15-418 Project Proposal

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

Basic Information

- Title: Parallelization and Analysis of Algorithm for Maximum Flow Problems
- Group Members: Gary Gao (wgao2), Mingxuan Li (mingxua3)
- Project Page: https://garygao33.github.io/ParallelFlow/

Short Summary

This project aims to implement and parallelize algorithm(s) for maximum flow problems using shared address space framework (OpenMP) on CPU. Detailed analyses of the algorithm's performance characteristics will also be performed.

Backgrounds

The maximum flow problem is a foundational problem in combinatorial optimization, with wide-ranging applications in computer networks, transportation systems, computer vision, bipartite matching, and more. 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.

Several algorithms exist for solving this problem, including the Ford-Fulkerson method and its refinements such as Edmonds-Karp. However, one of the most efficient and structurally suitable algorithms for parallelization is Dinic's algorithm. Unlike Ford-Fulkerson variants that augment one path at a time, Dinic's algorithm works in phases, each of which involves constructing a level graph via BFS and computing a blocking flow in that graph. These structured phases and the potential for simultaneous flow augmentations make Dinic's algorithm have a potential for parallel execution.

The high-level pseudocode for Dinic's algorithm is as follows:

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

Dinic's algorithm offers some major points of parallelism:

- Parallel BFS (Level Graph Construction): The BFS traversal used to build the level graph can be parallelized using a level-synchronous model. All nodes in the current frontier can explore their neighbors concurrently. This step benefits from atomic updates or bitmask-based visited arrays.
- Parallel Blocking Flow Computation: Once the level graph is built, multiple augmenting paths can be found simultaneously using concurrent DFS routines. As long as the paths are edge-disjoint or non-conflicting, flow can be pushed in parallel and the residual graph updated using lock-free or atomic operations. This step may involve node or edge marking to prevent contention and ensure correctness.

Challenges

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 atomic operations 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 DFS-based 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.

Dinic's algorithm alternates between two phases: level graph construction (BFS) and blocking flow computation (DFS). The BFS 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 DFS phase is irregular, with each thread maintaining its own path stack. Access patterns here are less predictable and often pointer-chasing through adjacency lists. 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 or atomic operations on shared graph structures. Cache locality is poor during DFS due to scattered memory access, and contention on frequently accessed data structures (like adjacency lists or level arrays) can reduce performance. Balancing work across threads in the presence of irregular subgraph sizes is another difficulty.

Resources

We will use the GHC machines (which have up to 8 cores) and the PSC machines (which support up to 128 threads) for experiments with parallel algorithms. This would allow us to perform analyses on the performances in a similar fashion to the recent programming assignments.

We will be implementing the algorithm from scratch, using C++ and OpenMP. We will refer to lecture notes from 15-451 (which are publicly available on the course website) for problem setup and algorithm overview. We will also implement testing code with some reference to the programming assignments in this course.

Currently, we do not see any necessity in using additional resources, including special machines.

Goals and Deliverables

We plan to achieve the following goals on :

- Implement a successful and efficient sequential version of the Dinic's algorithm.
- Implement a parallel version of the algorithm using OpenMP that can achieve reasonable speedup (we are unable to provide specific speedup numbers without further exploration).
- We will run experiments on GHC and PSC machines to analyze the performance. The experiments will include inputs of differing characteristics (e.g. graph size, density of the edges, edge capacities, etc.).

If we have extra time, we may explore parallelization of the Edmonds-Karp algorithm, or try to implement a parallel version for GPU using CUDA.

For the analysis component, our aim is to understand how well the algorithm adapts to parallel execution on a shared-memory system using OpenMP. Specifically, some questions are: how do different phases of the algorithm, such as level graph construction (BFS) and blocking flow computation (DFS), scale with increasing thread count, and which parts become bottlenecks due to synchronization or load imbalance. We also hope to analyze how graph structure (e.g., density, degree distribution, or size) affects parallel performance and whether certain graph types benefit more from parallelism than others. These insights will inform both algorithm design and system-level optimization strategies for irregular workloads such as graph processing.

Platform Choice

As mentioned in previous sections, we will be using C++ and OpenMP for implementation and the GHC and PSC machines for experiments and performance analysis. Using these readily available machines not only minimizes the cost of this project but also facilitates experimental processes considering our familiarity with these machines.

C++ with OpenMP is well suited to implement Dinic's algorithm due to its combination of performance, control, and shared memory systems. C++ provides efficient data structures and low-level memory control, which is important for handling irregular access patterns of graph algorithms. OpenMP allows for efficient parallelization of key components through pragma directives, while also supporting thread-private variables and atomic operations needed to safely update shared data such as residual capacities. This platform is ideal for exploiting the coarse-grained parallelism of Dinic's algorithm on multicore CPUs without the complexity of managing threads manually or dealing with GPU-specific challenges like control-flow divergence.

Schedule

- Week 1 (Mar 26 - Apr 1): Review the problem setup and algorithm. Brainstorm possible implementations.
- Week 2 (Apr 2 - Apr 8): Implement a sequential version of the algorithm. Produce basic suites of test inputs of different characteristics for performance analysis.
- Week 3 (Apr 9 - Apr 15): Implement initial, basic parallel implementations. Analyze the weaknesses and bottlenecks to prepare for improved versions. Complete the milestone report.
- Week 4 (Apr 16 - Apr 22): Improve parallel implementation. Finish building all test inputs for complete analysis.
- Week 5 (Apr 23 - Apr 28): Run the experiments and perform analysis. Complete the final report and the poster.