Shared-memory Exact Minimum Cuts

Monika Henzinger
University of Vienna
Vienna, Austria
monika.henzinger@univie.ac.at

Alexander Noe
University of Vienna
Vienna, Austria
alexander.noe@univie.ac.at

Christian Schulz
University of Vienna
Vienna, Austria
christian.schulz@univie.ac.at

***Abstract*—The minimum cut problem for an undirected edge-weighted graph asks us to divide its set of nodes into two blocks while minimizing the weighted sum of the cut edges. In this paper, we engineer the fastest known exact algorithm for the problem.**

State-of-the-art algorithms like the algorithm of Padberg and Rinaldi or the algorithm of Nagamochi, Ono and Ibaraki identify edges that can be contracted to reduce the graph size such that at least one minimum cut is maintained in the contracted graph. Our algorithm achieves improvements in running time over these algorithms by a multitude of techniques. First, we use a recently developed fast and parallel *inexact* minimum cut algorithm to obtain a better bound for the problem. Afterwards, we use reductions that depend on this bound to reduce the size of the graph much faster than previously possible. We use improved data structures to further lower the running time of our algorithm. Additionally, we parallelize the contraction routines of Nagamochi et al. . Overall, we arrive at a system that significantly outperforms the fastest state-of-the-art solvers for the *exact* minimum cut problem.

I. Introduction

Given an undirected graph with non-negative edge weights, the *minimum cut problem* aims at partitioning the vertices into two sets so that the sum of edge weights between the two sets is minimized. The problem has applications in many fields. In particular, for network reliability [17, 31], assuming equal failure probability edges, the smallest edge cut in the network has the highest chance of disconnecting the network; in VLSI design [22], a minimum cut can be used to minimize the number of connections between microprocessor blocks; and it is further used as a subproblem in the branch-and-cut algorithm for solving the Traveling Salesman Problem and other combinatorial problems [28].

As the minimum cut problem has many applications and is often used as a subproblem for complex problems, it is highly important to have algorithms that can solve the problem in reasonable time on huge data sets. As data sets are growing substantially faster than processor speeds, a good way to achieve this is efficient parallelization. While there is a multitude of algorithms which solve the minimum cut problem exactly on a single core [12, 15, 18, 25], to the best of our knowledge, there exists only one parallel exact algorithm for the minimum cut problem: Karger and Stein [18] present a parallel variant for their random contraction algorithm [18] which computes a minimum cut with high probability in polylogarithmic time using n^2 processors. This is, however, infeasible for large instances. There has been a MPI implementation of this algorithm by Gianinazzi et al. [9]. However, there have been no parallel implementations of the algorithms of Hao et al. [12] and Nagamochi et al. [25, 26], which outperformed other exact algorithms by orders of magnitude [6, 14, 16], both for real-world and generated networks.

All algorithms that solve the minimum cut problem exactly have non-linear running times, currently the fastest being the deterministic algorithm of Henzinger et al. [15] with running time $\mathcal{O}(m \log^2 n \log\log^2 n)$. There is a linear time approximation algorithm, namely the $(2+\varepsilon)$-approximation algorithm by Matula [24] and a linear time heuristic minimum cut algorithm by Henzinger et al. [14] based on the label propagation algorithm [30]. The latter paper also contains a shared-memory parallel implementation of their algorithm.

A. Contribution.

We engineer the fastest known *exact* minimum cut algorithm for the problem. We do so by (1) incorporating recently proposed *inexact* methods, (2) by using a novel combination of data structures and other optimizations as well as (3) parallelization.

Algorithms like the algorithm of Padberg and Rinaldi or the algorithm of Nagamochi, Ono and Ibaraki identify edges that can be contracted to reduce the graph size such that at least one minimum cut is maintained in the contracted graph. Our algorithm achieves improvements in running time by a multitude of techniques. First, we use a recently developed fast and parallel *inexact* minimum cut algorithm [14] to obtain a good approximate bound $\hat{\lambda}$ for the problem. As several graph reduction techniques depend on this bound, the better bound enables us to apply more reductions and reduce the size of the graph much faster. For example, edges whose incident vertices have a connectivity of at least $\hat{\lambda}$ can be contracted without affecting the minimum cut. Using a novel combination of well suited data structures as well as incorporating observations that help to save a significant amount of work in the contraction routine of Nagamochi et al. [26] further reduce the running time of our algorithm. For example, we observe a significantly higher performance on some graphs when using a FIFO bucket priority queue in combination with bounded priority queues as well as better bounds $\hat{\lambda}$. Additionally, we give a parallel variant of the contraction routines of Nagamochi et al. [26]. Overall, we arrive at a system that outperforms the state-of-the-art by a factor of up to 2.5 sequentially, and when run

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

in shared-memory parallel by a factor of up to 12.9 using 12 cores.

The rest of the paper is organized as follows. Chapter II gives preliminaries, an overview over related work and details of the algorithms of Nagamochi et al. [25, 26] and Henzinger et al. [14], as we make use of their results. Our shared-memory parallel exact algorithm for the minimum cut problem is detailed in Chapter III. In Chapter IV we give implementation details and extensive experiments both on real-world and generated graphs. We conclude the paper in Chapter V.

II. Preliminaries

A. Basic Concepts.

Let $G = (V, E, c)$ be a weighted undirected graph with vertex set V, edge set $E \subset V \times V$ and non-negative edge weights $c : E \to \mathbb{N}$. We extend c to a set of edges $E' \subseteq E$ by summing the weights of the edges; that is, $c(E') := \sum_{e=(u,v)\in E'} c(u, v)$. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges in G. The *neighborhood* $N(v)$ of a vertex v is the set of vertices adjacent to v. The *weighted degree* of a vertex is the sum of the weights of its incident edges. For brevity, we simply call this the *degree* of the vertex. For a set of vertices $A \subseteq V$, we denote by $E[A] := \{(u, v) \in E \mid u \in A, v \in V \setminus A\}$; that is, the set of edges in E that start in A and end in its complement. A cut $(A, V \setminus A)$ is a partitioning of the vertex set V into two non-empty *partitions* A and $V \setminus A$, each being called a *side* of the cut. The *capacity* of a cut $(A, V \setminus A)$ is $c(A) = \sum_{(u,v)\in E[A]} c(u, v)$. A *minimum cut* is a cut $(A, V \setminus A)$ that has smallest capacity $c(A)$ among all cuts in G. We use $\lambda(G)$ (or simply λ, when its meaning is clear) to denote the value of the minimum cut over all $A \subset V$. For two vertices s and t, we denote $\lambda(G, s, t)$ as the smallest cut of G, where s and t are on different sides of the cut. The connectivity $\lambda(G, e)$ of an edge $e = (s, t)$ is defined as $\lambda(G, s, t)$, the connectivity of its incident vertices. This is also known as the *minimum s-t-cut* of the graph or the *connectivity* or vertices s and t. At any point in the execution of a minimum cut algorithm, $\hat{\lambda}(G)$ (or simply $\hat{\lambda}$) denotes the smallest upper bound of the minimum cut that an algorithm discovered until that point. For a vertex $u \in V$ with minimum vertex degree, the size of the *trivial cut* $(\{u\}, V \setminus \{u\})$ is equal to the vertex degree of u. Hence, the minimum vertex degree $\delta(G)$ can serve as initial bound.

Many algorithms tackling the minimum cut problem use *graph contraction*. Given an edge $(u, v) \in E$, we define $G/(u, v)$ to be the graph after *contracting edge* (u, v). In the contracted graph, we delete vertex v and all edges incident to this vertex. For each edge $(v, w) \in E$, we add an edge (u, w) with $c(u, w) = c(v, w)$ to G or, if the edge already exists, we give it the edge weight $c(u, w) + c(v, w)$.

B. Related Work.

We now review algorithms for the global minimum cut and related problems. A closely related problem is the *minimum s-t-cut* problem, which asks for a minimum cut with nodes s and t in different partitions. Ford and Fulkerson [7] proved that minimum s-t-cut is equal to maximum s-t-flow. Gomory and Hu [11] observed that the (global) minimum cut can be computed with $n - 1$ minimum s-t-cut computations. For the following decades, this result by Gomory and Hu was used to find better algorithms for global minimum cut using improved maximum flow algorithms [18]. One of the fastest known maximum flow algorithms is the push-relabel algorithm [10] by Goldberg and Tarjan.

Hao and Orlin [12] adapt the push-relabel algorithm to pass information to future flow computations. When a push-relabel iteration is finished, they implicitly merge the source and sink to form a new sink and find a new source. Vertex heights are maintained over multiple iterations of push-relabel. With these techniques they achieve a total running time of $O(mn \log \frac{n^2}{m})$ for a graph with n vertices and m edges, which is asymptotically equal to a single run of the push-relabel algorithm. Gabow [8] gives an algorithm with running time of $O(m + \lambda^2 n \log(n/\lambda))$.

Padberg and Rinaldi [27] give a set of heuristics for edge contraction. Chekuri et al. [6] give an implementation of these heuristics that can be performed in time linear in the graph size. Using these heuristics it is possible to sparsify a graph while preserving at least one minimum cut in the graph. If their algorithm does not find an edge to contract, it performs a maximum flow computation, giving the algorithm worst case running time $O(n^4)$. However, the heuristics can also be used to improve the expected running time of other algorithms by applying them on interim graphs [6].

Nagamochi et al. [25, 26] give a minimum cut algorithm which does not use any flow computations. Instead, their algorithm uses maximum spanning forests to find a non-empty set of contractible edges. This contraction algorithm is run until the graph has only two remaining vertices. The algorithm has a running time of $O(mn + n^2 \log n)$. Stoer and Wagner [32] give a simpler variant of the algorithm of Nagamochi et al. [26], which has the same asymptotic time complexity. The performance of this algorithm on real-world instances, however, is significantly worse than the performance of the algorithms of Nagamochi et al. or Hao and Orlin, as shown in experiments conducted by Jünger et al. [16]. Both the algorithms of Hao and Orlin, and Nagamochi et al. achieve close to linear running time on most benchmark instances [6, 16]. To the best of our knowledge, there are no parallel implementation of either algorithm. Both of the algorithms do not have a straightforward parallel implementation.

Kawarabayashi and Thorup [19] give a deterministic near-linear time algorithm for the minimum cut problem, which runs in $O(m \log^{12} n)$. Their algorithm works by growing contractible regions using a variant of PageRank [29]. It was later improved by Henzinger et al. [15] to run in $O(m \log^2 n \log \log^2 n)$ time.

Based on the algorithm of Nagamochi et al., Matula [24] gives a $(2 + \varepsilon)$-approximation algorithm for the minimum cut problem. The algorithm contracts more edges than the algorithm of Nagamochi et al. to guarantee a linear time com-

plexity while still guaranteeing a $(2+\varepsilon)$-approximation factor. Karger and Stein [18] give a randomized Monte Carlo algorithm based on random edge contractions. This algorithm returns the minimum cut with high probability and a larger cut otherwise. In experiments, their algorithm was often outperformed by Nagamochi et al. and Hao and Orlin by orders of magnitude [6, 14, 16].

C. Nagamochi, Ono and Ibaraki's Algorithm.

We discuss the algorithm by Nagamochi, Ono and Ibaraki [25, 26] in greater detail since our work relies heavily on their results. The intuition behind the algorithm is as follows: imagine you have an unweighted graph with minimum cut value exactly one. Then any spanning tree must contain at least one edge of each of the minimum cuts. Hence, after computing a spanning tree, every edge not contained in the spanning tree can be contracted without losing the minimum cut. Nagamochi et al. extend this idea to the case where the graph can have edges with positive weight as well as the case in which the minimum cut is upper bounded by $\hat{\lambda}$. The first observation is the following: assume that you already found a cut in the current graph of size $\hat{\lambda}$ and you want to find out whether there is a cut of size $< \hat{\lambda}$. Then the contraction process only needs to ensure that the contracted graph contains all cuts having a value *strictly* smaller than $\hat{\lambda}$. To do so, Nagamochi et al. build *edge-disjoint maximum spanning forests* and contract all edges that are not in one of the $\hat{\lambda} - 1$ first spanning forests, as those connect vertices that have connectivity at least $\hat{\lambda}$. Note that the edge-disjoint maximum spanning forest *certifies* for any edge $e = (u, v)$ that is not in the forests that the minimum cut between u and v is at least $\hat{\lambda}$. Hence, the edge can be "safely" contracted. As weights are integer, this guarantees that the contracted graph still contains all cuts that are *strictly* smaller than $\hat{\lambda}$. Since it would be inefficient to directly compute $\hat{\lambda}-1$ edge disjoint maximum spanning forests, the authors give a modified algorithm to detect contractable edges faster. This is done by computing a lower bound on the connectivity of the endpoints of an edge which serves as a certificate for an edge to be contractable. The algorithm has worst case running time $\mathcal{O}\big(mn + n^2 \log n\big)$. In experimental evaluations [6, 14, 16] it is one of the fastest exact minimum cut algorithms, both on real-world and generated instances.

We now take a closer look at details of the algorithm. To find contractable edges, the algorithm uses a modified *breadth-first graph traversal* algorithm [25, 26]. More precisely, the algorithm starts at an arbitrary vertex. In each step, the algorithm visits (scans) the vertex v that is most strongly connected to the already visited vertices. For this purpose, a priority queue $\mathcal{Q}$ is used, in which the connectivity strength of each vertex $r : V \to \mathbb{R}$ to the already discovered vertices is used as a key. When scanning a vertex v, the value $r(w)$ is kept up to date for every unscanned neighbor w of v by setting i.e. $r(w) := r(w) + c(e)$. Moreover, for each such edge $e = (v, w)$, the algorithm computes a lower bound $q(e)$ for the connectivity, i.e. the smallest cut $\lambda(G, v, w)$, which places v and w on different sides of the cut. More precisely, as shown by [25, 26], if the vertices are scanned in a certain order (the order used by the algorithm), then $r(w)$ is a lower bound on $\lambda(G, v, w)$.

For an edge that has connectivity $\lambda(G, v, w) \geq \hat{\lambda}$, we know that there is no cut smaller than $\hat{\lambda}$ that places v and w in different partitions. If an edge e is not in a given cut $(A, V \backslash A)$, it can be contracted without affecting the cut. Thus, we can contract edges with connectivity at least $\hat{\lambda}$ without losing any cuts smaller than $\hat{\lambda}$. As $q(e) \leq \lambda(G, u, v)$ (lower bound), all edges with $q(e) \geq \hat{\lambda}$ are contracted.

Afterwards, the algorithm continues on the contracted graph. A single iteration of the subroutine can be performed in $\mathcal{O}(m + n \log n)$. The authors show that in each run, at least one edge of the graph can be contracted [25]. This yields a total running time of $O(mn + n^2 \log n)$. However, in practice the number of iterations is typically much less than $n - 1$, often it is proportional to $\log n$.

D. Inexact Shared-Memory Minimum Cuts.

`VieCut` [14] is a multilevel algorithm that uses a shared-memory parallel implementation of the label propagation algorithm [30] to find clusters with a strong intra-cluster connectivity. The algorithm then contracts these clusters as it is assumed that the minimum cut does not split a cluster, as the vertices in a cluster are strongly interconnected with each other. This contraction is followed by a linear-work shared memory run of the Padberg-Rinaldi local tests for contractible edges [27]. This whole process is repeated until the graph has only a constant number of vertices left and can be solved by the algorithm of Nagamochi et al. [26] exactly.

While `VieCut` can not guarantee optimality or even a small approximation ratio, in practice the algorithm finds near-optimal minimum cuts, often even the exact minimum cut, very quickly and in parallel. The algorithm can be implemented to have sequential running time $\mathcal{O}(n + m)$.

III. Fast Exact Minimum Cuts

In this section we detail our shared-memory algorithm for the minimum cut problem that is based on the algorithms of Nagamochi et al. [25, 26] and Henzinger et al. [14]. We aim to modify the algorithm of Nagamochi et al. [26] in order to find exact minimum cuts faster and in parallel. Their algorithm uses a routine described above in Section II-C, called CAPFOREST in their original work, in order to compute a lower bound $q(e)$ of the connectivity $\lambda(G, u, v)$ for each edge $e = (u, v)$.

If the connectivity between two vertices is at least as large as the current upper bound for the minimum cut, then it can be contracted. That also means that edges e with $q(e) \geq \hat{\lambda}$ can be safely contracted, The algorithm is guaranteed to find at least one such edge.

We start this section with optimizations to the sequential algorithm. First we use a recently published *inexact* algorithm to lower the minimum cut upper bound $\hat{\lambda}$. This enables us to save work and to perform contractions more quickly. We then give different implementations of the priority queue $\mathcal{Q}$

and detail the effects of the choice of queue on the algorithm. We show that the algorithm remains correct, even if we limit the priorities in the queue to $\hat{\lambda}$, meaning that elements in the queue having a key larger than that will not be updated. This significantly lowers the number of priority queue operations necessary. Then we adapt the algorithm so that we are able to detect contractible edges in parallel efficiently. Lastly, we put everything together and present a full system description.

A. Sequential Optimizations.

1) Lowering the Upper Bound $\hat{\lambda}$: Note that the upper bound $\hat{\lambda}$ for the minimum cut is an important parameter for exact contraction based algorithms such as the algorithm `NOI` of Nagamochi et al. [26]. Their algorithm computes a lower bound for the connectivity of the two incident vertices of each edge and contracts all edges whose incident vertices have a connectivity of at least $\hat{\lambda}$. Thus, it is possible to contract more edges if we manage to lower $\hat{\lambda}$ beforehand.

A trivial upper bound $\hat{\lambda}$ for the minimum cut is the minimum vertex degree, as it represents the trivial cut which separates the minimum degree vertex from all other vertices. We run `VieCut` to lower $\hat{\lambda}$ in order to allow us to find more edges to contract. Although `VieCut` is an *inexact algorithm*, in most cases it already finds the minimum cut [14] of the graph. As there are by definition no cuts smaller than the minimum cut, the result of `VieCut` is guaranteed to be at least as large as the minimum cut λ. As we set $\hat{\lambda}$ to the result of `VieCut` when running `NOI`, we can therefore guarantee a correct result.

A similar idea is employed by the linear time $(2+\epsilon)$-approximation algorithm of Matula [24], which initializes the algorithm of Nagamochi et al. [26] with $\hat{\lambda} = (\frac{1}{2} - \epsilon)\times$min degree.

2) Bounded Priority Queues: Whenever we visit a vertex, we update the priority of all of its neighbors in $\mathcal{Q}$ by adding the respective edge weight. Thus, in total we perform $|E|$ priority queue increase-weight operations. In practice, many vertices reach priority values much higher than $\hat{\lambda}$ and perform many priority increases until they reach their final value. We limit the values in the priority queue by $\hat{\lambda}$, i.e. we do not update priorities that are already $\hat{\lambda}$. Lemma 1 shows that this does not affect correctness of the algorithm.

Let $\tilde{q}_G(e)$ be the value $q(e)$ assigned to e in the modified algorithm on graph G and let $\tilde{r}_G(x)$ be the r-value of a node x in the modified algorithm on G.

Lemma 1. *Limiting the values in the priority queue $\mathcal{Q}$ used in the CAPFOREST routine to a maximum of $\hat{\lambda}$ does not interfere with the correctness of the algorithm. For every edge $e = (v, w)$ with $\tilde{q}_G(e) \geq \hat{\lambda}$, it holds that $\lambda(G, e) \geq \hat{\lambda}$. Therefore the edge can be contracted.*

Proof. Proof can be found in [13]. □

Lemma 1 allows us to considerably lower the number of priority queue operations, as we do not need to update priorities that are bigger than $\hat{\lambda}$. This optimization has even more benefit in combination with running `VieCut` to lower the upper bound $\hat{\lambda}$, as we further lower the number of priority queue operations.

3) Priority Queue Implementations: Nagamochi et al. [26] use an addressable priority queue $\mathcal{Q}$ in their algorithm to find contractible edges. In this section we now address variants for the implementation of the priority queue. As the algorithm often has many elements with maximum priority in practice, the implementation of this priority queue can have major impact on the order of vertex visits and thus also on the edges that will be marked contractible.

a) Bucket Priority Queue: As our algorithm limits the values in the priority queue to a maximum of $\hat{\lambda}$, we observe integer priorities in the range of $[0, \hat{\lambda}]$. Hence, we can use a bucket queue that is implemented as an array with $\hat{\lambda}$ buckets. In addition, the data structure keeps the id of the highest non-empty bucket, also known as the *top bucket*, and stores the position of each vertex in the priority queue. Priority updates can be implemented by deleting an element from its bucket and pushing it to the bucket with the updated priority. This allows constant time access for all operations except for deletions of the maximum priority element, which have to check all buckets between the prior top bucket and the new top bucket, possibly up to $\hat{\lambda}$ checks. We give two possible implementations to implement the buckets so that they can store all elements with a given priority.

The first implementation, `BStack` uses a dynamic array (`std::vector`) as the container for all elements in a bucket. When we add a new element to the array, we add it to the end of the array. $\mathcal{Q}$`.pop_max()` returns the last element of the top bucket (i.e. the element that was added last). Thus, our algorithm will always visit the element next whose priority was just increased. It thus does not fully explore all vertices in a region and instead behaves more similar to a depth-first search.

The other implementation, `BQueue` uses a double ended queue (`std::deque`) as the container instead. A new element is pushed to the back of the queue and $\mathcal{Q}$`.pop_max()` returns the *first* element of the top bucket. This results in a variant of our algorithm, which behaves more similar to a breadth-first search in that it first explores the vertices that have been discovered earlier, i.e. are closer to the source vertex in the graph.

b) Bottom-Up Binary Heap: A binary heap [35] is a binary tree (implemented as an array, where element i has its children in index $2i$ and $2i+1$) which fulfills the heap property, i.e. each element has priority that is not lower than either of its children. Thus the element with highest priority is the root of the tree. The tree can be made addressable by using an array of indices, in which we save the position of each vertex. We use a binary heap using the bottom-up heuristics [34], in which we sift down holes that were created by the deletion of the top priority vertex. Priority changes are implemented by sifting the addressed element up or down in the tree. Operations have a running time of up to $\mathcal{O}(\log n)$ to sift an element up or down to fix the heap property.

Algorithm 1 Parallel CAPFOREST

Input: $G = (V, E, c) \leftarrow$ undirected graph $\hat{\lambda} \leftarrow$ upper bound for minimum cut, $\mathcal{T} \leftarrow$ shared array of vertex visits
Output: $\mathcal{U} \leftarrow$ union-find data structure to mark contractible edges

```
1: Label all vertices v ∈ V "unvisited", blacklist B empty
2: ∀v ∈ V : r(v) ← 0
3: ∀e ∈ E : q(e) ← 0
4: Q ← empty priority queue
5: Insert random vertex into Q
6: while Q not empty do
7:     x ← Q.pop_max()                          ▷ Choose unvisited vertex with highest priority
8:     Mark x "visited"
9:     if T(x) = True then                      ▷ Every vertex is visited only once
10:        B(x) ← True
11:    else
12:        T(x) ← True
13:        α ← α + c(x) − 2r(x)
14:        λ̂ ← min(λ̂, α)
15:        for e = (x, y) ← unscanned edge, where y ∉ B do
16:            if r(y) < λ̂ ≤ r(y) + c(e) then
17:                U.union(x,y)                 ▷ Mark edge e to contract
18:            end if
19:            r(y) ← r(y) + c(e)
20:            q(e) ← r(y)
21:            Q(y) ← min(r(y), λ̂)
22:        end for
23:    end if
24: end while
```

In $\mathcal{Q}$`.pop_max()`, the `Heap` priority queue does not favor either old or new elements in the priority queue and therefore this implementation can be seen as a middle ground between the two bucket priority queues.

B. Parallel CAPFOREST.

We modify the algorithm in order to quickly find contractible edges using shared-memory parallelism. The pseudocode can be found in Algorithm 1. The proofs in this section show that the modifications do not violate the correctness of the algorithm. Detailed proofs for the original CAPFOREST algorithm and the modifications of Nagamochi et al. for weighted graphs can be found in [26].

The idea of the our algorithm is as follows: We aim to find contractible edges using shared-memory parallelism. Every processor selects a random vertex and runs Algorithm 1, which is a modified version of CAPFOREST [25, 26] where the priority values are limited to $\hat{\lambda}$, the current upper bound of the size of the minimum cut. We want to find contractible edges without requiring that every process looks at the whole graph. To achieve this, every vertex will only be visited by one process. Compared to limiting the number of vertices each process visits, this algorithm has the advantage that we also scan the vertices in sparse regions of the graph which might otherwise not be scanned by any process.

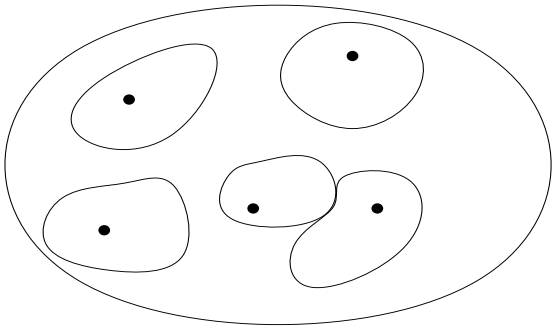

Fig. 1. Example run of Algorithm 1. Every process starts at a random vertex and scans region around the start vertex. These regions do not overlap.

Figure 1 shows an example run of Algorithm 1 with $p = 5$. Every process randomly chooses a start vertex and performs Algorithm 1 on it to "grow a region" of scanned vertices.

As we want to employ shared-memory parallelism to speed up the algorithm, we share an array $\mathcal{T}$ between all processes to denote whether a vertex has already been visited. If a vertex v has already been visited by a process, it will not be visited by any other processes. Additionally, every process keeps a local blacklist $\mathcal{B}$ for vertices that the process attempted to visit but that were already visited by another process before and were thus ignored by this process. Note that $\mathcal{B}$ is not shared between processes. We need this blacklist to ensure correctness, as a process may only contract edges that are

not adjacent to a vertex previously blacklisted by that process (proof in Lemma 2). For every vertex v we keep a value $r(v)$, which denotes the total weight of edges connecting v to already scanned vertices. Over the course of a run of the algorithm, every edge $e = (v, w)$ is given a value $q(e)$ (equal to $r(w)$ right after scanning e) which is a lower bound for the smallest cut $\lambda(G, v, w)$. We mark an edge e as contractible (more accurately, we union the incident vertices in the shared concurrent union-find data structure [1]), if $q(e) \geq \hat{\lambda}$. Note that this does not modify the graph, it just remembers which nodes to collapse. The actual node collapsing happens in a postprocessing step. Nagamochi and Ibaraki showed [26] that contracting only the edges that fulfill the condition in line 16 is equivalent.

As the set of disconnected edges is different depending on the start vertex, we looked into visiting every vertex by a number of processes up to a given parameter to find more contractible edges. However, this did generally result in higher total running times and thus we only visit every vertex once.

After all processes are finished, every vertex was visited exactly once (or possibly zero times, if the graph is disconnected). On average, every process has visited roughly $\frac{n}{p}$ vertices and all processes finish at the same time. We do not perform any form of locking of the elements of $\mathcal{T}$, as this would come with a running time penalty for every write and the worst that can happen with concurrent writes is that a vertex is visited more often, which does not affect correctness of the algorithm.

However, as we terminate early and no process visits every vertex, we can not guarantee anymore that the algorithm actually finds a contractible edge. However, in practice, this only happens if the graph is already very small (< 50 vertices in all of our experiments). We can then run the (sequential) CAPFOREST routine which is guaranteed to find at least one edge to contract.

In line 13 and 14 of Algorithm 1 we compute the value of the cut between the scanned and unscanned vertices and update $\hat{\lambda}$ if this cut is smaller than it. For more details on this we refer the reader to [26].

In practice, many vertices reach values of $r(y)$ that are much higher than $\hat{\lambda}$ and therefore need to update their priority in $\mathcal{Q}$ often. As previously detailed, we limit the values in the priority queue by $\hat{\lambda}$ and do not update priorities that are already greater than or equal to $\hat{\lambda}$. This allows us to considerably lower the number of priority queue operations per vertex.

Theorem 1. *Algorithm 1 is correct.*

As Algorithm 1 is a modified variant of CAPFOREST [25, 26], we use the correctness of their algorithm and show that our modifications can not result in incorrect results. In order to show this we need the following lemmas:

Lemma 2.
1) *Multiple instances of Algorithm 1 can be run in parallel with all instances sharing a parallel union-find data structure.*
2) *Early termination does not affect correctness.*
3) *For every edge $e = (v, w)$, where neither v nor w are blacklisted, $q(e)$ is a lower bound for the connectivity $\lambda(G, v, w)$, even if the set of blacklisted vertices $\mathcal{B}$ is not empty.*
4) *When limiting the priority of a vertex in $\mathcal{Q}$ to $\hat{\lambda}$, it still holds that the vertices incident to an edge $e = (x, y)$ with $q(e) \geq \hat{\lambda}$ have connectivity $\lambda(G, x, y) \geq \hat{\lambda}$.*

Proof. A run of the CAPFOREST algorithm finds a non-empty set of edges that can be contracted without contracting a cut with value less than $\hat{\lambda}$ [25]. We show that none of our modifications can result in incorrect results:

1) The CAPFOREST routine can be started from an arbitrary vertex and finds a set of edges that can be contracted without affecting the minimum cut λ. This is true for any vertex $v \in V$. As we do not change the underlying graph but just mark contractible edges, the correctness is obviously upheld when running the algorithm multiple times starting at different vertices. This is also true when running the different iterations in parallel, as long as the underlying graph is not changed. Marking the edge $e = (u, v)$ as contractible is equivalent to performing a `Union` of vertices u and v. The `Union` operation in a union-find data structure is commutative and therefore the order of unions is irrelevant for the final result. Thus performing the iterations successively has the same result as performing them in parallel.
2) Over the course of the algorithm we set a value $q(e)$ for each edge e and we maintain a value $\hat{\lambda}$ that never increases. We contract edges that have value $q(e) \geq \hat{\lambda}$ at the time when $q(e)$ is set. For every edge, this value is set exactly once. If we terminate the algorithm prior to setting $q(e)$ for all edges, the set of contracted edges is a subset of the set of edges that would be contracted in a full run and all contracted edges e fulfill $q(e) \geq \hat{\lambda}$ at termination. Thus, no edge contraction contracts a cut that is smaller than $\hat{\lambda}$.
3) Let $e = (v, w)$ be an edge and let $\mathcal{B}_e$ be the set of nodes blacklisted at the time when e is scanned. We show that for an edge $e = (v, w)$, $q(e) \leq \lambda(\bar{G}, v, w)$, where $\bar{G} = (\bar{V}, \bar{E})$ with vertices $\bar{V} = V \backslash \mathcal{B}_e$ and edges $\bar{E} = \{e = (u, v) \in E : u \notin \mathcal{B}_e \text{ and } v \notin \mathcal{B}_e\}$ is the graph G with all blacklisted vertices and their incident edges removed. As the removal of vertices and edges can not increase edge connectivities $q_{\bar{G}}(e) \leq \lambda(\bar{G}, v, w) \leq \lambda(G, v, w)$ and e is a contractible edge.
Whenever we visit a vertex b, we decide whether we blacklist the vertex. If we blacklist the vertex b, we immediately leave the vertex and do not change any values $r(v)$ or $q(e)$ for any other vertex or edge. As vertex b is marked as blacklisted, we will not visit the vertex again and the edges incident to b only affect $r(b)$. As edges incident to any of the vertices in $\mathcal{B}_e$ do not affect $q(e)$, the value of $q(e)$ in the algorithm with the blacklisted in G is equal to the value of $q(e)$ in $\bar{G}$, which does not contain the blacklisted vertices in $\mathcal{B}_e$

and their incident edges. On $\bar{G}$ this is equivalent to a run of CAPFOREST without blacklisted vertices and due to the correctness of CAPFOREST [26] we know that for every edge $e \in \bar{E} : q_{\bar{G}}(e) \leq \lambda(\bar{G}, v, w) \leq \lambda(G, v, w)$. Note that in $\bar{G}$ we only exclude the vertices that are in $\mathcal{B}_e$. It is possible that a node y that was unvisited when e was scanned might get blacklisted later, however, this does not affect the value of $q(e)$ as the value $q(e)$ is set when an edge is scanned and never modified afterwards.

4) Proof in Lemma 1.

We can combine the sub-proofs (3) and (4) by creating the graph $\bar{G}'$, in which we remove all edges incident to blacklisted vertices and decrease edge weights to make sure no $q(e)$ is strictly larger than $\hat{\lambda}$. As we only lowered edge weights and removed edges, for every edge between two not blacklisted vertices $e = (u, v)$, $q_G(e) \leq \lambda(\bar{G}', x, y) \leq \lambda(G, x, y)$ or $q_G(e) > \hat{\lambda}$ and thus we only contract contractible edges. As none of our modifications can result in the contraction of edges that should not be contracted, Algorithm 1 is correct. □

Parallel Graph Contraction: After using Algorithm 1 to find contractible edges, we use a concurrent hash table [23] to generate the contracted graph $G_C = (V_C, E_C)$, in which each block in $\mathcal{U}$ is represented by a single vertex: first we assign each block a vertex ID in the contracted graph in $[0, |V_C|)$. For each edge $e = (u, v)$, we compute a hash of the block IDs of u and v to uniquely identify the edge in E_C. We use this identifier to compute the weights of all edges between blocks. If there are two blocks that each have many vertices, there might be many edges between them and if so, the hash table spends considerable time for synchronization. We thus compute the weight of the edge connecting the two heavy blocks locally on each process and sum them up afterwards to reduce synchronization overhead. If the collapsed graph G_C has a minimum degree of less than $\hat{\lambda}$, we update $\hat{\lambda}$ to the value of this cut.

C. Putting Things Together.

Algorithm 2 shows the overall structure of the algorithm. We first run `VieCut` to find a good upper bound $\hat{\lambda}$ for the minimum cut. Afterwards, we run Algorithm 1 to find contractible edges. In the unlikely case that none were found, we run CAPFOREST [26] sequentially to find at least one contractible edge. We create a new contracted graph using parallel graph contraction, as shown in Section III-B. This process is repeated until the graph has only two vertices left. Whenever we encounter a collapsed vertex with a degree of lower than $\hat{\lambda}$, we update the upper bound. We return the smallest cut we encounter in this process.

Algorithm 2 Parallel Minimum Cut

Input: $G = (V, E, c)$

```
λ̂ ← VieCut(G), G_C ← G
while G_C has more than 2 vertices do
    λ̂ ← Parallel CAPFOREST(G_C, λ̂)
    if no edges marked contractible then
        λ̂ ← CAPFOREST(G_C, λ̂)
    end if
    G_C, λ̂ ← Parallel Graph Contract(G_C)
end while
return λ̂
```

If we also want to output the minimum cut, for each collapsed vertex v_C in G_C we store which vertices of G are included in v_C. When we update $\hat{\lambda}$, we store which vertices are contained in the minimum cut. This allows us to see which vertices are on one side of the cut.

IV. Experiments and Results

A. Experimental Setup and Methodology.

We implemented the algorithms using C++-17 and compiled all codes using g++-7.1.0 with full optimization (`-O3`). Our experiments are conducted on a machine with two Intel Xeon E5-2643 v4 with 3.4GHz with 6 CPU cores each and hyper-threading enabled, and 1.5 TB RAM in total. We perform five repetitions per instance and report average running time.

Performance plots relate the fastest running time to the running time of each other algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. The plots show the ratio $t_{\text{best}}/t_{\text{algorithm}}$ on the y-axis. A point close to zero indicates that the running time of the algorithm was considerably worse than the fastest algorithm on the same instance. A value of one therefore indicates that the corresponding algorithm was one of the fastest algorithms to compute the solution. Thus an algorithm is considered to outperform another algorithm if its corresponding ratio values are above those of the other algorithm. In order to include instances that were too big for an algorithm (as `HO-CGKLS` and `NOI-CGKLS` are limited to 32bit integers) we set the corresponding ratio below *zero*.

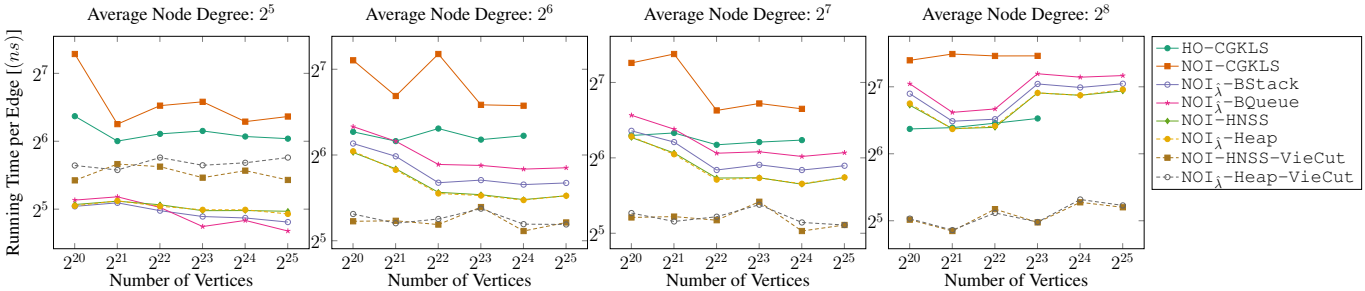

Fig. 2. Total running time in nanoseconds per edge in random hyperbolic graphs.

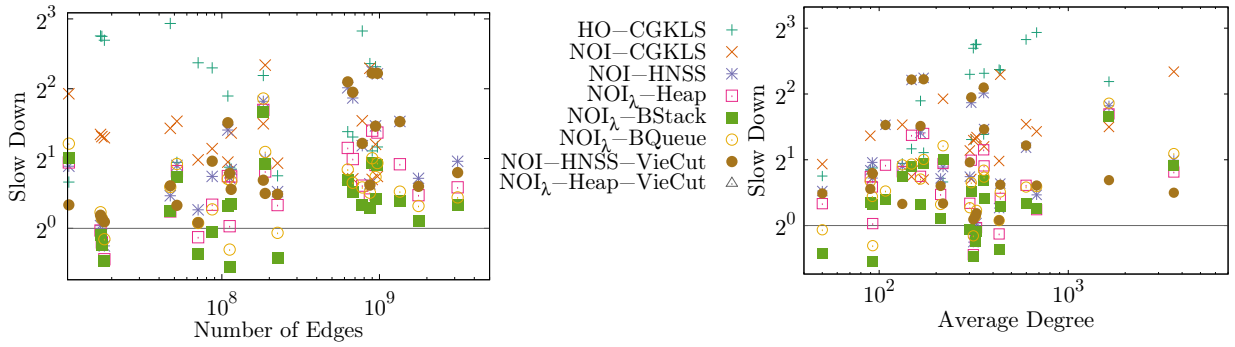

Fig. 3. Total running time in real-world graphs, normalized by the running time of NOI$_{\hat{\lambda}}$-Heap-VieCut.

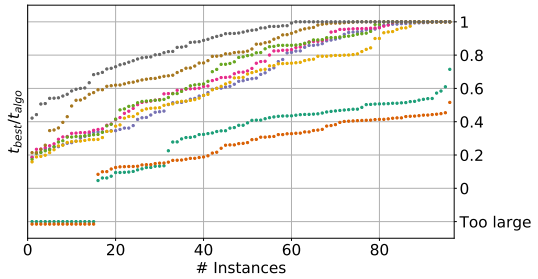

Fig. 4. Performance plot for all graphs (legend shared with Fig. 3 above).

Algorithms.: There have been multiple experimental studies that compare exact algorithms for the minimum cut problem [6, 14, 16]. All of these studies report that the algorithm of Nagamochi et al. [25, 26] and the algorithm of Hao and Orlin [12] outperform other algorithms, such as the algorithms of Karger and Stein [18] or the algorithm of Stoer and Wagner [32], often by multiple orders of magnitude. Among others, we compare our algorithms against two available implementations of the sequential algorithm of Nagamochi et al. [25, 26]. Henzinger et al. [14] give an implementation of the algorithm of Nagamochi et al. [25, 26], written in C++ (NOI-HNSS) that uses a binary heap. We use this algorithm with small optimizations in the priority queue as a base of our implementation. Chekuri et al. [6] give an implementation of the flow-based algorithm of Hao and Orlin using all optimizations given in the paper (variant ho in [6]), implemented in C, in our experiments denoted as HO-CGKLS. They also give an implementation of the algorithm of Nagamochi et al. [25, 26], denoted as NOI-CGKLS, which uses a heap as its priority queue data structure (variant ni-nopr in [6]). As their implementations use signed integers as edge ids, we include their algorithms only for graphs that have fewer than 2^{31} edges. Most of our discussions focus on comparisons to the NOI-HNSS implementation as this outperforms the implementations by Chekuri et al.

Gianinazzi et al. [9] give an MPI implementation of the algorithm of Karger and Stein [18]. We performed preliminary experiments on small graphs which can be solved by NOI-HNSS, NOI-CGKLS and HO-CGKLS in less than 3 seconds. On these graphs, their implementation using 24 processes took more than 5 minutes, which matches other studies [6, 14, 16] that report bad real-world performance of (other implementations of) the algorithm of Karger and Stein. Gianinazzi et al. report a running time of 5 seconds for RMAT graphs with $n = 16000$ and an average degree of 4000, using 1536 *cores*. As NOI can find the minimum cut on RMAT graphs [20] of equal size in less than 2 seconds using *a single core*, we do not include the implementation in [9] in our experiments.

As our algorithm solves the minimum cut problem exactly, we do not include the $(2+\epsilon)$-approximation algorithm of Matula [24] and the inexact algorithm VieCut in the experiments.

Instances.: We use a set of graph instances from the experimental study of Henzinger et al. [14]. The set of instances contains k-cores [3] of large undirected real-world graphs taken from the 10th DIMACS Implementation Challenge [2] as well as the Laboratory for Web Algorithmics [4, 5]. Additionally it contains large random hyperbolic graphs [21, 33] with $n = 2^{20} \sim 2^{25}$ and $m = 2^{24} \sim 2^{32}$. A detailed description of the graph instances is given in [13]. These graphs are unweighted, however contracted graphs that are created in the course of the algorithm have edge weights.

B. Sequential Experiments.

We limit the values in the priority queue $\mathcal{Q}$ to $\hat{\lambda}$, in order to significantly lower the number of priority queue operations needed to run the contraction routine. In this experiment, we want to examine the effects of different priority queue implementations and limiting priority queue values have on sequential minimum cut computations. We also include variants which run VieCut first to lower $\hat{\lambda}$.

We start with sequential experiments using the implementation of NOI-HNSS. We use two variants: NOI$_{\hat{\lambda}}$ limits values in the priority queue to $\hat{\lambda}$ while NOI-HNSS allows arbitrarily large values in $\mathcal{Q}$. For NOI$_{\hat{\lambda}}$, we test the three priority queue implementations, BQueue, Heap and BStack. As the priority queue for NOI-HNSS has priorities of up to the maximum degree of the graph and the contracted graphs can have very large degrees, the bucket priority queues are not suitable for NOI-HNSS. Therefore we only use the implementation of NOI-HNSS [14]. The variants NOI-HNSS-VieCut and NOI$_{\hat{\lambda}}$-Heap-VieCut first run the shared-memory parallel algorithm VieCut using all 24 threads to lower $\hat{\lambda}$ before running the respective sequential algorithm. We report the total running time, e.g. the sum of VieCut and NOI.

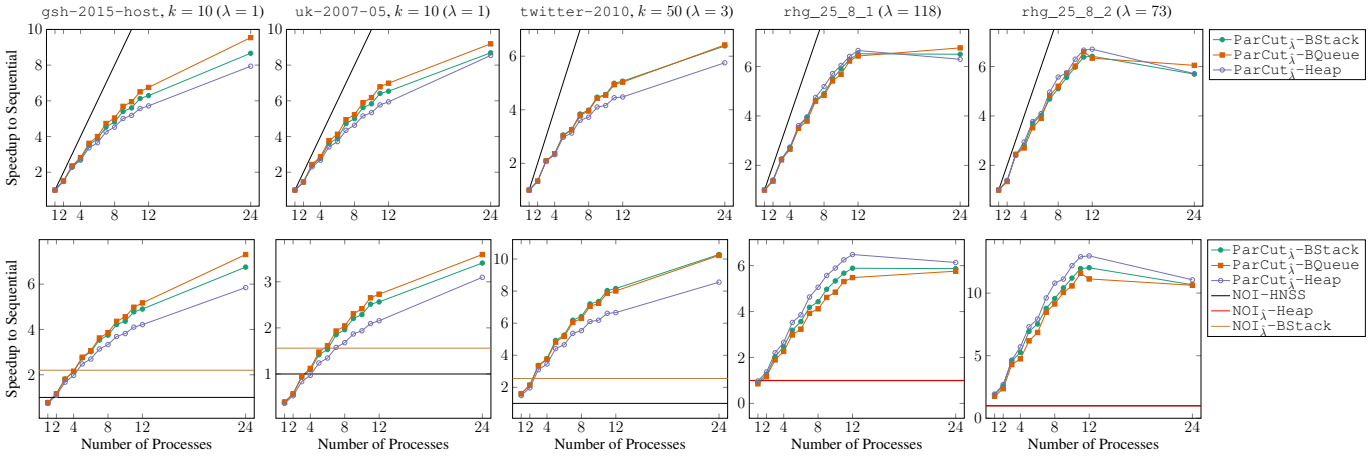

Fig. 5. Scaling plots for four large graphs. Top: Scalability. Bottom: Speedup compared to NOI-HNSS and fastest sequential algorithm (first 3 graphs: NOI$_{\hat{\lambda}}$-BStack, last 2 graphs: NOI$_{\hat{\lambda}}$-Heap).

a) Priority Queue Implementations: Figure 2 shows the results for random hyperbolic graphs and Figure 3 shows the results for real-world graphs, normalized by the running time of NOI$_{\hat{\lambda}}$-Heap-VieCut. Figure 4 gives performance plots for all graphs. We can see that in nearly all sequential runs, NOI$_{\hat{\lambda}}$-BStack is $5-10\%$ faster than NOI$_{\hat{\lambda}}$-BQueue. This can be explained as this priority queue uses std::vector instead of std::deque as its underlying data structure and thus has lower access times to add and remove elements. As all vertices are visited by the only thread, the scan order does not greatly influence how many edges are contracted.

In the random hyperbolic graphs, nearly no vertices in NOI-HNSS reach priorities in $\mathcal{Q}$ that are much larger than $\hat{\lambda}$. Usually, fewer than 5% of edges do not incur an update in $\mathcal{Q}$. Thus, NOI-HNSS and NOI$_{\hat{\lambda}}$-Heap have practically the same running time. NOI$_{\hat{\lambda}}$-BStack is usually 5% slower.

As the real-world graphs are social network and web graphs, they contain vertices with very high degrees. On these vertices, NOI-HNSS often reaches priority values of much higher than $\hat{\lambda}$ and NOI$_{\hat{\lambda}}$ can actually save priority queue operations. Thus, NOI$_{\hat{\lambda}}$-Heap is up to 1.83 times faster than NOI-HNSS with an average (geometric) speedup factor of 1.35. Also, in contrast to the random hyperbolic graphs, NOI$_{\hat{\lambda}}$-BStack is faster than NOI-HNSS on real-world graphs. Due to the low diameter of web and social graphs, the number of vertices in $\mathcal{Q}$ is very large. This favors the BStack priority queue, as it has constant access times. The average geometric speedup of NOI$_{\hat{\lambda}}$-BStack compared to NOI$_{\hat{\lambda}}$-Heap is 1.22.

b) Reduction of $\hat{\lambda}$ by VieCut: Now we reduce $\hat{\lambda}$ by VieCut before running NOI. While the other algorithms are slower for denser random hyperbolic graphs, NOI-HNSS-VieCut and NOI$_{\hat{\lambda}}$-Heap-VieCut are faster in these graphs with higher density. This happens as the variants without VieCut find fewer contractible edges and therefore need more rounds of CAPFOREST. The highest speedup compared to NOI$_{\hat{\lambda}}$-Heap is reached in random hyperbolic graphs with $n = 2^{23}$ and an average density of 2^8, where NOI$_{\hat{\lambda}}$-Heap-VieCut has a speedup of factor 4.

NOI$_{\hat{\lambda}}$-Heap-VieCut is fastest on most real-world graphs, however when the minimum degree is very close to the minimum cut λ, running VieCut can not significantly lower $\hat{\lambda}$. Thus, the extra work to run VieCut takes longer than the time saved by lowering the upper bound $\hat{\lambda}$. The average geometric speedup factor of NOI$_{\hat{\lambda}}$-Heap-VieCut on all graphs compared to the variant without VieCut is 1.34.

In the performance plots in Figure 4 we can see that NOI$_{\hat{\lambda}}$-Heap-VieCut is fastest or close to the fastest algorithm in all but the very sparse graphs, in which the algorithm of Nagamochi et al. [26] is already very fast [14] and therefore using VieCut cannot sufficiently lower $\hat{\lambda}$ and thus the running time of the algorithm. NOI-CGKLS and HO-CGKLS are outperformed on all graphs.

C. Shared-memory parallelism.

We run experiments on 5 of the largest graphs in the data sets using up to 24 threads on 12 cores. These graphs have $4M \sim 68M$ vertices and $670M \sim 4B$ edges. First, we compare the performance of Algorithm 2 using different priority queues: ParCut$_{\hat{\lambda}}$-Heap, ParCut$_{\hat{\lambda}}$-BStack and ParCut$_{\hat{\lambda}}$-BQueue all limit the priorities to $\hat{\lambda}$, the result of VieCut. In these experiments, VieCut takes up between $19-83\%$ of the total running time with an average of 51%. Thus, in average half of the total running time was used to find a good upper bound for the minimum cut using VieCut and the other half was used to compute a result guaranteed to be optimal.

Figure 5 shows the results of these scaling experiments. The top row shows how well the algorithms scale with increased number of processors. The lower row shows the speedup compared to the fastest sequential algorithm of Section IV-B. On all graphs, ParCut$_{\hat{\lambda}}$-BQueue has the highest speedup when using 24 threads. On real-world graphs, ParCut$_{\hat{\lambda}}$-BQueue also has the lowest total running time. In the large random hyperbolic graphs, in which the priority queue is usually only filled with up to 1000 elements, the worse constants of the double-ended queue cause the variant to be slightly slower than ParCut$_{\hat{\lambda}}$-Heap also even when running with 24

threads. In the two large real-world graphs that have a minimum degree of 10, the sequential algorithm NOI$_{\hat{\lambda}}$-BStack contracts most edges in a single run of CAPFOREST - due to the low minimum degree, the priority queue operations per vertex are also very low. Thus, ParCut$_{\hat{\lambda}}$ using only a single thread has a significantly higher running time, as it runs VieCut first and performs graph contraction using a concurrent hash table, as described in Section III-B, which is slower than sequential graph contraction when using just one thread. In graphs with higher minimum degree, NOI needs to perform multiple runs of CAPFOREST. By lowering $\hat{\lambda}$ using VieCut we can contract significantly more edges and achieve a speedup factor of up to 12.9 compared to the fastest sequential algorithm NOI$_{\hat{\lambda}}$-Heap. On twitter-2010, $k = 50$, ParCut$_{\hat{\lambda}}$-BQueue has a speedup of 10.3 to NOI-HNSS, 16.8 to NOI-CGKLS and a speedup of 25.5 to HO-CGKLS. The other graphs have more than 2^{31} edges and are thus too large for NOI-CGKLS and HO-CGKLS.

V. Conclusion

We presented a shared-memory parallel exact algorithm for the minimum cut problem. Our algorithm is based on the algorithms of Nagamochi et al. [25, 26] and of Henzinger et al. [14]. We use different data structures and optimizations to decrease the running time of the algorithm of Nagamochi et al. by a factor of up to 2.5. Using additional shared-memory parallelism we further increase the speedup factor to up to 12.9. Future work includes checking whether our sequential optimizations and parallel implementation can be applied to the $(2 + \epsilon)$-approximation algorithm of Matula [24].

References

[1] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 370–380. ACM, 1991.

[2] D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.

[3] V. Batagelj and M. Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[4] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[6] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, pages 324–333. SIAM, 1997.

[7] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[8] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 112–122. ACM, 1991.

[9] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler. Communication-avoiding parallel minimum cuts and connected components. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–232. ACM, 2018.

[10] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[11] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[12] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174. SIAM, 1992.

[13] M. Henzinger, A. Noe, and C. Schulz. Shared-memory exact minimum cuts. *arXiv preprint arXiv:1808.05458*, 2018.

[14] M. Henzinger, A. Noe, C. Schulz, and D. Strash. Practical minimum cut algorithms. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 48–61. SIAM, 2018.

[15] M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1938. SIAM, 2017.

[16] M. Jünger, G. Rinaldi, and S. Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000.

[17] D. R. Karger. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM Review*, 43(3):499–522, 2001.

[18] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

[19] K.-i. Kawarabayashi and M. Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing*, pages 665–674. ACM, 2015.

[20] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[21] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguná. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3):036106, 2010.

[22] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.

[23] T. Maier, P. Sanders, and R. Dementiev. Concurrent hash tables: Fast and general?(!). In *ACM SIGPLAN Notices*, volume 51, page 34. ACM, 2016.

[24] D. W. Matula. A linear time $2 + \varepsilon$ approximation algorithm for edge connectivity. In *Proceedings of the 4th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 500–504. SIAM, 1993.

[25] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[26] H. Nagamochi, T. Ono, and T. Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67(1):325–341, 1994.

[27] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47(1):19–36, 1990.

[28] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[29] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[30] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[31] A. Ramanathan and C. J. Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39(3):253–261, 1987.

[32] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

[33] M. von Looz, H. Meyerhenke, and R. Prutkin. Generating random hyperbolic graphs in subquadratic time. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC 2015)*, volume 9472 of *LNCS*, pages 467–478. Springer, 2015.

[34] I. Wegener. Bottom-up-heapsort, a new variant of heapsort beating, on an average, quicksort (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.

[35] J. W. J. Williams. Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.