Parallel Randomized Minimum Cuts and Parallel Tree Contractions

Yen-Hsiang Chang

Abstract

Finding minimum cuts is a classic graph theory problem that also has a variety of applications in graph analysis. On graphs with billions of edges, solving the minimum cut problem in a deterministic manner is computationally intensive. On the contrary, randomized minimum cut algorithms tend to have better asymptotic time complexity. Therefore, in this paper, we implement a portion of a recent parallel randomized minimum cut algorithm in shared memory, which relies on parallel tree contractions and parallel batched updates and queries. Experiments show that our parallel implementation is scalable but not competitive against the state-of-the-art deterministic parallel solver due to the inherent huge constants in the parallel data structures used.

1 Introduction

Given an undirected graph with n vertices and m edges, a minimum cut is a partition of the vertices into two subsets such that the total weights of edges crossing two subsets is minimized. Finding a minimum cut is a classic problem in graph theory, and it has a wide range of applications including approximating traveling salesman problems [CABC11], network reliability analysis [RC87] and cluster analysis [SS00]. Computing a minimum cut in a deterministic manner, however, is either expensive [GH61, NOI94, HO94, SW97, HNS19] or not practical [HLRW24]. Therefore, randomized approaches [Kar93, Kar00] were proposed in the past. Recent improvements have brought down sequential randomized minimum cut to $O(m \log^2 n)$ work [GMW19] and a parallel algorithm with the same work and in $O(\log^3 n)$ span [AB23].

Since [AB23] is a theory paper and no implementation is provided, our goal is to implement the first half of its parallel algorithm in shared memory, which performs parallel randomized edge contractions utilizing parallel tree contractions and parallel batched updates and queries. Our evaluation shows that although their parallel algorithm is scalable, it has inherent constant issues that make it way slower than a naive sequential implementation of randomized edge contractions. We also observe that this parallel randomized minimum cut algorithm is not competitive against the state-of-the-art deterministic parallel solver [HNS19], which benefits from heuristics that work well on real-world sparse datasets.

2 Background

Let $G = (V, E, w)$ be an undirected weighted graph where $|V| = n, |E| = m$ and $w : E \to \mathbb{R}^+$. A cut $C = (S, V \setminus S)$ is a partition of V into nonempty sets S and $V \setminus S$. The value of a cut is the total weights of edges crossing the cut. The goal is to find a cut that has minimum value c. See Figure 1 for an example.

In the rest of this section, we introduce the cornerstone, randomized edge contractions [Kar93], that is used in randomized minimum cut algorithms. We then describe its parallelization and elaborate on the tools used, including parallel tree contractions [MR85] and parallel batched updates and queries [AB23].

2.1 Randomized Edge Contractions

The contraction of an edge $e = \{u, v\}$ of G is defined as follows: merge u and v into a single vertex after deleting e, and merge the resulting multiple edges with the same pair of end vertices into a single edge whose weight is equal to the sum of weights of such edges. The following lemma is given in [Kar93]:

Lemma. *Consider a graph with n vertices and a minimum cut. By choosing an edge at random with probability proportional to its weight, the probability the edge is included in this minimum cut is at most $\frac{2}{n}$.*

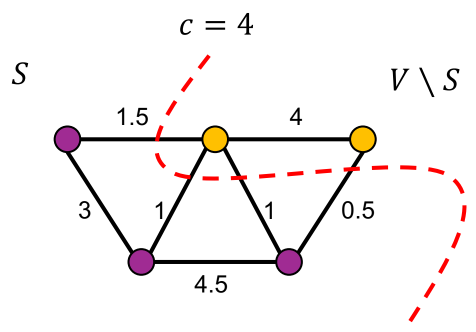


Figure 1: An example of minimum cut.

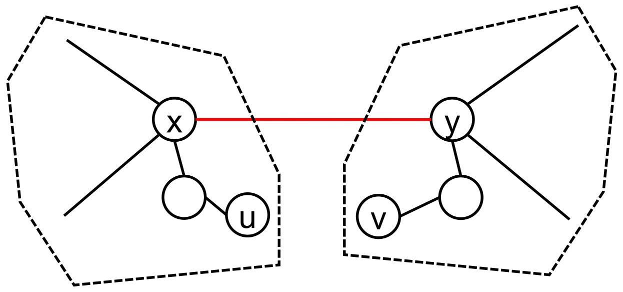


Figure 2: Black edges are the contracted edges in the MST and dotted regions are the connected components induced by these edges. The red edge $\{x, y\}$ contributes to the incident weight of the components of x and y if the edge $\{u, v\}$ having the largest index on the MST path from x to y is not contracted yet.

As contracting an edge that is not in the minimum cut does not affect the minimum cut after contraction, by repeatedly choosing an edge at random and contract the edge until two vertices remain, the following theorem is then given in [Kar93]:

Theorem. *A particular minimum cut can be produced via random edge contractions with probability* $\Omega(n^{-2})$.

Observe that contracting edges until only two vertices remain requires $O(m)$ time. If we also keep track of the minimum degree (without further specified, we use degree to denote the total incident weight of a vertex) of the resulting graphs during random edge contractions, two corollaries follow [Kar93]:

Corollary. *A minimum cut can be found in* $O(mn^2 \log n)$ *time with high probability.*

Corollary. *A* k*-approximation of the minimum cut can be found in* $O(mn^{2/k} \log n)$ *time with high probability.*

2.2 Parallel Randomized Minimum Cuts

Random edge contractions can be parallelized by finding the minimum spanning tree (MST) based on the order given in a weighted random edge permutation [Kar93]. A weighted random edge permutation is a permutation such that for two edges e_1 and e_2, the probability that e_1 has a smaller index than e_2 in the permutation is $\frac{w(e_1)}{w(e_1)+w(e_2)}$. This can be generated by drawing a sample for each edge e from the exponential distribution $Exp(w(e))$ and sort the samples drawn from each edge. The MST then naturally follows from the total order given in the permutation and edge contractions become contracting edges in the MST following the order in the permutation.

Maintaining the minimum degree of the resulting graphs during random edge contractions in a parallel fashion, however, is more complicated. Essentially, this is equivalent to starting with the vertices in the original graph G, joining vertices using the contracted edges, and keeping track of the total incident weight of each component induced by these edges. More formally, the following steps solve the same problem:

1. **Construct** a vertex-weighted tree from the MST, where weights are the degree of each vertex in the original graph G.
2. For each edge $e = \{u, v\}$ in the MST in the order given in the weighted random edge permutation:
   (a) **Determine** the set of edges in G where an edge $f = \{x, y\}$ in the set satisfies that $\{u, v\}$ has the largest index in the permutation among the edges on the MST path from x to y.
   (b) For each such edge identified, **Subtract** weights from x and y by $w(\{x, y\})$.
   (c) **Join** edge $\{u, v\}$.
   (d) **Query** the total incident weight of the component u is in induced by the joined edges.

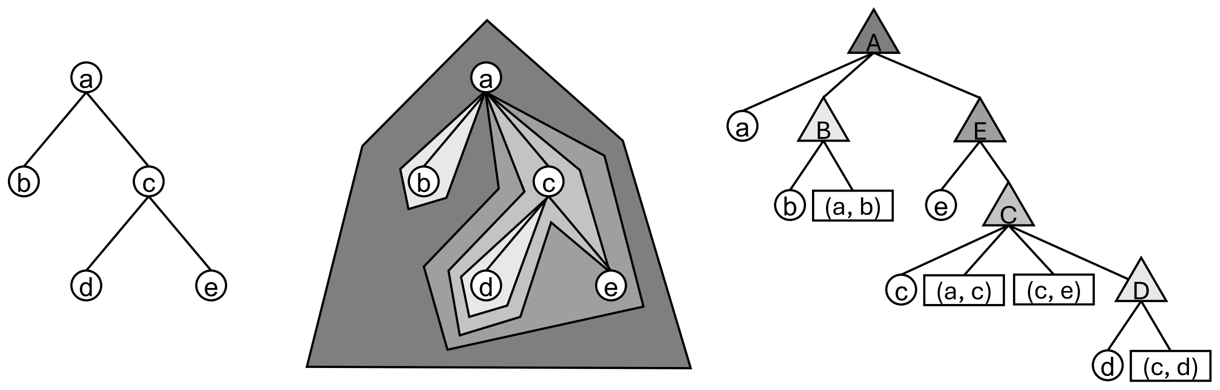


Figure 3: An example of building rake-and-compress tree where the input tree is provided on the left. The middle part demonstrates a recursive clustering using rake and compress operations, where clusters produced in earlier rounds are depicted using lighter colors. Only the vertex c is compressed in this example. The right hand side illustrates the resulting rake-and-compress tree, where vertices and edges from the original tree are located at the leaf clusters, and clusters produced are labeled with the uppercase letter of their corresponding vertices involved in the rake and compress operations.

As shown in Figure 2, the main idea is that an edge contributes to some vertex weights if its two endpoints are not in the same component, and this happens before all edges on its MST path are contracted.

In [AB23], the following theorem and corollary are given by using parallel tree contractions and parallel batched updates and queries to parallelize these operations, where an extra $\log n$ factor is involved compared to the sequential version:

Theorem. *A k-approximation of the minimum cut can be found in $O(mn^{2/k} \log^2 n)$ work and $O(\log^2 n)$ span with high probability.*

Corollary. *A $\log n$-approximation of the minimum cut can be found in $O(m \log^2 n)$ work and $O(\log^2 n)$ span with high probability.*

Since finding a minimum cut purely using random edge contractions is expensive, the first half of the parallel algorithm provided in [AB23] is to find a $\log n$-approximation of the minimum cut using $O(\log n)$ rounds of random edge contractions, and this is the part we focus on in this paper.

2.2.1 Parallel Tree Contractions

Parallel tree contraction is a technique to apply various operations over trees and maintain values in logarithmic parallel depth [MR85]. Tree contractions consist of a set of rake and compress operations. Given a bounded-degree rooted tree T of n vertices, a rake-and-compress tree is a recursive clustering of T by doing the following two types of operations, with an example in Figure 3:

1. **Rake**: A rake operation merges a leaf vertex into its parent.
2. **Compress**: A compress operation removes a vertex of degree two and replaces its two incident edges with an edge joining its two neighbors.

In [MR85], they observe that rakes and compresses can be applied in parallel if they are applied to an independent set of vertices, and the random-mate algorithm is introduced to achieve $O(n)$ work and $O(\log n)$ span with high probability. In each round of clustering, for vertices that are ready to be raked or compressed, a random bit is assigned to each of them. After that, a vertex is raked or compressed if its bit does not conflict with its parent: it has a bit 1 and its parent is either not ready or having a bit 0. It is then proven that with high probability at most $\frac{4}{5}$ of the vertices remain after a single round, hence achieving $O(\log n)$ span and the rake-and-compress tree of T has $O(\log n)$ depth.

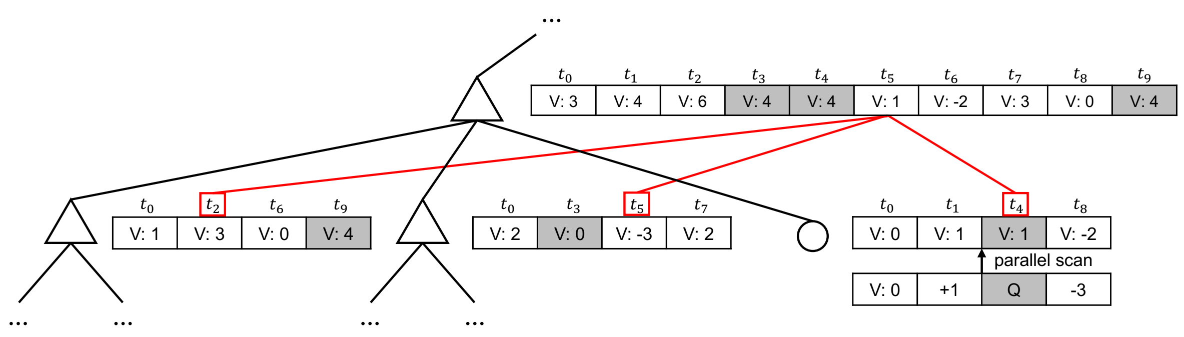


Figure 4: Parallel updates and queries where each cluster maintains the sum of weights from the leaf clusters in its subtree. Queries are covered by a darker color. After sorting updates and queries by leaf clusters and timestamps, parallel scans are performed to get the values for leaf clusters at different timestamps. After merging updates and queries from the child clusters, the value at each timestamp in the current cluster is then determined by the corresponding latest child values.

Maintaining values in clusters is a local operation that gathers information from its child clusters, which does not affect the random-mate algorithm. The only crucial thing is that it is a constant-time operation from the assumption that T has bounded degrees, so it does not affect the overall time complexity. A single update or query on the original tree T can be transformed into walking up from a leaf cluster to the root cluster in the rake-and-compress tree, and maintaining values or gathering contributions from the clusters visited. Since the depth of the rake-and-compress tree is $O(\log n)$ and there is a constant-time operation needed for each visited cluster, a single update and query can be done in $O(\log n)$ time. As these operations are unimportant for parallelization, see [AB23] for more details on how they are designed to support updates and queries for the operations needed in Section 2.2.

2.2.2 Parallel Batched Updates and Queries

Although each update or query can be performed in $O(\log n)$ time with the help of rake-and-compress tree, parallelizing these operations is not trivial. The reason is that there is little benefit that can be gained from parallelizing a single operation since each operation is tiny. To solve this issue, the idea of parallel batched mixed operations is introduced in [AB23], where all updates and queries are collected into a single batch, and these operations are parallelized altogether by resolving the dependency imposed by the time order. This method relies on the assumptions that updates and queries can be performed by walking up from a leaf cluster to the root cluster, and there is only a constant-time operation needed for each visited cluster. Additionally, they assume that updates only use associative operations to modify values in the leaf clusters of the rake-and-compress tree, followed by re-evaluation from the leaf cluster to the root cluster. With these assumptions, the parallelization scheme is then given using the following steps with an example shown in Figure 4:

1. Collect all updates and queries as a batch with their associated leaf clusters.
2. Timestamp the updates and queries by their time order.
3. Sort the updates and queries within each leaf cluster by their timestamps.
4. For each leaf cluster, use a parallel scan on the update values to calculate the value of the leaf cluster after each operation, starting from the initial value on the leaf cluster.
5. Initialize each query using the value it received from the parallel scan.
6. For each level of the rake-and-compress tree starting one above the deepest, and in parallel for every cluster:
   (a) Merge the updates and queries from each child cluster into a single list sorted by timestamp.

(b) Calculate for each element in the merged list, the latest value of each child cluster at or before the timestamp.

(c) For each list element, calculate the value at that timestamp from the values collected in the previous step.

(d) For each query, use the current values and child values to collect contributions.

It is then shown in [AB23] that given a batch of k updates and queries, the steps mentioned above can be implemented in parallel in $O(k \log n)$ work and $O(\log n \log k)$ span. Since there are $O(m)$ operations in total for randomized edge contractions, the parallel batched updates and queries require $O(m \log n)$ work and $O(\log^2 n)$ span, which has an extra $\log n$ factor comparing to its sequential counterpart. In conclusion, using parallel tree contractions and parallel batched updates and queries, the complexity given in 2.2 can be achieved.

3 Methodology

Although a parallel algorithm for randomized edge contractions is given in Section 2, it only contains theoretical analysis and there are still some gaps that need to be filled for practical implementations. In this section, we will describe how we implement this parallel algorithm using ParlayLib [BAD20] as the backbone, where we use their parallel primitives including merge, scan, sort, and their work-stealing scheduler. We also use GBBS [DBS21] to process graphs and utilize their parallel MST implementation, where graphs are represented using CSR formats.

3.1 Preprocessing

3.1.1 Weighted Random Edge Permutation and Minimum Spanning Tree

The first step is to generate a weighted random edge permutation from the input graph, where we use C++ built-in `std::exponential_distribution` to draw a sample for each edge based on their weights. Originally, a parallel sort follows in order to get the permutation. However, this is not needed since the downstream task is finding the MST based on this order, which does not assume a sorted edge list from the input. Still, we need to sort the outputs from the MST to get the time order of the operations.

3.1.2 Converting MST into a Ternary Tree

Before building rake-and-compress tree from the MST, we need to convert the MST into a bounded-degree rooted tree, which is needed for the assumption in the tree contraction algorithm. Figure 5 presents the primitive for this procedure: for each vertex with degree d larger than 3, we break it into $d-2$ vertices connected as a chain; for the original incident edges of this vertex, the first two are connected to the first vertex of the chain, the last two are connected to the last vertex of the chain, and the other edges are assigned to the internal vertex of the chain sequentially. Using this strategy, each vertex of the chain has degree three. Note that new vertices and edges are appended to the end of the original vertex list and edge list to avoid recomputing indices, and their weights are identity elements determined by the associative operations needed in Section 2.2. We then create a new CSR representation for the resulting ternary tree.

3.1.3 Converting Ternary Tree into a Rooted Binary Tree

After getting the ternary tree, we need to convert it into a rooted binary tree, where we choose a leaf as the root and determine the orientation of each edge. To achieve this in parallel, we use the Euler tour technique where an Euler tour is created from the CSR format as shown in Figure 6: for each edge (u, v) in the CSR format, we find its inverse edge (v, u) and find the next incident edge f of v following (v, u) (cyclic-shift if needed); we then create a parent pointer from f to (u, v), indicating that the Euler tour visits f right after visiting (u, v). Since these parent pointers form a cycle, we can break this cycle at the chosen root and the cycle becomes a linked list. Determining the orientation of each edge then becomes solving the list ranking problem, where each element (edge) finds its rank in the linked list since this corresponds to the timestamp of the Euler tour.

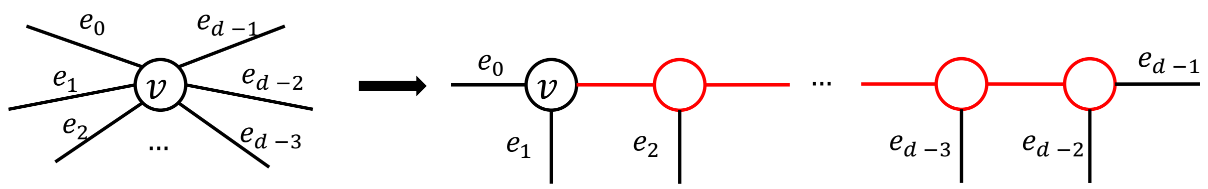
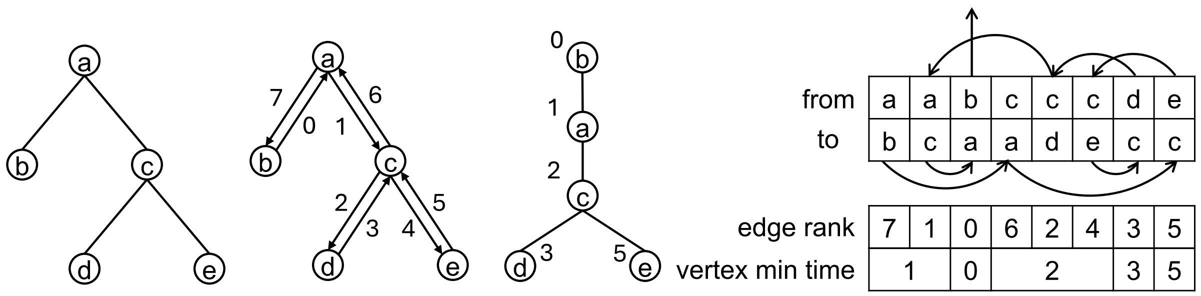


Figure 5: Converting a vertex of degree d ($d > 3$) into a chain of vertices where each vertex has degree 3. Red vertices and edges are the new ones that are appended to the original vertex list and edge list, and their weights are identity elements determined by the associative operations.



from	a	a	b	c	c	c	d	e
to	b	c	a	a	d	e	c	c
edge rank	7	1	0	6	2	4	3	5
vertex min time	1		0	2			3	5

Figure 6: Given a ternary tree on the left, to root the tree at vertex b, we can create a Euler tour starting from b and keep track of the first time each vertex is visited. The orientation of edges can therefore be decided by comparing the timestamps of two endpoints. This can be done in parallel illustrated on the right by creating a linked list via cyclic-shifting the CSR format, and use parallel list ranking to solve the problem.

Although parallel list ranking with n elements can be done in $O(n)$ work and $O(\log n)$ span [CV89], this is not the bottleneck of the algorithm hence we only implement an $O(n \log n)$ work and $O(\log n)$ span version using pointer jumping. To wrap up this conversion, we create a new CSR representation for the resulting rooted binary tree.

3.2 Parallel Tree Contractions

Note that there are only $O(n)$ vertices in the MST, but there will be $O(m)$ vertices after converting the MST into a rooted binary tree. As building the rake-and-compress tree from the rooted binary tree is not the bottleneck, we only write an implementation of $O(m \log n)$ work and $O(\log n)$ span with high probability instead of $O(m)$ work. The strategy is straightforward, where we implement the random-mate algorithm described in Section 2.2.1 by checking every vertex in each round. As each vertex is checked and there are $O(\log n)$ rounds, we achieve $O(m \log n)$ work. One thing noteworthy is that in the random-mate algorithm, we still require a two-phase approach to first identify independent vertices then perform rakes or compresses, and use atomic operations to maintain degrees. Again, since the tree has bounded degrees, this only imposes a constant in the complexity. Also, a more sophisticated task management is required to achieve $O(m)$ work [MR85] by not checking every vertex in each round. Simply using a queue to keep track of vertices ready to be raked or compressed does not work since it creates contention on the queue.

Although the random-mate algorithm is intuitive, the most effort for building the rake-and-compress tree comes from maintaining values in each cluster. We not only need to follow the design in Section 2.2 to maintain values needed in randomized edge contractions, we also need to keep track of properties of clusters such as boundaries and distances to the root cluster since these are required when maintaining values. However, we will not go over these details as they are not related to parallelization.

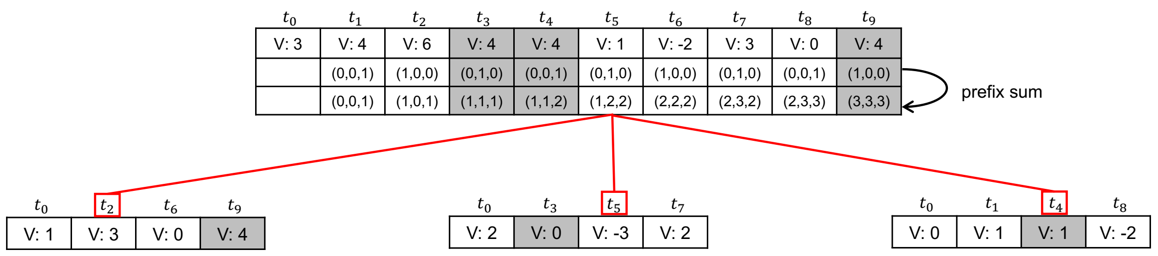


Figure 7: Identifying latest values from child clusters using prefix sum, where we use the same example from Figure 4. For each timestamp in the current cluster, except the initial value, we first identify which child cluster it comes from during parallel merges, and set the value to one in the corresponding entry. A prefix sum follows and each entry becomes the index of the latest timestamp from the corresponding child cluster. The value for the current timestamp is then maintained by collecting information from child clusters.

3.3 Parallel Batched Updates and Queries

Once the rake-and-compress tree is built and the updates and queries are collected as a batch, we perform parallel batched updates and queries described in Section 2.2. To sort the updates and queries within each leaf cluster by their timestamps, we can use a single parallel sort by first comparing the leaf clusters (assuming there is an order embedded in it, e.g., pre-order traversal) and breaking ties by comparing the timestamps. Then, parallel scans in the leaf clusters can also be done using a single parallel scan by forming segments from the sorted operations divided by the leaf clusters, and perform a segmented scan [Ble89].

After values are initialized in the leaf clusters, we iterate the levels from one above the deepest to the root level. For each cluster in the current level, we identify the values at different timestamps in its child clusters, and perform parallel merges to get a list of timestamps for the current cluster. Since the degree is bounded in the original tree, there is only a constant number of child clusters in the rake-and-compress tree, hence the complexity only depends on the length of the resulting list. Then, for each timestamp in the list of the current cluster, we need to maintain its value by considering the latest values before this timestamp in its child clusters, which can be found using a parallel scan. Suppose there are k child clusters for the current cluster. For each timestamp in the current list except the initial value, we create a k-tuple with all zeros except a one in the i'th entry if this timestamp comes from the i'th child cluster. After performing prefix sum on these tuples, the value p in the i'th entry denotes that the p'th value in the i'th cluster is the latest value in the i'th cluster right before the current timestamp. An example is shown in Figure 7. The queries then collect contributions along the way once the values are populated. After all levels are visited, all queries also contain the correct answers as they have gathered information on the path from the leaf clusters to the root cluster.

4 Experiments

4.1 Baselines and Implementations

Our primary baseline for finding minimum cuts is the VieCut [HNS19]. VieCut is a parallel deterministic minimum cut solver which parallelizes the $O(mn + n^2 \log n)$ algorithm from [NOI94]. It works well in practice due to its heuristics benefiting from real-world large sparse graphs. The reason why we choose a deterministic solver as our baseline is that there is no competitive parallel randomized solver in shared memory, and comparing to the state-of-the-art solver gives us an idea on whether it is worth completing the second half of the randomized algorithm we are implementing.

Our implementations include a parallel randomized edge contractions solver (ParREC) described in Section 3, a sequential randomized edge contractions solver using union-find [Tar75] (SeqREC-UF), and a sequential randomized edge contractions solver using rake-and-compress tree without parallel batched updates and queries (SeqREC-RC). We implement SeqREC-UF in order to check the correctness of our parallel implementations for rake-and-compress tree and batched updates and queries. Also, we only use path compression but not union by rank in the union-find data structure to achieve $O(m \log n)$ complexity that is

Table 1: Statistics for the lines of code in different categories.

Code categories	Lines of code
I/O + Ordering + MST	88
Cluster information in the rake-and-compress tree	393
Maintaining values in the rake-and-compress tree	280
Tree Conversions	238
Tree Contractions	182
Collecting updates and queries	12
ParREC	178
SeqREC-UF	28
SeqREC-RC	16

the same as the complexity of the single-threaded ParREC, so that we can have a fair comparison between the complicated ParREC implementation and the naive SeqREC-UF implementation. Lastly, SeqREC-RC decouples the overhead added by maintaining values in the rake-and-compress tree and the overhead added by merges and scans in batched updates and queries, enabling us to pinpoint the bottlenecks in our implementations. Without further specified, we only conduct one round of random edge contractions instead of $O(\log n)$ rounds required for the $\log n$-approximation of the minimum cut.

Table 1 shows the statistics for the lines of code used in different categories of our implementations. We can observe that half of the code is to maintain values and cluster information in the rake-and-compress tree, and the other half is to implement the core design mentioned in Section 3.

4.2 Environment and Datasets

Experiments are conducted on a CPU node of NERSC Perlmutter. It has 128 physical cores divided between two AMD EPYC 7763 (Milan) CPUs, where we use up to 64 threads in our experiments without hyper-threading. Our implementations are in C++ with OpenMP shared-memory parallelism. We use ParlayLib [BAD20] for their parallel primitives including merge, scan, sort, and their work-stealing scheduler. We also use GBBS [DBS21] to process graphs and utilize their parallel MST implementation, where graphs are represented using CSR formats.

Datasets are generated using the random hyperbolic generator from NetworKit [SSM14], which replicates features of real-world networks. We set the exponent of power-law degree distribution to 5, set the average degree to 64 or 256, and generate random hyperbolic graphs with 2^{16} or 2^{20} vertices. This generates four input graphs with $(n, m) = (65K, 2.1M), (65K, 8.4M), (1M, 34M)$ and $(1M, 134M)$, respectively. The weights are integers drawn uniformly from $[1, \log n)$.

4.3 Strong Scaling and Runtime Breakdown

Figure 8 shows the scalability experiments by comparing runtime breakdown for one round of randomized edge contractions using ParREC on the four input graphs. One observation is that the breakdown and scalability are similar across different input graphs, where ParREC only scales well when the number of threads is small. The reason ParREC does not scale well for large number of threads is that there is still load imbalance when processing updates and queries in deep levels of the rake-and-compress tree. As shown in Figure 9, the average load imbalance, defined as maximum over mean number of updates and queries processed across threads, is high in deep levels of the rake-and-compress tree. This is due to the fact that operations lists are small when they are close to leaf clusters, and parallelizing them across threads introduce lots of overheads. A more sophisticated scheduler or a redesign of the algorithm in deep level is needed in order to alleviate this issue.

4.4 Comparison across ParREC, SeqREC-UF and SeqREC-RC

Another observation from Figure 8 is that ParREC spends most of the time on processing updates and queries in parallel. Therefore, we also compare this part with SeqREC-UF and SeqREC-RC in order to

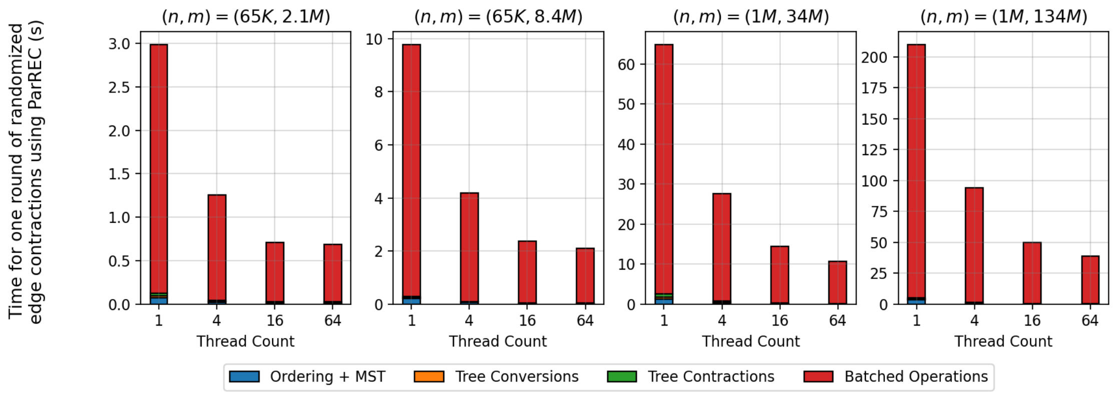


Figure 8: Runtime breakdown for one round of ParREC vs. thread count on 4 input graphs.

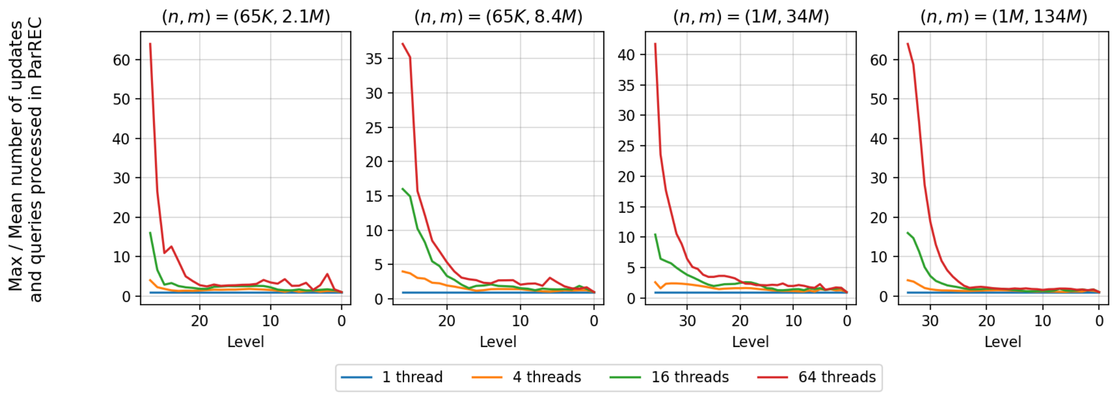


Figure 9: Average load imbalance, defined as maximum over mean number of updates and queries processed across threads, for ParREC at each level in the rake-and-compress tree.

identify the bottleneck. Figure 10 shows the runtime comparison when performing one round of randomized edge contractions. SeqREC-UF performs the best since it does not need to interact with the rake-and-compress tree; SeqREC-RC follows as maintaining values in the rake-and-compress tree adds overheads; ParREC performs the worst as parallel batched updates and queries add another layer of overheads on top of maintaining values, and it only catches up with SeqREC-RC when 64 threads are used. However, note that all the algorithms here have the same asymptotic time complexity $O(m \log n)$, and the overheads are merely coming from constant issues. On one hand, maintaining values in the rake-and-compress tree has a constant of 3 coming from the number of levels since the logarithmic base is $5/4$ instead of 2 as mentioned in Section 2.2.1, and another constant of 4 as there are at most 4 child clusters for any cluster. On the other hand, parallel batched updates and queries also suffer from the constant issue imposed by the number of child clusters, and this is aggregated by merging operation lists.

4.5 Comparison between ParREC and VieCut

Finally, we compare the performance between ParREC and VieCut to see if the parallel randomized algorithm we are implementing has the potential to beat the deterministic parallel solver. Figure 11 shows the performance comparison between VieCut and one round of ParREC. The only potential of ParREC is that it is more scalable than VieCut. Note that at the end we need to conduct $O(\log n)$ rounds of ParREC to first get a $\log n$-approximation of the minimum cut, but a single round of ParREC is already not competitive against VieCut. Therefore, before completing the second half of the parallel randomized minimum cut solver we are implementing, one needs to resolve the constant issues mentioned in Section 4.4, and optimize

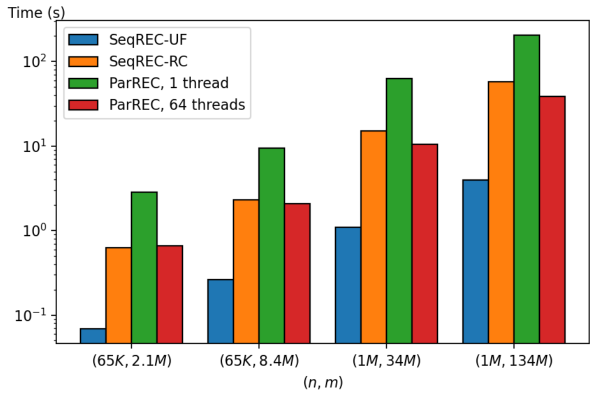


Figure 10: Runtime comparison across SeqREC-UF, SeqREC-RC, ParREC using 1 thread and ParREC using 64 threads when performing one round of randomized edge contractions.

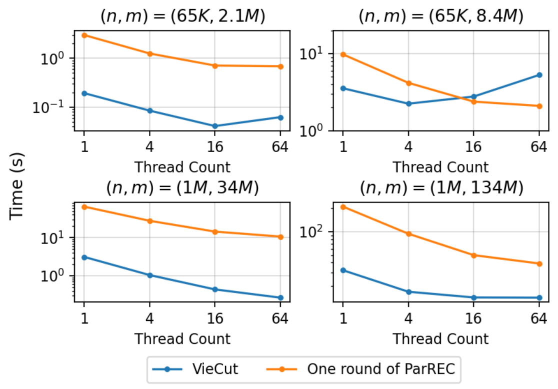


Figure 11: Runtime comparison between VieCut and one round of ParREC with different number of threads in four input graphs.

the performance more until $O(\log n)$ rounds of ParREC is faster than VieCut. Another direction is to find other interesting instances that have poor performance using VieCut, and demonstrate the prominence of randomized minimum cut solvers.

5 Conclusions and Future Work

We have demonstrated in this work that parallel randomized edge contractions using parallel tree contractions and parallel batched updates and queries are scalable in shared memory, but the inherent constant issues make the method not competitive against a naive serial implementation. This also leads to the question that whether parallel randomized minimum cut algorithm is attractive, as it is outperformed by the state-of-the-art deterministic parallel solver, which works well on real-world sparse datasets. Future work includes overcoming the huge constants by either optimizing the code or finding new parallelization paradigms to replace tree contractions and batched updates and queries. Completing the second half of the parallel randomized minimum cut solver is another direction that can make this work self-contained. Also, evaluating the robustness of the state-of-the-art deterministic parallel solver by finding inputs having poor performance can demonstrate the necessity of parallel randomized minimum cut solvers.

References

[AB23] Daniel Anderson and Guy E Blelloch. Parallel minimum cuts in $O(m \log^2 n)$ work and low depth. *ACM Transactions on Parallel Computing*, 10(4):1–28, 2023.

[BAD20] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib-A toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 507–509, 2020.

[Ble89] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.

[CABC11] William J Cook, David L Applegate, Robert E Bixby, and Vasek Chvátal. *The traveling salesman problem: A computational study.* Princeton university press, 2011.

[CV89] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and computation*, 81(3):334–352, 1989.

[DBS21] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[GH61] 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.

[GMW19] P. Gawrychowski, S. Mozes, and O. Weimann. Minimum cut in $O(m \log^2 n)$ time. *arXiv preprint arXiv:1911.01145*, 2019.

[HLRW24] Monika Henzinger, Jason Li, Satish Rao, and Di Wang. Deterministic near-linear time minimum cut in weighted graphs. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3089–3139. SIAM, 2024.

[HNS19] M. Henzinger, A. Noe, and C. Schulz. Shared-memory exact minimum cuts. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 13–22. IEEE, 2019.

[HO94] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994.

[Kar93] D. R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, page 21–30, USA, 1993. Society for Industrial and Applied Mathematics.

[Kar00] D. R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.

[MR85] Gary L Miller and John H Reif. Parallel tree contraction and its application. In *FOCS*, volume 26, pages 478–489, 1985.

[NOI94] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67:325–341, 1994.

[RC87] Aparna Ramanathan and Charles J Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39:253–261, 1987.

[SS00] Roded Sharan and Ron Shamir. Click: A clustering algorithm with applications to gene expression analysis. In *Proc Int Conf Intell Syst Mol Biol*, volume 8, page 16. Maryland, MD, 2000.

[SSM14] Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *arXiv preprint arXiv:1403.3005*, 2014.

[SW97] M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.