CMSC858N Final Project Report

Richard Zha

May 2024

1 Introduction

Max-flow min-cut is a fundamental algorithm to many classic computer vision problems, which range in difficulty from simple grid graph cases like image segmentation, to multi-camera scene reconstruction, surface fitting, and dynamic vision. In general, computer vision max flow instances are used to compute labels of every pixel in an image, according to a pre-specified energy function that describes interaction potential with other pixels. Each pixel is thus represented as a vertex in the graph, with edges linking adjacent pixels together. For these reasons, we are usually not interested in the max flow, but instead the minimum $s - t$ cut of the constructed graph.

2 Prior Work

2.1 Sequential Max Flow

Sequential max-flow is a very well studied problem in computer science with many existing algorithms. These usually fall into two distinct families of algorithms: (1) augmenting path-based algorithms and (2) pre-flow based methods.

Augmenting path algorithms make use of the Ford-Fulkerson Theorem: a given flow value f is maximum if and only if there is no augmenting path in the residual graph G_f. Therefore, these algorithms exhaust all augmenting paths in G_f, then return the max flow value at termination. Roughly speaking, the major distinction between algorithms in this family is the method of finding augmenting paths in the residual graph.

Pre-flow based methods relax the flow conservation constraint, and allow for *excess* at each vertex. Excess is defined as being the difference between the flow going into and coming out of a node. The most successful sequential algorithm in this area is the push-relabel algorithm due to Goldberg and Tarjan. It operates via two primitives: *push* and *relabel*. Whenever there is excess at a node, we try to push it one of its neighbors by transferring excess to the neighbor vertex. In order to push inefficiently, we maintain *labels* for all vertices that estimate the distance from the sink. We only allow for pushes from a higher label to a lower label. If an edge isn't saturated, and we fulfill the label condition, then the edge is *admissable*. If a vertex pushes along all edges and still has excess, we *relabel* it and continue to push. A vertex is considered *active* if it still has positive excess. When there are no active vertices remaining, the algorithm terminates.

In the context of computer vision, the standard sequential augmenting path algorithm used for graph-cuts is called the Boykov-Kolmogorov (BK) algorithm. Conceptually, the BK algorithm operates by growing two spanning trees from the source and sink concurrently. When the spanning trees intersect, this implies that an augmenting path exists. The algorithm pushes flow along this path, and continues to perform this procedure until termination.

The BK algorithm is well documented to perform well in practice, and is still the standard for graph-cuts amongst many computer vision practitioners. However, it is worth noting that this algorithm has no strongly polynomial bound, in contrast to many other standard max flow algorithms. This deficiency makes for a very interesting comparison between theory and practice in computer vision graph cuts.

Another algorithm that is found to perform well is the commercial GraphCut algorithm. Recent literature has found the sequential version of the algorithm to yield the best performance on specifically grid graph graph cut instances, and is competitive in many other problems.

2.2 Parallel Max Flow

Most practical implementations of parallel max flow are built on the pre-flow class of algorithms. This class of algorithms lends to better parallelism due to how both the push and relabel primitives only perform localized updates at a vertex. Augmenting path algorithms, on the other hand, naturally produce a longer chain of sequential dependencies, which results in less opportunities for parallelism.

Within the class of parallel push relabel algorithms, the main distinction is whether or not the algorithm synchronizes threads. In synchronous parallel push relabel, the algorithm processes the set of active vertices in rounds. That is, we make all available pushes and relabels from the active set in parallel, in lockstep between rounds. In asynchronous parallel push relabel, each thread is responsible for all push and relabel operations at a given vertex.

Notable implementations of the above are [2] for synchronous parallel push relabel, and [3] for asynchronous parallel push relabel.

3 Implementation

The rest of this report will be focused on the parlaylib implementation of the parallel push-relabel algorithm described in [2] and [1]. The code is in a github repository: repo.

3.1 Augmented vertices and edges

In order to avoid using locks and to ensure that all necessary operations are efficient, we augment the data structures for representing an edge and vertex. In terms of C++ structs, these are:

```
struct Edge {
    int capacity;
    int flow;
    int residual_capacity;
    Edge *residual_edge;
    int u;
    int v;
};
```

```
struct Vertex {
    int label; // d(v) in paper
    int newlabel; // d'(v) in paper
    int excess; // e(v) in paper
    atomic int addedExcess; // e'(v) in paper
    atomic bool isDiscovered;
    int work;
    int current;
    sequence<Edge> edges;
    sequence<int> discoveredVertices;
};
```

We use an adjacency list representation of the graph. At index u, we store a vertex object which holds a parlaylib sequence of edges for the corresponding vertex with id u.

3.2 Synchronous Parallel Push Relabel (SPPR) Algorithm

Below is the (general) pseudocode for the full synchronous parallel push relabel algorithm. Note that because this is a preflow algorithm, it only returns the maximum *preflow*, and not the maximum flow. The max flow can be recovered in $O(m)$ time due to the fact that the algorithm does correctly compute the min-cut.

The general idea is to establish an initial labeling via a global relabeling. Then, in parallel, we push and relabel (when needed), collecting the next set of active vertices. Whenever we cross the work threshold, do a global relabeling. When the active set is empty, the program terminates. The algorithm will return the excess at the sink.

```
// Synchronous Parallel Push Relabel
int sppr() {
    initialize augmented vertices and edges
    active = empty
    global_relabel()
    add source to active
    workSinceGR = 0
    while active is not empty {
        for each vertex v in the active set, in parallel {
            v.work = 0
            while v.excess > 0 and there are admissible edges (v,w) {
                push() flow along (v,w) to w.addedExcess
                if(!w.isDiscovered and CAS(w.isDiscovered, true)) {
                    v.discoveredVertices.push(w)
                }
            }
            if v.excess > 0 {
                if(!v.isDiscovered and CAS(v.isDiscovered, true)) {
                    v.discoveredVertices.push(v)
                }
            }
            if no more admissable edges {
                relabel(v)
            }
            v.work += outDeg(v) + beta // beta = 12 usually
        }
        for each vertex v in the active set, in parallel {
            add v.addedExcess to v.excess
            v.addedExcess = 0
            v.isDiscovered = false
        }
        active = flatten(v.discoveredVertices for all v)
        // global relabeling
        workSinceGR = reduce(v.work for all v)
        if(workSinceGR is above a threshold) {
            workSinceGr = 0
            global_relabel()
        }
    }

    // max preflow
    return vertices[sink].excess
}
```

3.3 Lock-free push and relabel

Lock free pushes and relabels are achieved by utilizing augmented vertices. In particular, we make use of a few facts:

1. Lock free push: because the algorithm parallelizes over the set of active vertices, when we try to perform a push operation on edge (u, v), we only need to ensure that we atomically update the excess value of v. This is because no vertex will be able to push to u and we only push along a given edge from u. The atomic variable `addedExcess` handles this.

2. Lock free relabel: to avoid a data race when vertex reads and writes overlap for labels, we use a shadow

copy of the label variable, `newlabel`. We only use this variable when setting a new label during a relabel operation, and only use the `label` field when we read labels.

3. Lock free active set management: we avoid adding duplicates to the set of active vertices from the previous set by using the `isDiscovered` boolean and `discoveredVertices` sequence local to each vertex. Because we use a CAS, we avoid having the same vertex added twice when we flatten at a later step.

3.4 Global Relabeling

Global relabeling is a heuristic used to improve the accuracy of labels throughout the execution of the algorithm. Inaccurate distance labels result in more iterations being taken to push from an active vertex, because edges may not be admissible until the correct distance label is found. Global relabeling simply runs a reverse breadth-first search from the sink through all non-saturated edges, giving each vertex its exact distance as its labeling.

Global relabeling is clearly very expensive, so its cost is amortized by only running it very sparsely throughout execution. The paper uses a *work* heuristic for each active vertex. At the end of a round of active vertices, if the total work since the last global relabel exceeds a threshold, we do a global relabel on that iteration.

The pseudocode for this in parlay is:

```
global_relabel() {
    d = tabulate() sequence of n ints, init to all to n
    frontier = sequence of vertices
    level = 0
    d[sink] = 0
    while (frontier is not empty) {
        level++
        out = flatten(map(frontier, u -> return u.edges))  // map frontier to edges
        // get next frontier, and assign their labels
        frontier = map(filter(out, e = (u,v) ->
            return !(e is satisfied) && CAS(d[v], level) && d[v] == n
        ))
    }
    for each vertex, in parallel {
        v.label = d[v]
    }
    active = filter(vertices, u -> return u.label > 0 && u.label < n && u.excess > 0)
    workSinceLastGR = 0
}
```

4 Results

The SPPR algorithm is benchmarked against the BK algorithm as a sequential baseline. SPPR was written using parlaylib. The BK algorithm uses an implementation available in Boost.

4.1 Hardware

All runs were done on a Dell T620 PowerEdge server, with Dual Intel Xeon E5-2643v2 processors, each with 6 cores at 3.5GHz using the Haswell architecture. All instances fit into RAM, as the server has 256Gb RAM available.

This server is quite old (I would estimate on the order of 10 years old). So, it is worth noting that absolute timing results are likely much better for both the BK algorithm and the SPPR algorithm. From basic testing on smaller graphs, the server is around 10x slower than my M1 Macbook Pro.

4.2 Benchmarks

All flow instances were sourced from the UWaterloo computer vision repository. The goal was to try a mix of larger and smaller vision instances, with varying $s - t$ path lengths.

Dataset	**Instance Type**	**Notes**	**Num. vertices**	**Num. edges**
BVZ-tsukuba15	Stereo	-	110594	518994
KZ2-venus20	Stereo	Both dense and sparse arcs	315628	2162357
BL06-camel-sml	Multi-view Reconstruction	-	1209602	5963582
BL06-camel-lrg	Multi-view Reconstruction	Cell complex structure	18900002	93749846
liver.n26c100	Segmentation	Short $s - t$ paths	4161602	108370821
adhead.n26c100	Segmentation	Long $s - t$ paths	12582914	327484556
LB067-bunny-lrg	Surface Fitting	Short $s - t$ paths	49544354	300838741

Table 1: Flow instances

4.3 Evaluation

Only a single run time is listed for all implementations. If multiple runs completed, we use the worst timing for all instances.

Table 2: SPPR Performance and Self Speedup

Dataset	Execution Time (seconds)		Speedup
	1 Thread	24 Threads	
BVZ-tsukuba15	4.4317	0.7689	5.897
KZ2-venus20	12.3682	1.6690	7.533
BL06-camel-sml	65.7572	7.8165	8.660
BL06-camel-lrg	2155.2748	232.8624	9.256
liver.n26c100	413.7807	31.3183	14.913
adhead.n26c100	-	61.7282	-
LB067-bunny-lrg	-	119.6573	-

Table 3: SPPR vs BK Speedup

Dataset	Execution Time (seconds)		Speedup
	SPPR (24 Threads)	BK	
BVZ-tsukuba15	0.7689	0.0093	-
KZ2-venus20	1.6690	4.521	2.708
BL06-camel-sml	7.8165	27.6144	3.537
BL06-camel-lrg	232.8624	1193.7129	5.128
liver.n26c100	31.3183	300.4233	9.587
adhead.n26c100	61.7282	947.764	15.357
LB067-bunny-lrg	119.6573	452.9831	3.784

5 Analysis

In terms of self speedup, the results indicated that there was notable improvements increasing the number of threads. 24 Threads were the default assigned by parlaylib, so those were used for evaluation. Both the adhead.n26c100 and liver.n26c100 datasets were run overnight, but neither completed within 6 hours for the single threaded version of SPPR.

Results compared to the BK algorithm show that besides on incredibly small datasets, e.g BVZ-tsukuba15, the SPPR algorithm achieves reasonable speedup over BK for most cases. This supports the idea found by [4] that most parallel algorithms are only practical are larger datasets. The two main outliers were the adhead.n26c100 dataset, and the liver.n26c100 data set. Both of these datasets were 26-connected grids, but as described in the problem instance section, adhead.n26c100 has many longer paths, while liver.n26c100 has many short paths. Given that the BK algorithm grows spanning trees from the source and sink, we expect and see that performance should degrade on ahead.n26c100.

References

[1] Niklas Baumstark. Speeding up maximum flow computations on shared memory platforms, 2014.

[2] Niklas Baumstark, Guy Blelloch, and Julian Shun. Efficient implementation of a synchronous parallel push-relabel algorithm. In *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 106–117. Springer, 2015.

[3] Bo Hong and Zhengyu He. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1025–1033, 2010.

[4] Patrick M. Jensen, Niels Jeppesen, Anders B. Dahl, and Vedrana A. Dahl. Review of serial and parallel min-cut/max-flow algorithms for computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2):2310–2329, 2023.