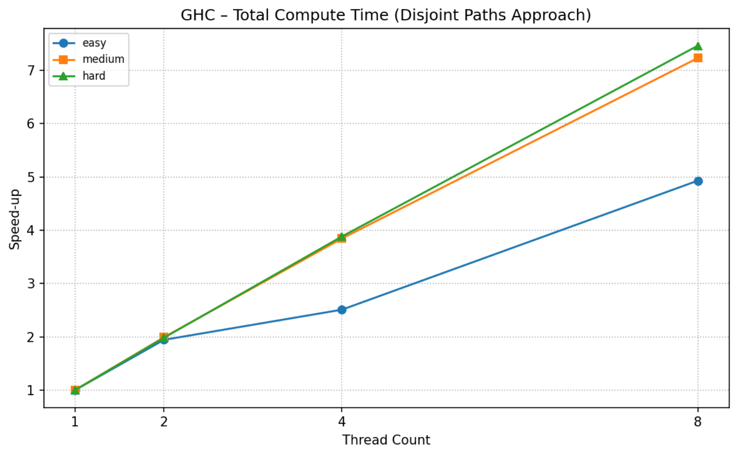
We have chosen 3 of the randomly generated graphs as the inputs for the scalability experiments. We believed that the randomly generated graphs are better than strucurally generated graphs in testing the robustness as well as the actual generalizable speedup scaling of the parallel implementations. We list the specifications of those inputs below:

Input	Easy	Medium	Hard
Number of Vertices	10000	25000	50000
Number of Edges	1000097	6250273	25000492

4.1 Scaling on GHC



(a) Edge Flow Reservation



(b) Disjoint Paths

Figure 4: Speedup Graphs on GHC

Figure 4 shows the plots of GHC scalablity results.

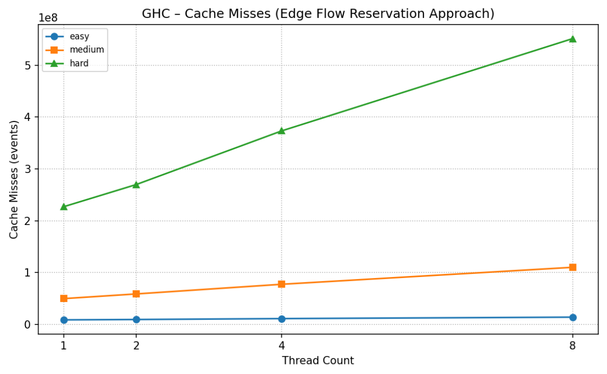
Discussion:

- The disjoint paths approach shows reasonable speedup for the GHC scalability experiments. The speedup growth is only slightly sublinear on the medium and hard inputs, and grows steadily for the easy input, although it has a bigger gap from the linear growth line. Among the three inputs, the scalability is higher for the harder inputs. This can be attributed to the fact that harder inputs have more vertices and available paths to explore for each thread, which decreases the chance of data races, during which atomic updates to the next-edge pointer array becomes sequential and decreasing speedup.
- On the other hand, for the edge flow reservation approach, the growth is obvious sublinear with diminishing growth, with all speedups less than 2. The curves follow the shape of Amdahl's law, which suggests that there may have been a significant portion of sequential

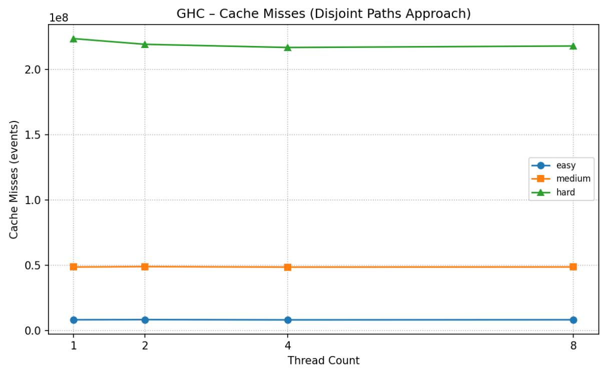
work. However, this is unlikely because as we have analyzed before, both phases are well-parallelizable in theory, and the sequential transitions between the phases cannot account for more than 50% of total work. Some may suspect that workload imbalance have contributed to increased synchronization stalls when thread counts are higher. However, we have run the TAU profiler on 8 threads, and we found that the busy time in main computation functions for all 8 threads were not significantly different.

- One likely explanation for the behavior of the edge flow reservation approach is that as the thread count increases, it is more likely for a thread's calculated augmenting path to be fully invalid due to other threads using up flow of a common edge, or have marked an edge as dead-end before its flow gets refund. This subsequently causes that less augmenting paths are found in a single blocking flow computation phase, with even more total work completed by the threads. The significant amount of extra work then hindered the speedup of the total compute time.
- In addition, the necessary refund check for valid augmenting paths caused the phenomenon that easier inputs have higher speedup for the edge flow reservation approach. Recall that this approach requires to update the refunded flows even for valid augmenting paths, which is more likely to happen as the augmenting paths gets longer. And, in harder inputs, since there are more vertices, the effective depth of the graph is higher, meaning that augmenting paths are likely to be longer.

4.2 Cache Misses



(a) Edge Flow Reservation



(b) Disjoint Paths

Figure 5: Cache Misses on GHC

We would also like to analyze the trend of the number of cache misses to analyze the locality and/or extra work present in the parallelization approaches.

Figure 5 shows the plots of cache misses for the main inputs on GHC.

Discussion:

- To begin with, we should emphasize that such graph algorithms on randomly generated graphs have relatively bad cache locality. This is because the access pattern of stored edges are completely disrupted as the program chases along the next-edge pointers all over the graph, and different threads are likely to access different paths (therefore, neither temporal nor spatial locality are present). Our experiments also show that the overall cache miss rate ranges from 30% to 50% in these inputs, confirming the lack of locality in the algorithms.

- For the disjoint paths approach, we can see that the total number of cache misses does not significantly change (i.e. within random error) as the thread counts increase. This is expected since erratic access patterns from pointer-chasing is unlikely to become worse than in the sequential case.
- On the other hand, the amount of cache misses for the edge flow reservation approach increases steadily as the thread counts increase. Similar to the previous point, parallel searching of augmenting paths is unlikely to worsen the bad locality and cause an increase in cache misses. Our experimental results also show that the total number of cache references increased in a similar rate. This supports our hypothesis that extra works is done when using more threads with the edge flow reservation approach, as stated in the previous section.

4.3 Sensitivity: Graph Density

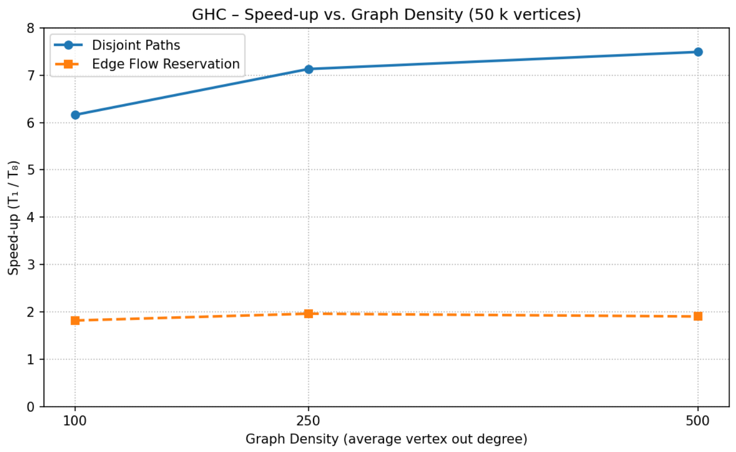


Figure 6: Speed Up v.s. Average Vertex Out Degrees

Another dimension of analyze program behavior is to examine how the program behaves as inputs vary. The way to study this is via sensitivity studies, in which we vary one certain aspect of the input graphs and observe the trend in which speedup changes.

Graph density is an important characteristic of graphs. The relative difference in the degrees of vertices and the total number of vertices is even more important in this program, because one of the parallelization strategies specifically relies on the parallelization over edges out of the same vertex, in the sense that each thread is expected to access different outgoing edges of a vertex of their current augmenting paths reach the same vertex.

We therefore used the random sparse graph generation code to generate 3 input graphs. All of those 3 inputs have 50000 graphs, and their vertices have degree 100, 250, and 500, respectively.

We ran both parallel versions against these inputs using 1 and 8 threads, and the trend of the speedups are presented in Figure 6.

Discussion:

- For the edge flow reservation approach, the speedup stayed steady at a little below 2, similar to the 8-thread speedup performance as the main GHC experiments. This suggests that this particular approach is not sensitive to graph density at this thread count. This is expected since the speedup are already low, with parallelism severely hindered by the extra work introduced by the extra rollbacks on augmenting paths. This effect dominated the relatively small effect of input variations at this scale.
- For the disjoint paths approach, we are able to observe a small speedup boost as the input graph becomes more dense. This is expected because . With higher degrees on vertices, it is less likely for augmenting paths to reach the same vertex within a short period of time (different threads' exploring paths become more divergent), lowering the contention on atomic captures of the next-edge pointer arrays. Furthermore, with a fixed block size, a higher number vertices in fact increases the total amount of blocks available in the parallel for directives, making the granularity of dynamic work assignment higher, helping workload balance as well.

4.4 Sensitivity Studies: Vertex Counts

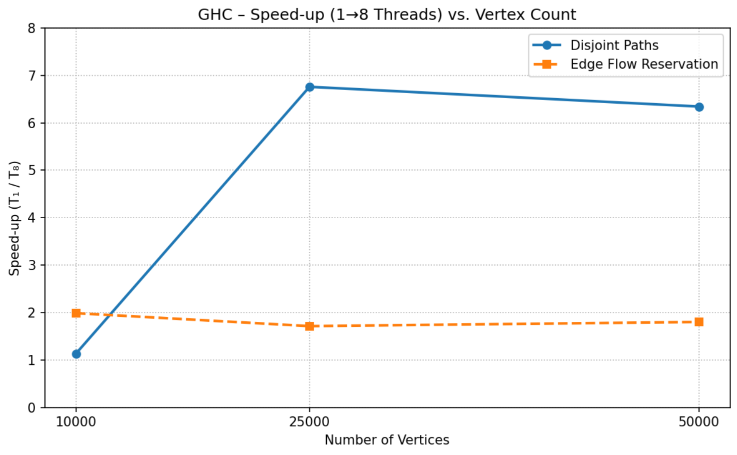


Figure 7: Speed Up v.s. Total Number of Vertex

Total number of vertices is the key data scaling factor in our project. Given fixed graph density (i.e. average out degree per vertex), the number of vertices directly decides the problem scale. When the total number of vertices is low, we observe that there are little benefits to parallelization.

Thus, we decided to examine the effect of increasing graph size (i.e. number of vertices) using this experiment. We constructed 3 large-scale inputs with the same average out degrees as well as flow and capacity properties. There sizes are adjusted to reflect 3 different categories of graph sizes in max flow problem: small (10000 vertices), medium (25000), and large (50000).

We observe that the Disjoint Path approach and Edge Flow Reservation aproach demonstrated completely different scaling pattern here. While both approaches have limited speed up on the small graph, presumably due to the overly tiny amount of work and the comparably high overhead of scheduling and synchronizing 8 threads, Disjoint Paths approach is able to achieve nearly linear speed up for medium-sized graph and to maintain roughly the same speed up on large graphs. In contrary, the Edge Flow reservation approach displays the same sublinear scaling ($\sim$ 2x) across different graph sizes.

This difference in pattern is determined by their different properties in parallelizing operations in Dinics' Algorithm. Since Disjoint Paths approach examines validity of entire augmenting path before update and synchronizations, a large graph size increases the pool of path candidates that can be examined for path augmentation. Increasing graph size means decreasing probability of wasted work due to conflict among threads. Thus, with large graph inputs, threads are able to search and update augmenting path with little conflict. Thus, each thread is able to find a amount of augmenting path close to the sequential implementation, achieving linear scaling.

On the other hand, despite smaller chances of conflicts and contention, Edge Flow Reservation approach suffers from a rapidly increasing communication cost as graph size increase. This is because Edge Flow Reservation find augmenting path in a optimistic fashion: it always tries to reserve the maximum possible amount of flow on every passing edge, and when reach sink at the end send excessively reserved flow back one edge at a time. Since each of such flow reservation and return operation needs to be atomic to resolve data races among threads, Edge Flow Reservation requires multiple times of number of passing vertices amount of atomic operations for each augmenting path candidate. This forms a bottleneck on how much work each thread can do, because a larger proportion of threads' augmenting path candidate will have to wait for atomic operation as number of vertices sale. Therefore, as graph grows, the amount of work each thread can accomplish is capped, resulting in this steady sublinear scaling ($\sim$2x) across different graph sizes that we observe.

4.5 Sensitivity Studies: Graph Depth

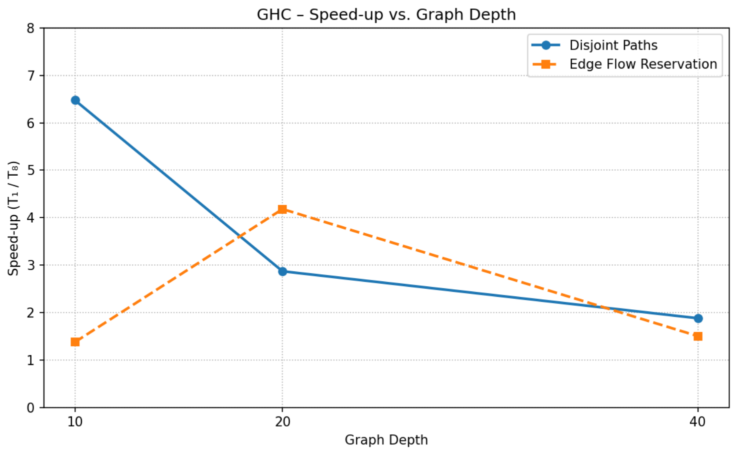


Figure 8: Speed Up v.s. Depth of Graph

In this problem, we define depth as the shortest distance from source to sink in the flow chart. This is a property of the graph that has high impact on the complexity and total amount of work of the max-flow problem, as it determines the size of the search space for possible augmenting path candidates in Dinic's Algorithm.

We toggled the graph depth between (10,20,40) while keeping the total number of vertices as well as other properties of the graph unchanged. We observed that Disjoint Paths approach's speed up scales significantly while the graph depth decreases, while the Edge Flow Reservation approach's speed up increases at first then decrease as graph depth further decreases.

This difference in scaling pattern is well within our expectation.

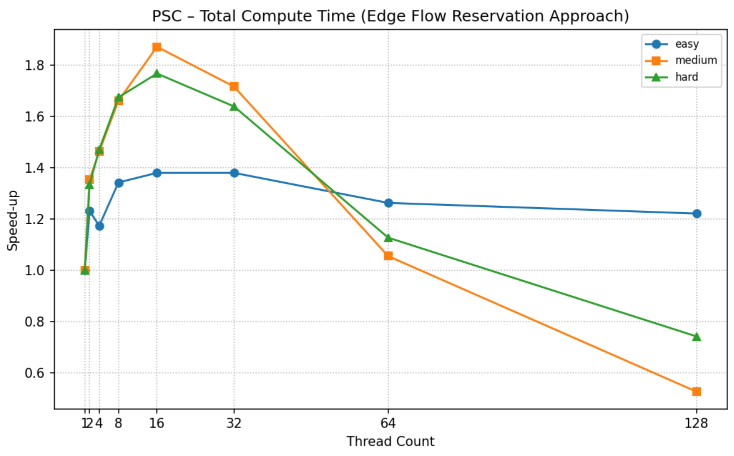
Recall that every augmenting path is a path with available capacity from source to sink vertex. Notice that due to the change in residual, we will always be searching for such path by traversing from source and check for arrival at sink. This is tight to one of Dinic's Algorithm's most important property, the Level Construction stage and Path Augmenting Stage: Dinic's Algorithm always categorizes vertices based on their distances from source (max distance considered capped at depth), and then only consider vertices at the next level in distance for path augmenting candidates.

For both of our approaches, this implies that despite we are parallelizing over different options during path augmentations, the number of times such that we need to choose which edge to augment is lower bounded by the depth of the graph, because we need at least this many edges to reach sink.

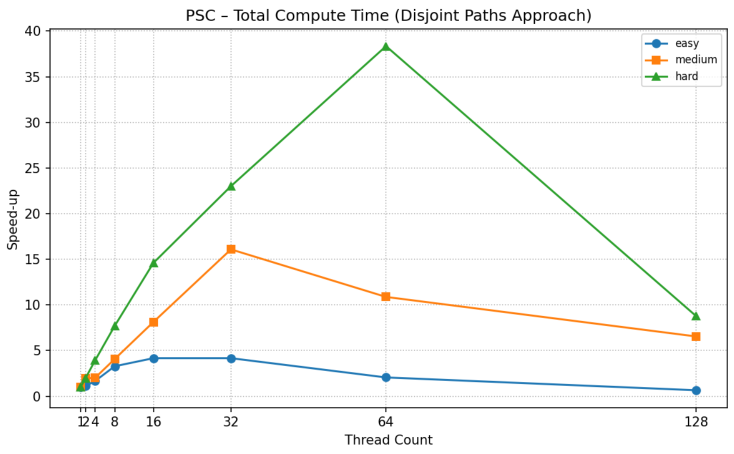
Thus, Disjoint Path benefits greatly from decrease in graph depth, as all of its path candidates' length decrease linearly w.r.t. the graph depth. Also since synchronization happens after the computatoin on each path candidates, the synchronization cost is not negatively affected by decrease in depth. Therefore, it is reasonable for us to observe such great improvement on speed up in Disjoint Paths approach.

On the other hand, since we kept the total number of vertices unchanged, decreased depth leads to increased number of vertices on each level. Therefore, for relatively the same pattern of edge connection among vertices, each vertices would have more in coming edges from the previous level. While Edge Flow Researvation approach also benefits from decreased path length and achieves speedup when graph depth decreases moderately, it suffers greatly from the increased synchronization cost as the number of in coming edges increase to a relatively large amount. Recall from sections above that Edge Flow Researvation approach need to use atomic operation at every passing edge in every augmenting path candidates to update and refund flow in the graph. Thus, as the incoming edges of vertices grow, there is a much high chance of contention for such atomic regions, causing threads to constantly waiting for each other, limiting the speed up to very little.

4.6 Scaling on PSC



(a) Edge Flow Reservation



(b) Disjoint Paths

Figure 9: Speedup Graphs on PSC

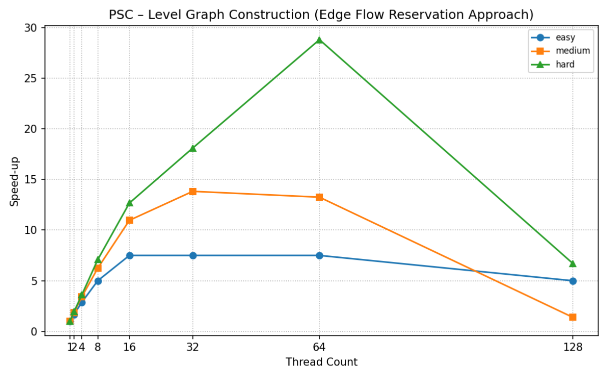
So far, all our experiments have been conducted on the GHC machines, and the scalability study could only be on up to 8 thread due to hardware limitation. To study the scalablity to a further extent, we evaluated the runtime and speedup of both approaches on the PSC machines against the same main inputs, for up to 128 threads. The results are shown in Figure 9.

Discussions:

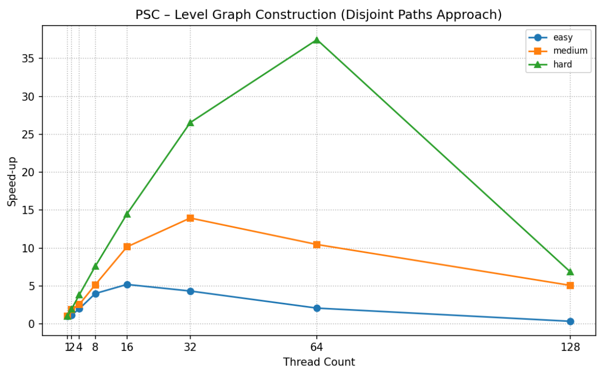
- As we can see, although the speedup improves slowly as the number of threads increases to 16, the performance of the edge flow reservation approach decreased significantly as the thread counts increased further, and becomes even slower than 1-thread performance at 128 threads. This in fact aligns with the GHC experiments because it shows that this implementation have issues of introducing extra works that also scale with thread count. Also, we can recall that this approach requires all accesses to the edge flow information in the residual graph to be atomic. When more threads are used, there is a much larger change of multiple threads attempting to access the same edge, making these atomic accesses sequential and costly, further decreasing the performance.

- We also noticed that the relative positions of the speedup curves are different from the GHC results in the first subsection, with the easy input having the lowest speedup on lower thread counts. We attribute this to the fact that single PSC cores have worse performance than the GHC cores. There are less parallelism in easy inputs since there are less vertices and vertices of lower outgoing degrees, so the high single-core perfomrance on GHC machines boosts performance on easy inputs more than the other two, since the slowest single-core determines the total runtime.
- For the speedup trends of the disjoint paths approach, we observe robust increases first, but the speedup peaked for all 3 inputs and then the speedup decreased. In the first half the curves, we can see that there are significant separation in the speedups: the easy input speedup peaks at less than 5x, while the hard input speedup peaks at nearly 40x. This vast difference confirms our earlier analysis in the GHC experiments as well as in the sensitivity studies that the parallelization capability of the disjoint paths approach is sensitive to input density.
- The decrease in speedup trend when the thread count grows even larger is likely caused by the increasing contention on the data races. A higher thread count increases the possibility of races on the atomic capture of the next-edge pointer of the same vertex. Also, the larger drop from 64 threads to 128 threads could also be the effect of across-node communication. The PSC machine has a maximum of 64 threads per node, so when 128 threads are used, the cost of communication due to updates in shared address spaces increase, further decreasing performance.

4.7 Individual Phases

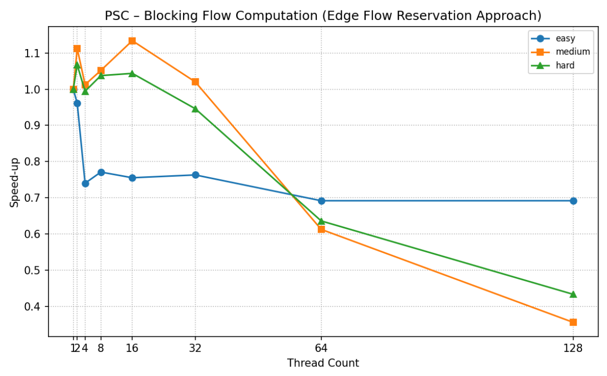


(a) Edge Flow Reservation

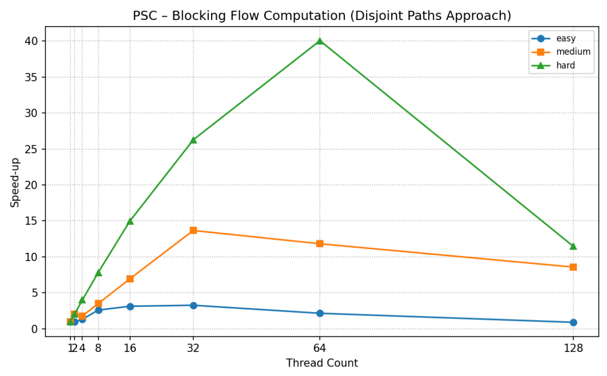


(b) Disjoint Paths

Figure 10: Level Graph Construction Phase Speedup Graphs on PSC



(a) Edge Flow Reservation



(b) Disjoint Paths

Figure 11: Blocking Flow Computation Phase Speedup Graphs on PSC

So far, our analysis has been focused on the total compute time speedups. However, Dinic's algorithm specifically feature alternating two-phase computations, and we have parallelized the level graph construction phase and blocking flow computation phase separately in our implementations. It is therefore of our interest how the speedup within each phase scales with the thread count.

Using the same main experiment inputs, we ran the experiments again on PSC, with separate timing for the two phases (the time spent in each phase is added up). The speedup trends are shown in figure 10 and figure 11.

Discussions:

- In addition to the speedup trends, we also found that for both edge flow reservation and disjoint paths approaches, the time that the program spent in each phase are approximately half and half (within 10% of even split every time). This suggests that the scalablity in both phases are equally important for a parallel Dinic's algorithm to have good scalability.
- For both approaches (which actually have the same implementation for this phase), the level graph construction phase speedup resembled the speedup trend of the disjoint paths approach on PSC: the curves first increase steadily, peak, and then decrease; and, the scalability is better for harder inputs. In general, the level-graph construction phase benefits from input in the same way as the disjoint-paths approach: they both parallelize over neighbors, and hence edges going out of one or a group of vertices. The larger the average degree in the input graph, the more parallelization and better workload balance (from fine-grained dynamic assignment) they can have. The decrease in performance at higher thread counts can be similarly attributed to increased contentions, because the level graph construction phase needed to properly handle vertices reached by multiple threads.
- For the disjoin paths approach, the speedup plot resembles the total computation, which is obviously expected. We analyzed the shape of those curves in the previous section.
- For the edge flow reservation approach, the speedup performance is even worse when examining the blocking flow computation phase. The highest speedup ever achieved is only about 1.1, with most data points below 1. This shows that for the edge flow reservation approach, most total compute time speedup came from the other phase. The significant work introduced

by the necessity to rollbacks and the frequent data race harm the performance of this parallel implementation even at low thread counts, showing the degree of severity of this issue.

5 References

Lecture 12 Notes, Spring 2023, 15-451, Carnegie Mellon University:
`https://www.cs.cmu.edu/~15451-s23/lectures/lec12-flow2.pdf`

Lecture 12 Notes, Spring 2025, 15-451, Carnegie Mellon University:
`https://www.cs.cmu.edu/~15451-s25/notes/lecture12.pdf`

6 Work Distribution

The two team members (Gary Gao and Mingxuan Li) each contributed to 50% of this project.

Specifically,

- Both conducted background research and contributed in brainstorming.
- Both implemented sequential code.
- Mingxuan implemented the first phase parallelization and the first version of the second phase parallelization. Gary implemented input generation and the second phase parallelization and the first version. Both debugged together.
- The experiments and contents on the report are divided up equally between the two.