Communication-Avoiding Parallel Minimum Cuts and Connected Components

Lukas Gianinazzi
ETH Zurich
Department of Computer Science
glukas@student.ethz.ch

Pavel Kalvoda[*]
ETH Zurich
Department of Computer Science
kalvodap@student.ethz.ch

Alessandro De Palma
ETH Zurich
Department of Computer Science
depalmaa@student.ethz.ch

Maciej Besta
ETH Zurich
Department of Computer Science
maciej.besta@inf.ethz.ch

Torsten Hoefler
ETH Zurich
Department of Computer Science
htor@inf.ethz.ch

Abstract

We present novel scalable parallel algorithms for finding global minimum cuts and connected components, which are important and fundamental problems in graph processing. To take advantage of future massively parallel architectures, our algorithms are *communication-avoiding*: they reduce the costs of communication across the network and the cache hierarchy. The fundamental technique underlying our work is the *randomized sparsification* of a graph: removing a fraction of graph edges, deriving a solution for such a sparsified graph, and using the result to obtain a solution for the original input. We design and implement sparsification with $O(1)$ synchronization steps. Our global minimum cut algorithm decreases communication costs and computation compared to the state-of-the-art, while our connected components algorithm incurs few cache misses and synchronization steps. We validate our approach by evaluating MPI implementations of the algorithms on a petascale supercomputer. We also provide an approximate variant of the minimum cut algorithm and show that it approximates the exact solutions well while using a fraction of cores in a fraction of time.

CCS Concepts • **Theory of computation → Distributed algorithms**;

Keywords Parallel Computing, Minimum cuts, Randomized Algorithms, Graph Algorithms

ACM Reference Format:
Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler. 2018. Communication-Avoiding Parallel Minimum Cuts and Connected Components. In *PPoPP '18: PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3178487.3178504

1 Introduction

Graph computations are behind many problems in machine learning, social network analysis, and computational sciences [28]. An important and fundamental class are graph connectivity algorithms, such as finding minimum cuts or connected components.

The *global* minimum cut problem is a classic problem in graph theory; it finds a variety of applications in network reliability studies [23], combinatorial optimization [25], matrix diagonalization, memory paging, gene-expression analyses [39], and large-scale graph clustering [40]. *Connected components* is a well-studied problem with a plethora of applications, for instance in medical imaging [46], image processing [21, 32], and computer vision [49].

Designing efficient parallel graph algorithms is challenging due to their properties such as irregular and data-driven communication patterns or limited locality. These properties result in movements of large amounts of data on shared-memory (e.g., cache misses) and distributed-memory systems (e.g., network communication), having a negative impact on performance [20, 31, 33]. Moreover, synchronization barriers that enforce data dependencies are costly and have to be used with caution [7].

Communication-avoiding algorithms, which require asymptotically less communication than their alternatives [20], alleviate these issues. Developing and analyzing such schemes requires models that explicitly incorporate the cost of communication. One example is the Cache-Oblivious (CO) model [11] which enables designing algorithms which reduce cache

[*] Pavel Kalvoda was a student at ETH Zurich at the time of his involvement, but is now employed by Google Inc.

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02...$15.00
https://doi.org/10.1145/3178487.3178504

	Supersteps	Computation	Communication Volume	Cache Misses	Space
Previous BSP△ [4]	$O(\log n \log^2 p)$	$\Theta\left(\frac{n^2 \log^3 n \log p}{p}\right)$	$\Theta\left(\frac{n^2 (\log^2 n) \log^2 p}{p}\right)$	–	$O\left(\frac{n^2 \log^2 n}{p}\right)$
This paper ⋆	$O\left(\log\left(\frac{pm}{n^2}\right)\right)$	$O\left(\frac{n^2 \log^3 n}{p}\right)$	$O\left(\frac{n^2 (\log^2 n) \log p}{p}\right)$	$O\left(\frac{n^2 \log^3 n}{Bp}\right)$	$O\left(\min(m, \frac{n^2 \log^2 n}{p})\right)$
CO Karger-Stein [13]	–	$O(n^2 \log^3 n)$	–	$O\left(\frac{n^2 \log^3 n}{B}\right)$	$O(n^2)$

Table 1. Bounds for Computing a Minimum Cut. All algorithms are randomized and return correct results with high probability. Cache misses of the first entry have not been studied. CO Karger-Stein is a sequential algorithm. (⋆) assuming $pB \leq n^{1-\epsilon}$, (△) assuming $p \leq n$.

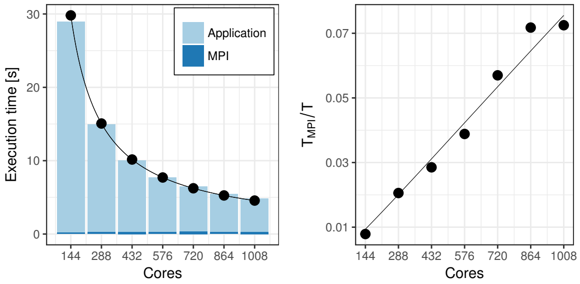
misses without the knowledge of the memory hierarchy. Another example is the Bulk Synchronous Parallel (BSP) model [47] which facilitates explicit reasoning about network communication and synchronization costs. In this work, we combine these two models for a more detailed analysis of the communication costs of algorithms for finding global minimum cuts and connected components.

Our randomized algorithms are based on a technique called *Iterated Sampling*. The idea is to iteratively *sparsify* the graph, i.e., derive a sparse random sample of the graph that preserves the vertices but removes selected edges. This technique was used to find minimum cuts in parallel [25]; we show that it also gives a communication-avoiding and practical connected components algorithm. We perform Iterated Sampling using $O(1)$ synchronization steps with high probability, ensuring $O(1)$ such steps for finding connected components and approximate minimum cuts. Moreover, it allows us to compute minimum cuts that are exact with high probability using $O(\log p)$ synchronisation steps, where p is the number of processors.

Specifically, we improve upon a previous BSP approach [4] to the global minimum cut problem in terms of both communication and computation costs, as shown in Table 1. We also obtain a number of cache misses matching that of a recent cache-oblivious sequential variant of the same algorithm [13]. Our experiments indicate that our algorithm indeed spends little time communicating (Figure 1b), behaves according to our model predictions (Figure 1a), and outperforms sequential codes with only a few processors.

We also propose an approximate variant of the minimum cut algorithm that has near-linear work and gives an $O(\log n)$-approximate minimum cut.

Finally, our sparsification technique gives a communication-avoiding connected components algorithm with theoretical bounds close to the state-of-the-art [2] when the average degree is larger than the number of processors. When this is the case, our connected components algorithm outperforms established distributed memory and shared memory algorithms in practice. If run sequentially, it is also faster than a depth first search. We explain this by showing that although we perform more instructions than the graph search, our algorithm incurs three times fewer cache misses on sparse graphs with about a million vertices.



(a) Strong scaling **(b)** MPI time

Figure 1. Strong scaling on an Erdös-Rényi graphs ($n = 96'000$, $d = 32$). Figure 1a shows the execution time, broken down into MPI and application code; the points connected by the line represent the execution time predicted by the model. Figure 1b shows the ratio of MPI to application time. The lines represent the model prediction.

2 Preliminaries

We first introduce the relevant concepts associated with communication modeling and primitives (§ 2.1), discuss our model of randomness (§ 2.2), and then define our problem statement (§ 2.3). Finally, we overview the necessary background material (§ 2.4).

2.1 Machine and Cost Model

We now describe the unified communication model combining network communication costs and data transfers in caches; we also discuss the used collective operations and our model of randomness.

Network Model We use the Bulk Synchronous Parallel (BSP) model for network modeling and analysis. In BSP, the computation is divided into a sequence of *supersteps*. In each superstep, p processors first perform local computations and then exchange messages. In particular, local computation can only depend on messages sent in previous supersteps.

The *computation time* of a superstep is the largest number of local operations performed by any processor in that superstep. The *communication volume* of a superstep is the largest number of unit-size messages sent or received by any processor during the superstep. The communication volume and computation time of an algorithm is the sum of the communication volume and computation time of all supersteps, respectively.

Cache Model We use the *Cache-Oblivious (CO) Model* [11] to analyze cache misses. This model assumes a single fully-associative cache with an optimal replacement strategy, organized into blocks of B words and a total size of M words. B and M cannot be used in the algorithm description (hence the name *oblivious*). This model has two advantages: bounds proven for a simple two-level hierarchy generalize to an arbitrarily deep hierarchy [11], and they hold up to constant factors when executed with the Least-Recently Used (LRU) eviction policy. Throughout this paper, we assume a tall cache, that is $M \in \Omega(B^2)$. This assumption is necessary [42] and sufficient [11] to transpose an $n \times n$ matrix incurring the optimal $O(n^2/B)$ cache misses.

Unified Communication Model We model both cache effects and communication together by using the CO model within the context of BSP. To achieve this, consistently with the BSP definitions of computation time and communication volume, we define the number of cache misses of a superstep to be the largest number of cache misses incurred by any processor during the superstep.

Collective Communication We use the following *collective* operations from the Message Passing Interface (MPI) [43], of which there are practical implementations [34]. For the following, k elements $x = x_1, \ldots x_k$ are stored (as an array) at a selected root processor.

BROADCAST: The root sends all k elements to all processors.

For the following, k elements $x = x_1, \ldots, x_k$ are distributed equally among the processors and there is a root processor.

REDUCE: Let $\star$ be an associative operator. The root receives x and computes $x_1 \star x_2 \star \ldots \star x_k$.
GATHER: Each processor sends its k/p elements to the root.
ALL-REDUCE/ALL-GATHER: Computes a REDUCE / GATHER and BROADCASTS the result.

The above collectives can be implemented in $O(1)$ supersteps, $O(k)$ communication volume and time, and $O(k/B + 1)$ cache misses.

2.2 Randomness

Randomization has been a powerful tool in the design of sequential [24, 35, 44] and parallel [12, 16, 37] algorithms.

We assume that each processor has access to an independent, uniformly random, $\Omega(\log n)$-bit word in unit time.

A statement holds *with high probability* (w.h.p.) if it holds with probability at least $1 - \frac{1}{n^c}$ for all c. For simplicity, we provide proofs for fixed c, but it is straightforward to increase the probabilities.

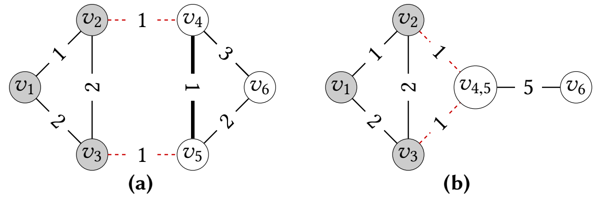


Figure 2. The two partitions of a minimum cut are indicated by the vertex shading. In 2a, the initial graph is shown. The dashed edges cross the minimum cut, with weight 2. The graph in 2b shows the result of contracting edge (v_4, v_5) — the minimum cut does not change.

2.3 Graphs, Cuts, Connected Components

Here, we present our graph model and define minimum cuts.

Graph Model We consider an undirected graph G with vertex set V, edge set $E \subseteq V \times V$, and weight function $w : E \longrightarrow \mathbb{N}_+$. We write $|V| = n$ and $|E| = m$. The weight of an edge $e = (v, u)$ is $w(v, u)$ or $w(e)$. The average degree of G is denoted by d.

Minimum Cuts A cut V' is a nonempty proper subset of V. The value of a cut is the sum of the weights of the edges with one endpoint in the cut and the other in its complement. A (global) *minimum cut* is a cut of the smallest value. A cut of value within a multiplicative factor α of the minimum cut is an *α-approximate minimum cut.*

Throughout, we assume for simplicity of exposition that the edge weights are bounded by the minimum cut value times a polynomial factor in n. This assumption can be removed by a preprocessing step [25, Section 7.1] without increasing the presented bounds.

Connected Components A graph G is said to be connected if a path exists between any pair of vertices. The connected components of G are its maximal connected subgraphs.

2.4 Fundamental Techniques

Next, we discuss the fundamental techniques we use.

Edge Contractions To *contract* an edge is to merge its endpoints into a single vertex, remove loops, and combine parallel edges, see Figure 2. This operation was used previously for evaluating expression trees [36, Chapter 3.3] and computing minimum spanning trees [24]. More importantly, repeatedly contracting random edges gives an algorithm to compute a minimum cut [25].

There is a crucial tradeoff for an approach to compute minimum cuts based on random edge contraction: The fewer vertices remain after contraction, the smaller the probability that a minimum cut survives the contraction. On the other hand, the fewer vertices remain, the faster we can process the remaining graph. For now, we keep the number of vertices that remain after the contraction as a parameter t. Specifically, a graph is *randomly contracted* to t vertices by repeatedly selecting an edge with probability proportional to its weight and contracting it, until t vertices remain.

Edge contraction does *not decrease* the value of a minimum cut. On the other hand, the value of a minimum cut *can increase* when an edge that crosses all minimum cuts is contracted.

It can be shown that a minimum cut survives random contraction with non-negligible probability. Intuitively, this holds because the total weight of the edges of a particular minimum cut is small compared to the total edge weight.

Lemma 2.1 ([25] § 2)**.** *The probability that randomly contracting a graph to t vertices does not change the value of a minimum cut is at least $t(t-1)/n(n-1)$.*

Iterated Sampling In parallel, we want to contract many random edges simultaneously for better performance. For this purpose, Karger and Stein [25] introduced Iterated Sampling. Instead of merging vertices and combining resulting parallel edges after each contraction, they randomly select a suitably sized set of edges $E' \subseteq E$. Then, they contract as many edges of E' as possible while at least t vertices remain. If the edges in E' do not suffice to reduce the number of vertices to t, they contract all the edges in E' and repeat the process. Fix some constant $0 < \sigma < 1$.

Iterated Sampling repeats the following until $|V| = t$:

① ***Sparsification*** Sample an array $E' = E'_1, \ldots, E'_s$ of $s = n^{1+\sigma}$ edges. Sample every entry by choosing an edge with probability proportional to its weight.

② ***Prefix Selection*** Find the longest prefix (subarray starting from the first element) $P = E'_1, \ldots, E'_k$ of E' such that the graph (V, P) has at least t connected components.

③ ***Bulk Edge Contraction*** Contract all the edges in P.

Notably, if the sum of the edge weights is bounded by the minimum cut value times a polynomial factor in n, then w.h.p. only $O(1)$ iterations are required [25]. Intuitively, this is because (with high probability) the sampled edges E' have a large combined weight compared to the total weight of all edges in E, thus contracting the edges in E' reduces the total edge weight significantly.

In PRAM, Iterated Sampling takes $O(\log^2 n)$ time using $O(m/\log n + n^{1+\sigma})$ processors, for any fixed $0 < \sigma < 1$. Directly implementing the PRAM version would imply $\Omega(\log^2 n)$ supersteps. In § 3 and § 4, we show how to reduce this number to only $O(1)$ supersteps. By contracting the graph until no edges are left, iterated sampling can be used to find connected components in $O(1)$ supersteps (cf. § 3.2).

Recursive Contraction Recursive Contraction [25] uses random contraction to guess the minimum cut: ① Randomly contract the graph to $\lceil n/\sqrt{2} \rceil + 1$ vertices (e.g., using iterated sampling). ② Copy the contracted graph and proceed recursively on the two copies. ③ Once the number of vertices is constant, compute the minimum cut deterministically.

Lemma 2.2 ([25] § 4)**.** *Recursive Contraction finds a particular minimum cut with probability at least $1/\Omega(\log n)$. Sequentially, it takes $O(n^2 \log n)$ time.*

3 Sparsification and Graphs

A key common step is to implement a sparsification scheme with $O(1)$ supersteps to efficiently create a sparse sample of G (§ 3.1). This first gives a simple communication-avoiding connected components algorithm based on Iterated Sampling (§ 3.2). Then, relying on this result, we derive a fast approximation algorithm to the minimum cut problem (§ 3.3). Both algorithms use $O(1)$ supersteps and have near-linear (in n) communication volume. We use an additional section (§ 4) to describe our exact minimum cut algorithm.

Graph Representation We first present our graph representations, as they are key to good performance for graph algorithms. Adjacency lists, simple and successful in PRAM settings [36], are difficult to distribute evenly across processors. Even in a graph of low d there can be many vertices of high degree that, if stored in the same subset of processors, become a bottleneck. Thus, we employ a *distributed array of edges* where each processor maintains an array of $O(m/p)$ weighted edges. Initially, the order is arbitrary. We allow for parallel edges and denote with $w_i(e_j)$ the sum of the weights of the parallel edges representing e_j stored in processor i. In particular, $w_i(e_j) = 0$ if processor i does not store e_j and $w(e) = \sum_i w_i(e)$. If a graph is sufficiently dense ($m \geq n^2/\log n$), we store it as a *distributed adjacency matrix* (AM), where every processor holds $\Theta(n/p)$ rows of the matrix. The AM representation is crucial for enabling consistent performance even on very dense graphs.

3.1 Communication-Avoiding Sparsification

Sparsification consists of sampling a sparser subgraph where each edge e is chosen with probability proportional to its weight $w(e)$. When the graph is distributed across many processors, we need to schedule the sampling carefully to avoid communication.

Intuition We sample s random edges in a batched manner. First, we choose how many edges should be sampled from each processor's slice. Then, each processor samples that many edges. Finally, the samples have to be randomly permuted, because the order matters for correctness: The probability for an edge to end up in a particular position must be the same for all positions [25].

Details ① First, every processor p_i computes the sum W_i of its slice's edge weights. These values are gathered at the root. ② The root determines how the s edges are distributed among the processors. Repeatedly (i.e., s times), it chooses a processor p_i with probability $W_i/\sum_z W_z$. The root notifies each processor of how many edges it should sample. ③ Each processor p_i chooses as many edges as requested. Repeatedly, it chooses an edge e with probability $\frac{w_i(e)}{W_i}$ and adds e to the array of sampled edges. The sampled edges are gathered at the root. ④ The root permutes the edges randomly, yielding the final sample $A = a_1, \ldots, a_s$.

Theory We show that we indeed sample edges independently in A with probability proportional to their weight.

Lemma 3.1. *The sample* $a_1, \ldots, a_s$ *is such that each element* a_i *satisfies* $P[a_i = e] = \frac{w(e)}{\sum_{e_k \in E} w(e_k)}$ *for all edges* $e \in E$.

Proof. Let $P(i)$ be the random variable for the processor whose slice was used to sample a_i. We proceed by conditioning on $P(i)$. Given that we sampled a_i from processor p_j, the probability that a particular edge e ends up in a_i is $P[a_i = e | P(i) = p_j] = w_j(e)/W_j$. To determine the probability that some processor p_j was used to sample a_i, we condition on K_j, the number of times that processor p_j was chosen by the root. This yields:

$$\begin{aligned} P[P(i) = p_j] &= \sum_k P[P(i) = p_j | K_j = k] P[K_j = k] \\ &= \frac{1}{s}\left(\sum_k kP[K_j = k]\right) = \frac{W_j}{\sum_z W_z} \quad . \end{aligned}$$

For the second equality, we used that if k samples are taken from processor p_j's slice, then the probability of a sample from p_j ending up in a given position i is k/s. This holds because we apply a random permutation to the edge samples. For the third equality, we used that $(\sum_k kP[K_j = k])$ is the mean of K_j, which is binomially distributed. We conclude:

$$\begin{aligned} P[a_i = e] &= \sum_j P[a_i = e | P(i) = p_j] P[P(i) = p_j] \\ &= \sum_j \left(\frac{w_j(e)}{W_j} \frac{W_j}{\sum_z W_z}\right) = \frac{w(e)}{\sum_{e_k \in E} w(e_k)} \quad . \end{aligned}$$

□

Lemma 3.2. *Constructing a weighted edge sample of size* s *takes* $O(1)$ *supersteps,* $O(s + p)$ *communication volume,* $O\left(s \log n + \frac{m}{p}\right)$ *time, and* $O\left(s \log n + \frac{m}{pB}\right)$ *cache misses.*

Proof. The computation of the local edges' cumulative weight takes $O(m/p)$ time. Next, each entry can be sampled in $O(\log n)$ amortized time and cache misses (w.h.p.) after a linear-time preprocessing step [25, § 5]. A processor obtains at most s samples. The communication volume is $O(p)$ to send and receive the sizes of the samples at the root and $O(s)$ at every processor to send and receive the edges in the corresponding subsample. Permuting the sample takes $O(s \log n)$ time and $O\big((s/B) \log_M s\big)$ cache misses. □

3.2 Connected Components

We now introduce a connected components algorithm that employs Iterated Sampling to contract every connected component into a single vertex. The algorithm, based on Sparsification, uses $O(1)$ supersteps and is work-efficient when $m/n^{1+\epsilon} \geq p$, for some $\epsilon > 0$. In the semi-external setting (the vertices fit into fast memory, while the edges do not) and $m \geq pBn^{1+\epsilon}$, our algorithm incurs the optimal number of cache misses, $O(m/pB)$.

Intuition We obtain a sparser subgraph (with Sparsification), for which we sequentially compute connected components. We then contract these components and repeat until no edges are left. The remaining vertices correspond to the connected components of the original input graph.

Details The root processor holds a vertex-indexed array $C = C_1, \ldots, C_n$ associating each vertex i to its connected component C_i. Initially $C_i = i$. We fix some small constant $\epsilon > 0$ and repeat the following as long as there is some edge left. ① We first sparsify the graph by choosing $n^{1+\epsilon/2}$ edges E'. We gather these edges at the root. ② The root then computes the connected components of the graph $(\{C_i \mid i \in V\}, E')$, creating a mapping g from every vertex to its connected component's label. The root broadcasts g and updates every C_i to $g(C_i)$. ③ We locally replace each edge (u, v) with $(g(u), g(v))$ and remove all loops.

Theory We now present the algorithm's bounds.

Theorem 3.3. *The communication-avoiding connected components algorithm takes, w.h.p.,* $O(1)$ *supersteps,* $O\big(n^{1+\epsilon}\big)$ *communication volume, and* $O\big(m/p + n^{1+\epsilon}\big)$ *computation time. If* $M \geq 2n$*, it takes* $O\big(m/pB + n^{1+\epsilon}\big)$ *cache misses.*

Proof. As the algorithm is essentially Iterated Sampling without Bulk Edge Contraction, $O(1)$ iterations suffice w.h.p. until all the edges are contracted. The communication bounds then follow from our Sparsification algorithm (Lemma 3.2). If g fits into cache, the renaming of the endpoints takes $O(m/pB)$ cache misses. □

By replacing the sequential connected components computation at the root with a parallel algorithm, Sparsification could be used to speed up other connected components algorithms.

Since in the connected components problem edge weights are irrelevant, we can work on unweighted graphs. This allows us to reduce the time to sparsify the graph from $O(s \log n + m/p)$ to $O(s + m/p)$, an improvement that turned out to be crucial in practice, even though it is not necessary for Theorem 3.3.

Intuition Step ② of our sparsification algorithm from § 3.1 can be avoided: each processor just samples slightly more edges than we would expect, so that with high probability enough edges are sampled. Moreover, sampling on unweighted edges is cheaper.

Details We fix some $0 < \delta < 1$. If the expected number of edges μ_i that will be sampled from processor i is at least $(9 \log n)/\delta^2$, we sample $\lceil (1+\delta)\mu_i \rceil$ edges from processor i. Otherwise, include every edge of processor i in the sample.

Theory The number of iterations does not increase if we sample too many edges from a particular processor. A Chernoff bound can be used to bound the probability that more than $(1+\delta)\mu_i$ edges would have been sampled using the original procedure. Finally, sampling can be done in $O(1)$ time per edge selection.

3.3 Approximate Minimum Cuts

Here, we sample subgraphs of varying expected sparsity and test their connectivity; the sparsity at the moment that the graph becomes disconnected estimates a minimum cut.

Intuition The connectivity of a random subgraph is related to the minimum cut value. Thus, the point at which the sampled subgraph becomes disconnected gives an estimate of the minimum cut. To implement this efficiently, we employ our communication-avoiding connected components algorithm (§ 3.2) and show that we need only a single connected components query.

Details Our algorithm computes an $O(\log n)$-approximate cut as follows. ① We compute the sum of all the edge weights W with an ALL-REDUCE. ② We perform $\lceil \ln W \rceil$ iterations, where the i-th iteration consists of $\Theta(\log n)$ repetitions (*trials*) of the following: (2.1) sample a subgraph G_i of G by keeping each edge e with probability $1-(1-2^{-i})^{w(e)}$, and (2.2) test if the graph G_i is connected.

The output of the algorithm is 2^j, where j is the smallest iteration such that in at least one of its trials the graph G_j is disconnected.

In order to parallelize the procedure efficiently, the iterations are pipelined: in each trial, every processor independently samples a subgraph from its slice of the distributed edge array. The vertices of each subgraph are assigned unique labels associated to the trials.

Finally, a single connected components computation on the union of the labeled subgraphs (over all trials of all iterations) yields the results at once.

Theory We now show the bounds. For correctness, see the online-only appendix.

Theorem 3.4. *The algorithm computes a $O(\log n)$-approximate minimum cut w.h.p. in $O(1)$ supersteps, $O(n^{1+\epsilon})$ communication volume, and $O\left(\frac{m \log^3 n}{p} + n^{1+\epsilon}\right)$ time. If $M \geq cn \log^2 n$ (for some constant c), it takes $O\left(\frac{m \log^2 n}{pB} + n^{1+\epsilon}\right)$ cache misses.*

Proof. The subgraphs can be generated in $O((m \log n)/p)$ time and $O(m/Bp)$ cache misses each. If the edge weights are polynomial in n, only $O(\log^2 n)$ subgraphs are generated overall. Note that this assumption could be removed. The bounds then follow from our results on connected components (Theorem 3.3). □

In practice, we found that it does not pay off to pipeline the outer loop. Instead, we perform the iterations one after the other and stop at the first iteration where some graph is disconnected. The number of supersteps of this variant is $O(\log \mu)$, where μ is the minimum cut value. The space is reduced by a $\log n$ factor by this change and the time becomes $O\big((m \log \mu \log^2 n)/p + n^{1+\epsilon}\big)$. This variant is faster when the minimum cut value is $o(n)$.

4 Communication-Avoiding Mincuts

We now present our exact global minimum cut algorithm, which uses Iterated Sampling (see § 2.4) at its core.

Overview To make Iterated Sampling communication-avoiding, we use our communication-avoiding implementation of Sparsification from § 3.1. In contrast to the connected components in § 3.2, here we need Bulk Edge Contraction in order to combine the parallel edges that come from contraction, so that the graph representation remains concise throughout the algorithm. We show how to implement Bulk Edge Contraction in § 4.1. To obtain consistent performance on graphs of different densities, we give two implementations: one for edge arrays and one for adjacency matrices.

Recursive Contraction (§ 2.4) combines multiple executions of random contraction to increase the probability of preserving minimum cuts and, if repeated $O(\log^2 n)$ times, it does so w.h.p. [25].

Intuition Recursive Contraction yields a fast minimum cut algorithm for dense graphs (details in § 4.3). To achieve $O(1)$ supersteps and reduce communication for sparser graphs (where $p \leq n^2/m$), we randomly contract the graph to $\Theta(\sqrt{m})$ vertices with Iterated Sampling before running Recursive Contraction once. We then repeat the sequence of the two different contractions steps a number of times. We call the first step *Eager*, as it reduces the number of vertices very quickly before proceeding recursively.

Details Our minimum cut algorithm performs a number of $t = \Theta(\frac{n^2}{m} \log^2 n)$ trials, each of which returns a cut. The result are the cuts of smallest value.

A trial has two main steps:

① ***Eager Step*** Randomly Contract the graph to $\lceil \sqrt{m} \rceil + 1$ vertices with a sparse implementation of Iterated Sampling (§ 4.2).

② ***Recursive Step*** Run Recursive Contraction using a dense implementation of Iterated Sampling to perform random contraction (§ 4.3).

We parallelize the trials depending on the number of processors. If there are more trials than processors ($p \leq t$), broadcast the graph, distribute the trials equally among the processors, and perform them sequentially. Otherwise ($p > t$), split the processors into equally sized groups. Each group performs one trial in parallel.

4.1 Bulk Edge Contraction

Given a weighted graph G and a mapping $g : V \mapsto V'$, the task of Bulk Edge Contraction is to merge the vertices of G according to g, i.e., merge the vertices that map to the same label in V'.

Dense Bulk Edge Contraction We begin with the case where the graph is represented as a distributed adjacency matrix. We use this representation in Recursive Contraction. There, the graphs can get arbitrarily dense even if the initial graph has $O(n)$ edges.

Intuition In an adjacency matrix, contraction sums the rows and the columns of the vertices that map to the same label in V'.

Details ① For every vertex i in the contracted graph, the processors set column i to be the sum of all columns which map to i. ② The processors transpose the matrix and the columns are combined as in the previous step. ③ The diagonal of the matrix is set to zero. At this point, the shrunken matrix has t rows and columns.

Theory The following Lemma shows the procedure's bounds.

Lemma 4.1. *If $p \leq n$, bulk edge contraction on a distributed adjacency matrix takes $O(1)$ supersteps, $O\big(n^2/p\big)$ communication volume and computation time. If $pB \leq n$, it takes $O\big(n^2/pB\big)$ cache misses.*

Proof. Combining the columns is a local operation, which takes $O\big(n^2/p\big)$ time and $O\big(n^2/(pB) + n\big)$ cache misses.

Transposing the matrix requires communication volume and time $O\big(n^2/p + n\big)$. □

Sparse Bulk Edge Contraction We cannot use the dense version for the Eager Step, because it would incur $\Theta\big(n^2\big)$ work for each trial. As there are $\Omega((n^2 \log^2 n)/m)$ trials, we can only afford $O(m \log n)$ work per trial to ensure $O\big(n^2 \log^3 n\big)$ work.

Therefore, we use a different paradigm for the Eager Step.

Intuition After sorting the edges globally according to their endpoints, every set of parallel edges will lie either in a single processor, or in adjacent ones. This allows us to combine them in a communication-avoiding way.

Details ① The processors locally rename the endpoints (replacing (u, v) with $(g(u), g(v))$) and remove loops. Think of the modified graph as a multigraph whose parallel edges we have to combine to obtain a graph. ② Globally sort the edges by their endpoints (first by the smaller endpoint, then by the other endpoint). ③ Each processor locally combines the parallel edges it has after the sorting. At every processor i, this gives an array $A_i = l_i^1, l_i^2, \ldots, l_i^{n_i}$, for some n_i. ④ ALL-GATHER the first edge of each of the A_i, such that each processor holds $l = l_1^1, \ldots l_p^1$. The list l can still contain parallel edges, but for each such parallel edge (u, v), there can be at most one processor which has an edge parallel to (u, v) which is not also in l. ⑤ For each distinct pair of endpoints (u, v) in l, the leftmost processor which has an edge with endpoints (u, v) increments its weight by the total weight of the edges in l that are parallel to (u, v). If processor i combined some parallel edges from l into l_i' its result is $l_i', l_i^2, \ldots, l_i^{n_i}$, otherwise it is $l_i^2, \ldots, l_i^{n_i}$.

For the proof of the following bounds, please refer to the only-only appendix.

Lemma 4.2. *If $p \leq \sqrt{m}/\log n$, sparse bulk edge contraction takes, with high probability, $O(1)$ supersteps, $O(m/p)$ communication volume, $O\Big(\frac{m}{p} \log n + n\Big)$ time, and $O\Big(\frac{m}{pB} \log_M + n/B\Big)$ cache misses.*

Notice that our edge contraction algorithm can be generalized to group values by an arbitrary comparable key and then combining them using any associative operator.

4.2 Eager Step

Running distributed edge array sparsification (Lemma 3.2) and then sparse bulk edge contraction (Lemma 4.2), we obtain a sparse implementation for Iterated Sampling. This is the Eager Step.

4.3 Recursive Step

As mentioned previously, the graphs can get arbitrarily dense after contraction. Thus, it is crucial to be efficient on dense graphs in the Recursive Step. We use a dense bulk edge contraction to obtain a communication-avoiding adaptation of Recursive Contraction.

Details ① In each recursive call, half of the processors continue to operate in parallel on one copy and the other half continue to operate on the other copy of the graph. ② Once a single processor is left, it computes a cut sequentially using CO Karger-Stein [13].

4.4 Correctness and Bounds

We illustrated the main parts of the mincut algorithm. For the proof of correctness, please refer to the online-only appendix.

Lemma 4.3. *The communication-avoiding minimum cut algorithm finds all minimum cuts w.h.p..*

As summarised in Table 1, we get the following bounds:

Theorem 4.4. *If $p \leq n^{1-\epsilon}$ for some $\epsilon > 0$, the communication-avoiding minimum cut algorithm takes, w.h.p., $O\Big(\log \frac{pm}{n^2}\Big)$ supersteps, $O\Big(\frac{n^2}{p} \log^2 n \log \frac{pm}{n^2}\Big)$ communication volume, and $O\Big(\frac{n^2}{p} \log^3 n\Big)$ computation time. If $pB \leq n^{1-\epsilon}$, it takes $O\Big(\frac{n^2 \log^3 n}{pB}\Big)$ cache misses. It uses $O\Big(\min(m, \frac{n^2 \log^2 n}{p})\Big)$ space.*

5 Experiments

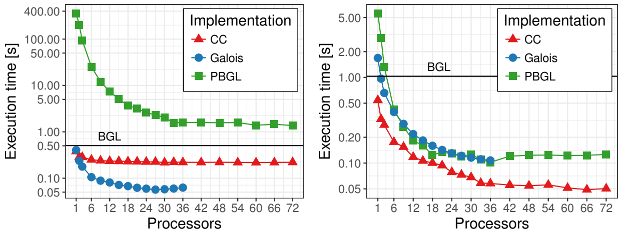
We now illustrate the advantages of our algorithms compared to the state-of-the-art. To investigate the applicability of the presented algorithms, we have devised MPI implementations in C++. We present the performance of our code on several classes of inputs in different experimental settings. Throughout the following, we refer to our implementations as Connected Components (CC), Approximate Minimum Cut (AppMC), and Minimum Cut (MC).

Experimental Setup The experiments were conducted on CSCS Piz Daint, a leadership-class Cray XC50 computer, using a homogeneous subset of nodes, each with two 18-core Intel Xeon E5-2695 v4 CPUs (3.30 GHz, 45 MiB Last Level Cache (LLC) shared among all cores) and 64 GiB DDR3 memory, using all 36 physical cores per node. The interconnect is Cray Aries (Dragonfly topology [26]). The filesystem does not influence the results.

Considered Metrics and Events We measure several metrics indicative of overall performance. All measurements are taken over the course of a single execution, which starts when the input has been loaded into memory at the initial set of nodes and ends when the result is available at a designated root processor. The metrics are: *execution time*; *time spent in MPI* (the cumulative time spent in any `MPI_*` function over the course of the execution); the number of *LLC misses*, as measured by a hardware counter; the number of *completed instructions* as measured by a hardware counter. We derive *Instructions per Miss (IPM)* as the ratio of the number of instructions to the number cache misses. The time spent in MPI is a conservative estimate of the communication cost, as it also includes synchronization costs incurred due to imbalance.

Methodology For all the metrics collected in a single execution, we always choose the maximum among all participating processors. The presented datapoints always represent the median of several executions. We collected measurements until the 95% confidence interval (CI) for the median was within 5% of the reported values, which provides a non-parametric reliability guarantee [18, 27]. In cases where we were not able to connect enough measurements, the 95% CI is shown. All the experiments involving our randomized algorithms were conducted with 0.90 minimum success probability. Each execution of every experiment uses a different fixed seed for the pseudorandom number generator (*PRNG*). We use the PRNG presented by Salmon et al. [38] to ensure uncorrelated parallel streams. We have observed no significant effect of seed choice on the performance.

Tested Inputs In the systematic experiments, we use four classes of synthetic graphs with distinct vertex degree distributions as well as spectral (and thus connectivity) properties: Watts-Strogatz [48] small-world graphs (with edge rewiring probability $p = 0.3$), Barabasi-Álbert [3] scale-free graphs, R-MAT graphs [6] (with $a = 0.45$, $b = c = 0.22$), and Erdös-Rényi $G(n, M)$ graphs [9] with a Poisson vertex degree distribution.



(a) Sparse graph **(b)** Dense graph

Figure 3. CC strong scaling on a sparse graph (3a, Barabasi-Álbert graph with $n = 1M$, $d = 32$) and a dense graphs (3b, R-MAT graph with $n = 128'000$, $d = 2'000$). The line shows the BGL execution time.

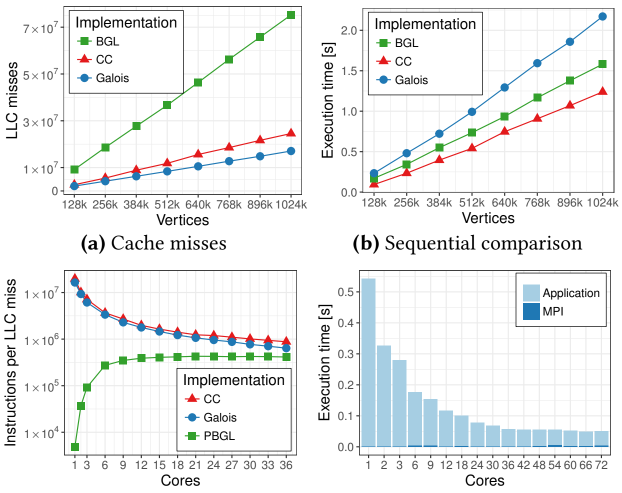
5.1 Connected Components

Baselines We compare to both the Boost Graph Library (BGL) sequential implementation [41], which uses a linear-time graph traversal, as well as to the Parallel Boost Graph Library (PBGL) implementation [15], which is based on an $O((n + m) \log n)$ work algorithm that takes $O(\log n)$ supersteps [14]. However, the PBGL implementation has weaker guarantees. Moreover, we also compare to the asynchronous implementation provided with Galois [30], a state-of-the-art shared memory graph processing framework.

Performance and Scalability We consider strong scaling on both sparse (Figure 3a) and dense (Figure 3b) graphs. On the former, CC initially demonstrates speedups over both Galois and PBGL, but shows only limited scaling. This is due to the inherently limited parallelism of our approach on sparse graphs, especially compared to that of Galois. Sequentially, we have found our algorithm to be slightly faster than both BGL and Galois, which are about one order of magnitude faster than PBGL.

In contrast, the dense graphs (Figure 3b) provide enough parallelism to allow CC to demonstrate scalability comparable to that of PBGL and Galois while being consistently faster than both of them.

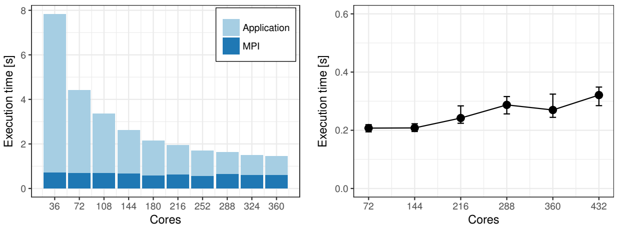
Cache Efficiency Figure 4 offers several insights into the cache and network behavior of the algorithms under consideration. Running sequentially, both CC and Galois incur significantly fewer cache misses compared to PBGL as the inputs grow larger. Interestingly, BGL exhibits worse performance despite being explicitly designed and tuned for sequential scenarios, arguably due to the following inefficiency: While BGL uses about 33% fewer instructions compared to our algorithm, our significantly higher IPM (Figure 8b) results in higher Instructions per Cycle (IPC), which offsets the extra work and produces a better trend as the problem size increases (Figure 4b). This also suggests that efficient use of the memory hierarchy has much greater practical effect than



(a) Cache misses **(b)** Sequential comparison

(c) Parallel comparison **(d)** Strong scaling

Figure 4. Sequential CC cache efficiency compared to BGL (4a, 4b) on an R-MAT graph with $d = 256$ and increasing number of vertices, and to PBGL in parallel (4c) on an R-MAT graph with $n = 128'000$, $d = 2'048$. Figure 4d shows strong scaling on the same graph as Figure 4c.



(a) Strong scaling **(b)** Weak scaling

Figure 5. AppMC scalability. Left: strong scaling on a dense graph R-MAT graph ($n = 256'000$, $d = 4'096$). Right: weak scaling with increasing edge count on an R-MAT graph with $n = 16'000$ and $2'048'000$ edges per node.

the $O(n^\epsilon)$ work inefficiency incurred by our algorithm in theory. When executed in parallel, compared to PBGL, both CC and Galois incur a lower number of misses per instruction when the parallelism is low, but the IPM is eventually matched as the parallelism is exhausted.

Network Communication We have observed that the MPI time constituted 9.6% of overall execution time on 72 cores, growing steadily from 2.8% on 36 cores. In our experiments, the MPI time ratio seems to depend on the number of nodes rather than cores, which results in plateaus or declines between some measurements.

5.2 Approximate Minimum Cuts

We compare AppMC to MC. On sparse graphs, AppMC is an order of magnitude faster than MC for the same inputs as used on Figure 1, but, as expected, it does not scale as far. However, on dense graphs with an average degree in the order of thousands (Figure 5a), AppMC scales up to hundreds of processors. Figure 5b shows that the execution time remains close to constant when the number of edges and the number of processors increase at the same rate. In particular, increasing both the number of edges and processors by a factor 8 increases the time by only a factor 1.55.

Communication Efficiency As the algorithm is built on top of CC, its communication behaves similarly (see Figure 5a): the time spent in MPI takes up about 26% of the total time on 144 cores.

5.3 Minimum Cuts

Baselines For smaller instances, we compare the execution time of MC to two sequential baselines: the BGL implementation of a deterministic $O\left(nm + n^2 \log n\right)$ time algorithm by Stoer and Wagner [45] and a cache-oblivious implementation of Karger and Stein's algorithm [13]. We refer to the baselines as SW and KS respectively. To the best of our knowledge, no parallel implementation of any (global) minimum cut algorithm has been published.

Performance Model To link the observed data with the theory, we developed and fitted a simple constant-factor performance model for our code, translating the BSP bounds to execution times. The model consists of the BSP computation time, the BSP communication volume times $\log p$ (a factor accounting for the MPI implementation overhead [19]), and a constant term for overhead.

Strong Scaling First, we analyze strong scaling on a sparse graph, where trials are not parallelized beyond their distribution among processors (Figure 1). We see that increases in parallelism yield good speedups and communication costs remain low compared to computation at less than 9% on 1008 processors. There is a slight decrease in efficiency, which is in part due to rounding in the distribution of the trials. Compared to KS, we achieve about 20-fold speedup on 144 cores and 115 on 1008 cores, whilst SW is about 40x slower than KS. The same trend applies to other classes of graphs. For Watts-Strogatz and Barabasi-Álbert graphs, we have observed around 4% difference in execution and MPI times. The sparse implementation is thus largely insensitive to graph structure, as expected.

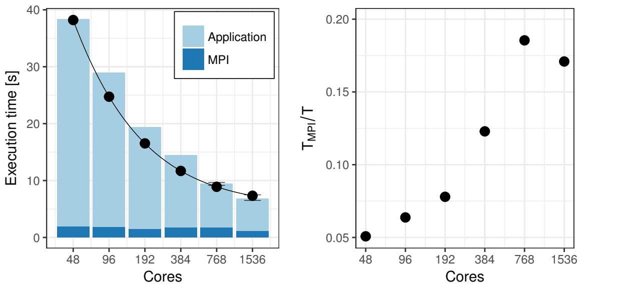


Figure 6. Strong scaling on an RMAT graph with $n = 16'000$, $d = 4'000$. Left: execution time, with the model prediction denoted by the black points. Right: Fraction of time spent in MPI.

In Figure 6, we investigate strong scaling on denser graphs. The performance trend is similar, with near-linear scaling;

the efficiency is better than on the sparse graph due to the significantly bigger input size. We also see that while communication costs decrease proportionately to p as predicted, they generally constitute a bigger fraction of the total time than for the sparse algorithm. This is largely due to the much more complex communication pattern. Finally, both of the baselines timed out on these inputs, taking more than three hours of compute time.

Weak Scaling In order to analyze the performance with an increasing input size, we present a weak scaling experiment summarized in Figure 7. Since our algorithm's execution time is close to n^2, we expect the execution time to grow linearly as we set a fixed value of n/p and increase both n and p proportionally. Indeed, the trend holds closely for both sparse and dense graphs, which implies good scalability with the increasing input size. In particular, this highlights the fact that our communication-efficient design ensures that the relative cost of communication increases only slightly with increasing scale, as predicted by our model.

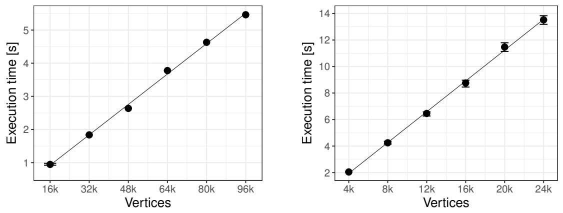
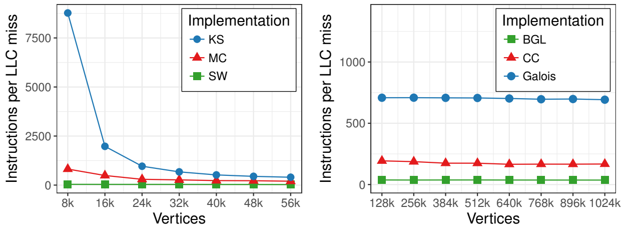
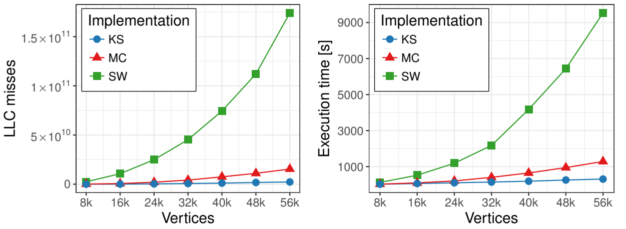


Figure 7. Weak scaling on a sparse graph (left, Watts-Strogatz graph with $d = 32$ and 4′000 vertices per node) and a dense graph (right, R-MAT with $d = 1'000$ and 2′000 vertices per node). The lines interpolate the trend.



(a) Minimum cuts **(b)** Connected components

Figure 8. IPM rate of MC (Figure 8a, setup identical to Figure 9) and CC (Figure 8b, setup identical to Figure 4).



(a) Cache Misses **(b)** Execution Time

Figure 9. Comparison of sequential cache efficiency on an Erdös-Rényi graph with $d = 32$ and a varying number of vertices.

Cache Efficiency Figure 9 offers several important insights into the cache efficiency of KS, SW, and MC. We see that while on graphs with $m = O(n)$ all three algorithms have a similar execution time of approximately $O(n^2)$, SW incurs dramatically more cache misses than both KS and MC, as shown in Figure 9a. Next, KS is significantly more efficient than MC as it was specifically designed for sequential cache efficiency, enabling a more compact representation without buffers and other intermediate structures.

6 Related Work

Both connected components and minimum cuts have been studied extensively in a variety of sequential and parallel settings.

Global Minimum Cuts Numerous sequential deterministic minimum cut algorithms have been proposed [5, 17, 29, 45], the fastest of which runs in $O\left(nm + n^2 \log n\right)$ time. Randomized algorithms have better bounds: A minimum cut can be found w.h.p. in $O\left(m \log^3 n\right)$ time [22]. Recently, this algorithm has been adapted to the cache-oblivious model [13]. A parallel algorithm [25] that runs in polylogarithmic time on a n^2 processors PRAM has been adapted to the BSP model [4]. We obtain improved bounds (see Table 1).

ST Minimum Cuts and Maximum Flows Related to the minimum cut problem is that of a minimum s-t-cut. This problem requires that the vertices s and t are separated by the cut (s is in the cut, t is not). The smallest minimum s-t-cut over all (s, t) pairs is a minimum cut. In a flow network [10], the value of a minimum s-t cut corresponds to the value of a maximum s-t-flow. Recently, there has been some practical work on performing approximate maximum s-t-flow computations in parallel [50]. However, $n - 1$ maximum s-t-flow computations are required to find a minimum cut and such an approach yields a $\Omega(mn)$ work bound [50] compared to our $O\left(m \log^3 n + n^{1+\epsilon}\right)$ work algorithm presented in § 3.3.

Connected Components Connected components can be computed with $O(\log n)$ span and $O(m + n)$ work [12]. In a cache oblivious setting, computing connected components incurs $O(m \log m)$ time and $O\left((m \log_M m)/B\right)$ cache misses w.h.p. [1]. In the BSP model, Adler et al.[2] proposed an algorithm which takes $O(1)$ supersteps, $O(m/p + n)$ computation time, and $O\left(m/p^{1+\epsilon/2}\right)$ communication volume if $p^{1+\epsilon} \leq m/n$ for some $\epsilon > 0$. Dehne et al. [8] presented a $\Theta(\log p)$ supersteps algorithm.

7 Conclusion

We designed and implemented practical communication-avoiding algorithms for connected components and minimum cuts. We obtained distributed-memory algorithms that often perform only a constant number of collective communication and synchronization operations, matching and sometimes improving on previous theoretical bounds.

A Artifact description

A.1 Abstract

This artifact provides all the executables and the associated scripts of the experiments of the paper "Communication-Avoiding Parallel Minimum Cuts and Connected Components".

A.2 Artifact check-list (meta-information)

- **Algorithm: Iterated Sparse Sampling; Parallel Minimum Cuts**
- **Program: CC; AppMC; MC**
- **Compilation: GCC 5.3 with `-O3 -march=native -flto` via CMake. Linked with Boost 1.63.0**
- **Binary: `parallel_cc`, `approx_cut`, `square_root` for the respective programs**
- **Data set: Publicly available graphs; Synthetic graphs generated by open source software**
- **Run-time environment: Cray MPICH-2 7.2.2 MPI libraries and runtime; SUSE SLES Linux 11 (kernel 3.0.101); Slurm 15.08.11**
- **Hardware: CSCS Piz Daint cluster (Cray XC50); Homogeneous set of nodes each with two 18-core Intel Xeon E5-2695 v4 CPUs (Broadwell-E, 3.30 GHz, 45 MiB LLC) and 64 GiB DDR3; Cray Aries interconnect**
- **Execution: Using** 0.9 **minimum success probability for all programs, with a new, independent random seed for every run**
- **Output: Minimum cut value, minimum cut value approximation, number of connected components depending on the program along with instrumentation measurements for each execution; Aggregated and processed separately**
- **Publicly available?: Yes**

A.3 Description

A.3.1 How delivered

Open source under the GPLv3 license. The code and parts of the dataset as well as the complete experiment automation is hosted on GitHub (https://github.com/PJK/comm-avoiding-cuts-cc).

A.3.2 Hardware dependencies

For result replication: An MPI system of comparable specification and size.

A.3.3 Software dependencies

Required:

- C++11 compiler
- Boost C++ libraries
- MPI libraries and runtime

Facultative:

- PAPI 5.4 or newer (for cache complexity measurement)
- Parallel Boost Graph Library (PBGL) and the Boost MPI utilities, which are an optional non-header-only part of Boost (for PBGL baselines)
- BASH (or a compatible shell environment), Ruby 2, standard GNU-like utilities, and the Slurm scheduling system (for experiment automation)
- Python 3 with the NetworkX package 1.10 or newer, PaRMAT
- An up to date R language platform with the following packages `ggplot2, scales, reshape2, plyr, dplyr, cowplot` packages (for statistical evaluation)

Finally, we also bundle some dependencies with the code base. More information can be found in `README.md`.

A.3.4 Datasets

We use a number of synthetic graphs as outlined in Section 5 of the paper, as well as several graphs from SNAPThese graphs were either generated or preprocessed by the scripts located in `input_generators` and `utils`. For bigger experiments, this is completely automated, as described in section A.5.

A.4 Installation

Download and unpack the artifact. Assuming and you are in a designated working folder and the source location is `/path/to/mincuts`, simply run `cmake /path/to/mincuts`
to configure the build. Some functionality has to enabled by adding the flags described in the README, e.g. `-DPAPI_PROFILE=ON`. This step will also inform you about any missing dependencies. Run the build by executing `make` in the same directory. The executables can then be located in `src/executables`. We encourage you to verify full functionality on some of the small test cases located in `test_inputs` before proceeding.

A.5 Experiment workflow

In our particular setup, we uploaded the sources to the cluster and performed the process described in the previous section using the `build_daint.sh` script.

Then, evaluation inputs were generated. Every input graph is contained in a single file, stored as a list of edges together with associated metadata.

For smaller experiments, this was done manually by invoking the generators, as described in the README. For the bigger experiments, we use scripts located in `input_generators` that often generate the complete set of inputs.

For example, in the AppMC weak scaling experiment (Figure 6 in the paper), codenamed AWX, the inputs were first generated by running `input_generators/awx_generator.sh`
which outputs the graphs in the corresponding folder.

In order to execute the experiments, we run the scripts located in `experiment_runners` .
Each script describes one self-contained experiment. Following our earlier example, we would run the

```
experiment_runners/awx.sh
```

script to execute the experiment. This submits a number of jobs corresponding to the different data points to the scheduling system.

Every job outputs a comma-separated list of values (CSV) describing properties of execution, similar to the one shown in Listing 1.

Once all the jobs finish, we filter, merge, and copy relevant data from the cluster to a local computer using

```
experiment_runners/pull_fresh_data.sh
```

which results in one CSV file per experiment or part of experiment. The output mirrors the input folder structure and is located in `evaluation/data`. For reference, we have included the measurements we used for the figures in this paper. These are located in `evaluation/data`.

Listing 1. Output format example. First line describes PAPI counter values, second one is the profiling output, consisting of input and seed information, execution and MPI time, parallelism information, and the summarized result.

```
PAPI,0,39125749,627998425,1184539166,1012658737,35015970,5382439,0.0119047
/scratch/inputs/cc1/ba_1M_16.in,5226,1,1024000,16383744,0.428972,0.011905,cc,1
```

The data is then loaded into a suite of R scripts located in `evaluation/R`. The `evaluation/R/common.R` file is perhaps of most interest, as it contains the routines that aggregate the data and verify the variance. These routines are used to build a separate analysis for every experiment. Referring back to our example experiment, the `evaluation/R/awx.R` is the script that was used to produce Figure 6.

In case the statistical significance of results is found to be unsatisfactory during this step (verified by the `verify_ci` routine found in `evaluation/R/common.R`) , we repeat the experiment execution and the following steps. One presented datapoint is typically based on 20 to 100 individual measurements.

A.6 Evaluation and expected result

Our presentation is focused on performance, which we validate against our model in Section 5 of the paper. We therefore limit ourselves to a brief description of measurement tools, and elaborate on steps we have taken to validate correctness.

A.6.1 Performance

We measure the time by a simple source-level instrumentation. Unless detailed profiling is enabled at compile-time, only a constant number of measurements is taken per execution.

We use a monotonous variant of C++
`std::chrono::high_resolution_clock`
for our timers, which guarantees that the underlying timer has at least the precision and resolution of POSIX `clock_gettime` on our system, which is more than sufficient given the relatively high quantities we deal with.

The quantity referred to as "time spent in MPI" throughout the paper is obtained by instrumenting all MPI library calls using the aforementioned timers.

The cache usage data are obtained using PAPI. On our system, all the events we used map directly to hardware counters, providing cycle-precise results. In order to prevent interference between executions of `parallel_cc`, we perform a pointer chase to ensure eviction of any data from the previous trial. It's effectiveness has been statistically verified.

A.6.2 Correctness

Since our algorithms pose the challenge of Monte Carlo randomization, we have constructed the code such that all non-determinism is controlled by a single initial seed. This enables us to easily verify that multiple executions are consistent, in the sense that executions with the same random events produce the same result.

However, this gives no particular guarantee of correctness. For this purpose, we use three main approaches for the MC:

- We have a set of corner-cases with known, deterministic cut values, generated by
  `input_generators/verification_graphs.sh`
  against which we repeatedly test.
- For smaller inputs where running a deterministic sequential algorithm is possible, we simply check the result against the reference baseline. We have observed no failures, which may in part be due to the fact that practical graph models do not provoke the worst-case behavior.
- For big inputs where this is not a possibility, we compare multiple randomly seeded runs of `square_root` on the same input and verify that all results are the same. Since every execution succeeds with probability $p \geq 0.9$ and we conduct at least 20 runs for every datapoint, the probability $p_f \leq (1-p)^{20}$ of all of them being wrong is negligible.

For the AppMC, we compare the cut value approximation with the result given by MC and have observed an approximation ratio below 11 for all inputs.

For the CC, we have looked at several test inputs in detail. The bigger graphs were checked against the BGL baseline, and we have observed no failures at all.

A.7 Experiment customization

Experiments can be modified or added by modifying the corresponding scripts in the `experiment_runners` folder.

A.8 Notes

Additional technical information is provided in the source code.

Visit https://github.com/PJK/comm-avoiding-cuts-cc to send feedback, report issues, or collaborate on further development.

References

[1] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. 1998. A Functional Approach to External Graph Algorithms. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA '98)*. Springer-Verlag, London, UK, UK, 332–343. http://dl.acm.org/citation.cfm?id=647908.740141

[2] Micah Adler, Wolfgang Dittrich, Ben Juurlink, Miroslaw Kutylowski, and Ingo Rieping. 1998. Communication-optimal Parallel Minimum Spanning Tree Algorithms (Extended Abstract). In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*. ACM, New York, NY, USA, 27–36. https://doi.org/10.1145/277651.277662

[3] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Rev. Mod. Phys.* 74 (Jan 2002), 47–97. Issue 1. https://doi.org/10.1103/RevModPhys.74.47

[4] Friedhelm Meyer auf der Heide and Gabriel T. Martinez. 1998. Communication-efficient parallel multiway and approximate minimum cut computation. In *LATIN'98: Theoretical Informatics*. Springer, 316–330.

[5] Michael Brinkmeier. 2007. A Simple and Fast Min-Cut Algorithm. *Theory Comput. Syst.* 41, 2 (2007), 369–380. https://doi.org/10.1007/s00224-007-2010-2

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn (Eds.). SIAM, 442–446. https://doi.org/10.1137/1.9781611972740.43

[7] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. 1990. The Impact of Synchronization and Granularity on Parallel Systems. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 239–248. https://doi.org/10.1145/325096.325150

[8] Frank K. H. A. Dehne, Afonso Ferreira, Edson Cáceres, Siang W. Song, and Alessandro Roncato. 2002. Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP. *Algorithmica* 33, 2 (2002), 183–200. https://doi.org/10.1007/s00453-001-0109-4

[9] Paul Erdös and Alfréd Rényi. 1959. On random graphs, I. *Publicationes Mathematicae (Debrecen)* 6 (1959), 290–297.

[10] L.R. Ford and Delbert R. Fulkerson. 1962. *Flows in networks.* Vol. 1962. Princeton Princeton University Press.

[11] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Trans. Algorithms* 8, 1 (2012), 4:1–4:22. https://doi.org/10.1145/2071379.2071383

[12] Hillel Gazit. 1986. An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph. In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. IEEE Computer Society, 492–501. https://doi.org/10.1109/SFCS.1986.9

[13] Barbara Geissmann and Lukas Gianinazzi. 2017. *Cache Oblivious Minimum Cut.* Springer International Publishing, Cham, 285–296. https://doi.org/10.1007/978-3-319-57586-5_24

[14] Steve Goddard, Subodh Kumar, and Jan F. Prins. 1994. Connected components algorithms for mesh-connected parallel computers. In *Parallel Algorithms, Proceedings of a DIMACS Workshop, Brunswick, New Jersey, USA, October 17-18, 1994 (DIMACS Series in Discrete Mathematics and Theoretical Computer Science)*, Sandeep Nautam Bhatt (Ed.), Vol. 30. DIMACS/AMS, 43–58. http://dimacs.rutgers.edu/Volumes/Vol30.html

[15] Douglas Gregor and Andrew Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)* 2 (2005), 1–18.

[16] Shay Halperin and Uri Zwick. 1996. Optimal randomized EREW PRAM Algorithms for Finding Spanning Forests and for other Basic Graph Connectivity Problems. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, 28-30 January 1996, Atlanta, Georgia.* 438–447. http://dl.acm.org/citation.cfm?id=313852.314099

[17] Jianxiu Hao and James B. Orlin. 1992. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 165–174.

[18] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems. IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC15).

[19] Torsten Hoefler, William Gropp, Rajeev Thakur, and Jesper Larsson Träff. 2010. Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues. In *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings (Lecture Notes in Computer Science)*, Rainer Keller, Edgar Gabriel, Michael M. Resch, and Jack Dongarra (Eds.), Vol. 6305. Springer, 21–30. https://doi.org/10.1007/978-3-642-15646-5_3

[20] Mark Hoemmen. 2010. *Communication-avoiding Krylov Subspace Methods.* Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Demmel, James W. AAI3413388.

[21] Keechul Jung, Kwang In Kim, and Anil K. Jain. 2004. Text information extraction in images and video: a survey. *Pattern Recognition* 37, 5 (2004), 977 – 997. https://doi.org/10.1016/j.patcog.2003.10.012

[22] David R. Karger. 2000. Minimum cuts in near-linear time. *J. ACM* 47, 1 (2000), 46–76. https://doi.org/10.1145/331605.331608

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

[24] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. 1995. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *J. ACM* 42, 2 (1995), 321–328. https://doi.org/10.1145/201019.201022

[25] David R. Karger and Clifford Stein. 1996. A new approach to the minimum cut problem. *Journal of the ACM (JACM)* 43, 4 (1996), 601–640.

[26] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly-scalable dragonfly topology. In *ACM SIGARCH Computer Architecture News*, Vol. 36. IEEE Computer Society, 77–88.

[27] Jean-Yves Le Boudec. 2010. *Performance evaluation of computer and communication systems.* EPFL Press.

[28] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters* 17, 1 (2007), 5–20. https://doi.org/10.1142/S0129626407002843

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

[30] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 456–471. https://doi.org/10.1145/2517349.2522739

[31] Simone F. Oliveira, Karl Fürlinger, and Dieter Kranzlmüller. 2012. Trends in Computation, Communication and Storage and the Consequences for Data-intensive Science. In *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012.* 572–579. https://doi.org/10.1109/HPCC.2012.83

[32] Víctor Osma-Ruiz, Juan I. Godino-Llorente, Nicolás Sáenz-Lechón, and Pedro Gómez-Vilda. 2007. An Improved Watershed Algorithm Based on Efficient Computation of Shortest Paths. *Pattern Recogn.* 40, 3 (March 2007), 1078–1090. https://doi.org/10.1016/j.patcog.2006.06.025

[33] David A. Patterson. 2004. Latency lags bandwith. *Commun. ACM* 47, 10 (2004), 71–75.

[34] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack Dongarra. 2007. Performance analysis of MPI collective operations. *Cluster Computing* 10, 2 (2007), 127–143. https://doi.org/10.1007/s10586-007-0012-0

[35] Michael Oser Rabin. 1976. Probablistic Algorithms. In *Algorithms and Complexity*, Joseph F. Traub (Ed.). Academic Press, 21–36.

[36] John H. Reif. 1993. *Synthesis of Parallel Algorithms* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[37] John H. Reif and Sandeep Sen. 1989. Polling: A New Randomized Sampling Technique for Computational Geometry. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. 394–404. https://doi.org/10.1145/73007.73045

[38] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. 2011. Parallel Random Numbers: As Easy As 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 16, 12 pages. https://doi.org/10.1145/2063384.2063405

[39] Satu Elisa Schaeffer. 2007. Survey: Graph Clustering. *Comput. Sci. Rev.* 1, 1 (Aug. 2007), 27–64. https://doi.org/10.1016/j.cosrev.2007.05.001

[40] Roded Sharan and Ron Shamir. 2000. CLICK: a clustering algorithm with applications to gene expression analysis. In *Proc Int Conf Intell Syst Mol Biol*, Vol. 8. 16.

[41] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. 2001. *Boost Graph Library: User Guide and Reference Manual, The.* Pearson Education.

[42] Francesco Silvestri. 2007. On the Limits of Cache-oblivious Matrix Transposition. In *Proceedings of the 2Nd International Conference on Trustworthy Global Computing (TGC'06)*. Springer-Verlag, Berlin, Heidelberg, 233–243. http://dl.acm.org/citation.cfm?id=1776656.1776677

[43] Marc Snir. 1998. *MPI–the Complete Reference: The MPI core*. Vol. 1. MIT press.

[44] Robert Solovay and Volker Strassen. 1977. A Fast Monte-Carlo Test for Primality. *SIAM J. Comput.* 6, 1 (1977), 84–85. https://doi.org/10.1137/0206006

[45] Mechthild Stoer and Frank Wagner. 1997. A simple min-cut algorithm. *J. ACM* 44, 4 (1997), 585–591. https://doi.org/10.1145/263867.263872

[46] Jayaram K. Udupa and Venkatramana G. Ajjanagadde. 1990. Boundary and object labelling in three-dimensional images. *Computer Vision, Graphics, and Image Processing* 51, 3 (1990), 355–369. https://doi.org/10.1016/0734-189X(90)90008-J

[47] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[48] Duncan J. Watts and Steven H Strogatz. 1998. Collective Dynamics of small-world networks. *Nature* 393 (1998).

[49] Andrew D. Wilson. 2006. Robust Computer Vision-based Detection of Pinching for One and Two-handed Gesture Input. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 255–258. https://doi.org/10.1145/1166253.1166292

[50] Yao Zhu and David F. Gleich. 2016. A parallel min-cut algorithm using iteratively reweighted least squares targeting at problems with floating-point edge weights. *Parallel Comput.* 59 (2016), 43–59. https://doi.org/10.1016/j.parco.2016.02.003