Scalable Community Detection via Parallel Correlation Clustering [Scalable Data Science]

Jessica Shi
MIT
Cambridge, Massachusetts
jeshi@mit.edu

Laxman Dhulipala
MIT
Cambridge, Massachusetts
laxman@mit.edu

David Eisenstat
Google
New York, New York
eisen@google.com

Jakub Łącki
Google
New York, New York
jlacki@google.com

Vahab Mirrokni
Google
New York, New York
mirrokni@google.com

ABSTRACT

Graph clustering and community detection are central in modern data mining. The increasing need for analyzing billion-scale data sets calls for faster and more scalable algorithms. There are certain trade-offs between quality and speed of such clustering algorithms. In this paper, we aim to design scalable algorithms that achieve high quality when evaluated based on ground truth.

We develop a generalized sequential and shared-memory parallel framework that encompass modularity and correlation clustering. This frameworks applies to LAMBDACC objective (introduced by Veldt et al.) unifying several quality measures. Our framework consists of highly-optimized implementations that demonstrably scale to large data sets of up to billions of edges and that obtain high-quality clusters compared to ground-truth data, on both unweighted and weighted graphs. Our empirical evaluation shows that this framework improves the state-of-the-art trade-offs between speed and quality of scalable community detection. For example, on a 30-core machine with two-way hyper-threading, our implementations achieve orders of magnitude speedups over other correlation clustering baselines, and up to 28.44x speedups over our own sequential baselines while maintaining or improving quality.

PVLDB Reference Format:
Jessica Shi, Laxman Dhulipala, David Eisenstat, Jakub Łącki, and Vahab Mirrokni. Scalable Community Detection via Parallel Correlation Clustering [Scalable Data Science]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jeshi96/parallel-correlation-clustering.

1 INTRODUCTION

As a fundamental tool in modern data mining, graph clustering, or community detection, has a wide range of applications spanning data mining [21], social network analysis [14], bioinformatics [22], and machine learning [20], and has been well-studied under many frameworks [1, 34]. As the need to analyze larger and larger data sets increases, designing scalable algorithms that can handle graphs with billions of edges has become a central part of graph clustering. A big challenge is to design algorithms that can achieve fast speed at high scale while retaining high quality as evaluated on data sets with ground truth. Many graph clustering algorithms have been proposed to address this challenge, and our goal is to develop a state-of-the-art algorithm from both speed and quality perspectives. In particular, we adopt a new LAMBDACC framework, introduced by Veldt *et al.* [39], which provides a general objective encompassing modularity [17] and correlation clustering [4]. Veldt *et al.* note that LAMBDACC framework unifies several quality measures, including modularity, sparsest cut, cluster deletion, and a general version of correlation clustering. Modularity is a widely-used objective that is formally defined as the fraction of edges within clusters minus the expected fraction of edges within clusters, assuming random distribution of edges. The goal of correlation clustering is to maximize agreements or minimize disagreements, where agreements and disagreements are defined based on edge weights indicating similarity and dissimilarity.

It is NP-hard to approximate modularity (or correlation clustering) within a constant factor [9], so as such, optimizing for modularity, and by extension optimizing for the LAMBDACC objective, is inherently difficult. As a result, the most successful and widely-used modularity clustering implementations focus on heuristic algorithms, notably the popular Louvain method [6]. Indeed, the Louvain method has been well-studied for use in modularity clustering, with highly optimized heuristics and parallelizations that allow them to scale to large real-world networks [25, 35–37].

In this paper, we design, implement, and evaluate a generalized sequential and shared-memory parallel framework for Louvain-based algorithms including modularity and correlation clustering. In particular, we optimize the LAMBDACC objective with demonstrably fast performance, scaling to graphs with billions of edges. We also show that there is an inherent bottleneck to efficiently parallelizing the Louvain method, in that the problem of obtaining a clustering matching that given by the Louvain method on the LAMBDACC objective, is P-complete. As such, we explore heuristic optimizations and relaxations of the Louvain method, and demonstrate their quality and performance trade-offs for the LAMBDACC objective.

As part of our comprehensive empirical study, we show that our sequential implementation is orders of magnitude faster than the

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

proof-of-concept implementation of Veldt *et al.* [39]. We note that for both LAMBDACC and correlation clustering objective, we are unaware of any existing implementation that would scale to even million-edge graphs and achieve comparable quality. We further show that our parallel implementations obtain up to 28.44x speedups over our sequential baselines on a 30-core machine.

Moreover, we show that optimizing for the correlation clustering objective is of particular importance, by studying cluster quality with respect to ground truth data. We observe that optimizing for correlation clustering yields higher quality clusters than the ones obtained by optimizing for the celebrated modularity objective. In addition, we compare our implementation to two other prominent scalable algorithms for community detection: Tectonic [38] and SCD [28] and in both cases obtain favorable results, improving both the performance and quality.

Finally, even in the highly competitive and extensively studied area of optimizing for modularity, we obtain an up to 3.5x speedup over a highly optimized parallel modularity clustering implementation in NetworKit [35].

Our code and the full version of the paper are publicly available at https://github.com/jeshi96/parallel-correlation-clustering.

Further related work. Optimizing for correlation clustering has been studied empirically in the case of complete graphs, which is equivalent to LAMBDACC objective with resolution $\gamma = 0.5$ [11, 27]. In this very restricted setting, to the best of our knowledge, the only scalable implementation is ClusterWild! [27]. We observed that it typically obtains a negative objective, which makes it unsuitable for optimizing for LAMBDACC objective.

Implementing a highly-scalable modularity clustering has been extensively studied both in the shared-memory [12, 13, 19, 25, 35, 41] and distributed memory [15, 29, 31, 33] settings. The two fastest implementations that we identify are NetworKit [35] and Grappolo [15, 19]. Both of them offer comparable performance, but we observed the NetworKit typically computes solutions with slightly larger objective, and thus we compare our implementation to NetworKit in our empirical evaluation. We also note that compared to these papers, our algorithm optimizes for a more general LAMBDACC objective.

2 PRELIMINARIES

Graph Notation. We consider weighted graphs $G = (V, E, w)$, where $w : E \to \mathbb{R}$ denotes the weight of each edge, and unweighted graphs $G = (V, E)$, where we take $w_{uv} = 1$ for all $(u, v) \in E(G)$. We use $n = |V|$ to denote the number of vertices, $m = |E|$ to denote the number of edges, and d_v to denote the degree of a vertex v.

Objective Function. We use a generalized correlation clustering objective that is equivalent to the LAMBDACC objective given by Veldt *et al.* [39]. Note that under a specific set of parameters, our objective similarly reduces to the classic modularity objective. Moreover, our definition can be more generally applied to weighted graph inputs.

We fix a clustering resolution parameter $\lambda \in (0, 1)$. We define non-negative *vertex weights* $k : V \to \mathbb{R}_0^+$, where unless otherwise specified, we take $k_v = 1$ for all $v \in V$ (a redefinition of k is required for the modularity objective). We also define the *rescaled weight* w' of each pair of vertices $(u, v) \in V \times V$ to be

$$w'_{uv} = \begin{cases} 0, & \text{if } u = v \\ w_{uv} - \lambda k_u k_v, & \text{if } (u, v) \in E \\ -\lambda k_u k_v, & \text{otherwise.} \end{cases}$$

Then, the goal is to maximize the following CC objective:

$$\text{CC}(x) = \sum_{(i,j) \in V \times V} w'_{ij} \cdot (1 - x_{ij})$$

where $x = \{x_{ij}\}$ represents the distance between vertices i and j in a given clustering. Specifically, $x_{ij} = 0$ if i and j are in the same cluster, and $x_{ij} = 1$ if i and j are in different clusters.

The modularity objective can be obtained from the CC objective by defining vertex weights v and setting λ appropriately. Note that Reichardt and Bornholdt [30] defined a modularity objective with a fixed scaling parameter $\gamma \in (0, 1)$ to be

$$Q(x) = \frac{1}{2m} \sum_{i \neq j} (A_{ij} - \gamma \frac{d_i d_j}{2m})(1 - x_{ij})$$

where $A_{ij} = 1$ if i and j are adjacent, and $A_{ij} = 0$ otherwise. Setting $\gamma = 1$, this objective is equivalent to the simpler modularity objective given by Girvan and Newman [17]. To modify CC to match the modularity objective, we set the node weights $k(v) = d_v$ for each $v \in V$, and we set the resolution $\lambda = \gamma/(2m)$. Maximizing the two objective functions are then equivalent, since they differ only by a constant factor of $1/(2m)$.

Parallel primitives. We make use of various parallel primitives in our algorithms. A *reduce* takes as input a sequence A of length n and a binary associative function f, and returns the sum A with respect to f. A *prefix sum* takes as input a sequence A of length n, an identity ε, and an associative binary operator $\oplus$, and returns the sequence B of length n where $B[i] = \bigoplus_{j<i} A[j] \oplus \varepsilon$. A *filter* takes as input a sequence A of length n and a predicate function f, and returns the sequence B containing $a \in A$ such that $f(a)$ is true, in the same order that these elements appeared in A. Additionally, we make use of efficient parallel aggregation algorithms, including a fast implementation of *parallel hash tables* [16].

3 CORRELATION CLUSTERING ALGORITHMS

In this section, we present our main algorithm for parallel Louvain-based correlation clustering. We start in Section 3.1 by describing the baseline sequential algorithm, and we introduce our parallelization in Section 3.2, along with heuristic practical optimizations that offer trade-offs between performance and quality.

3.1 Sequential Louvain Method

We begin by describing in Algorithm 1 the classic sequential Louvain method from Blondel *et al.* [6], SEQUENTIAL-CC, adapted for the correlation clustering objective. The main idea of this method is to repeatedly move vertices to the cluster that would maximize the objective, and once no vertices can be moved, compress clusters into vertices and repeat this process on the compressed graph.

Algorithm 1 Sequential Louvain method for correlation clustering

```
1:  procedure SEQUENTIAL-CC(G, k)
2:      C ← singleton clusters for each v ∈ V
3:      do
4:          σ ← random permutation of V
5:          for each v = σ(i) do
6:              Move v to the cluster in C that maximizes the CC objective
7:      while the objective CC(C) has increased
8:      if no moves were made then
9:          return C
10:     G′, k′ ← SEQUENTIAL-COMPRESS(G, C)
11:     C′ ← SEQUENTIAL-CC(G′, k′)
12:     return SEQUENTIAL-FLATTEN(C, C′)
```

In more detail, the algorithm takes as input a graph G and node weights k, and begins with singleton clusters (Line 2). Then, it iterates over each vertex in a random order, and locally moves vertices to clusters that maximize the CC objective (Lines 5 – 6).

Note that the computation necessary to determine the cluster that a vertex v should move to given the objective function is omitted from the pseudocode for simplicity, but it can be efficiently performed by maintaining in every iteration the total vertex weight of each cluster in C. More precisely, if we denote the total vertex weight of a cluster c by K_c, the change in objective of a vertex v moving from its current cluster c to a new cluster c' (where $c \neq c'$) is given by

$$\left(\sum_{u \in c', (u,v) \in E} w_{uv} - \lambda k_v K_{c'}\right) - \left(\sum_{u \in c, (u,v) \in E} w_{uv} - \lambda k_v K_c + \lambda k_v^2\right).$$

In other words, the change in objective depends solely on K_c, $K_{c'}$, and the weights of the edges from v to its neighbors in c and c'.

After all vertices have been moved, the algorithm repeats this step of locally moving vertices until no vertices have performed non-trivial moves (Line 7). If no vertices changed clusters in Lines 5 – 6 in the first iteration of this step, then SEQUENTIAL-CC terminates (Line 9). However, if vertices did move during this step, once a stable state has been achieved, the algorithm compresses the graph G (SEQUENTIAL-COMPRESS, Line 10) by creating a new graph G', where each cluster c in G corresponds to a vertex in G' with vertex weight equal to K_c. Edges (u, v) in G are maintained as edges between the vertices corresponding to their clusters in G', where multiple edges incident on the same vertices are combined into a single edge with weight equal to the sum of their weights.

Finally, the algorithm recurses on G' and k'. The algorithm takes the returned clustering C' on the compressed graph G', and composes it with the original clustering C (SEQUENTIAL-FLATTEN, Line 12). It assigns the cluster of a vertex v in G to be the cluster of its corresponding vertex in the compressed graph G', composing the clustering obtained in the recursion onto the original graph.

Note that SEQUENTIAL-COMPRESS and SEQUENTIAL-FLATTEN can be efficiently parallelized, which we discuss in Section 3.2.4. One of the main bottlenecks in parallelizing SEQUENTIAL-CC is the sequential dependencies in moving each vertex v to the cluster in C that maximizes the objective (Lines 5 – 6). Importantly, we prove the following theorem in the appendix.

THEOREM 3.1. *The problem of obtaining a clustering equivalent to that given by the Louvain method maximizing for the* CC *objective is P-complete.*

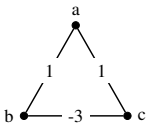

Figure 1: An example where parallel local vertex moves lowers the total objective. Assume $\lambda = 0$ and initial clusters are singletons with objective 0. If b and c are both scheduled to move at the same time, they each choose cluster $\{a\}$, leading to a single cluster $\{a, b, c\}$ with objective -1.

Algorithm 2 Parallel Louvain method for correlation clustering

```
1:  procedure BEST-MOVES(G, k, num_iter, C)
2:      Define id(c) to be the index of cluster c ∈ C
3:      D ← array of size n
4:      V′ ← V
5:      for i in range(num_iter) do
6:          parfor v ∈ V′ do
7:              D[v] ← id(c) such that moving v to c ∈ C would maximize
    CC(C)
8:              Move v to D[v]
9:          if no moves were made in iteration i then break
10:         V′ ← { neighbors of v | v moved}
11:     return C

1:  procedure PARALLEL-CC(G, k, num_iter)
2:      C ← singleton clusters for each v ∈ V
3:      C ← BEST-MOVES(G, k, num_iter, C)
4:      if no moves were made in BEST-MOVES then
5:          return C
6:      G′, k′ ← PARALLEL-COMPRESS(G, C)
7:      C′ ← PARALLEL-CC(G′, k′, num_iter)
8:      C ← PARALLEL-FLATTEN(C, C′)
9:      C ← BEST-MOVES(G, k, num_iter, C)
10:     return C
```

A key component of our proof shows that the problem of obtaining a clustering that is equivalent to any clustering C obtained following Lines 2 – 7 is P-complete. Intuitively, this is because a single vertex's move can affect another vertex's desired move, potentially leading to a long chain of sequential dependencies.

As such, to obtain an empirically efficient implementation, we heuristically relax the sequential dependency and allow vertices to move to clusters concurrently. Thus, vertices may move to clusters that would individually maximize the objective, but in tandem lead to a lower total objective; there is no guarantee of convergence. We show in Figure 1 a simple example of this. Note that this is a common parallelization technique in Louvain methods for modularity clustering, and it has been observed that in practice this technique converges for the modularity objective [35].

We now discuss optimizations that can be used with this relaxation to improve the performance of the parallel Louvain method.

3.2 Parallel Louvain Method and Optimizations

Algorithm 2 contains the pseudocode for each of the main optimizations that we consider in our parallelization of the Louvain method for the CC objective. Each optimization is highlighted in blue, and we display in the pseudocode only the optimal settings that offer a reasonable trade-off between quality and performance, as we show in

Section 4.1. We discuss in the following sections the other modular options available in our framework for each of these optimizations.

Note that the heuristic relaxation to allow vertices to move concurrently to their desired clusters is encapsulated in the subroutine BEST-MOVES. We include an additional parameter, *num_iter*, which bounds the number of iterations in which we move each vertex to its desired cluster; this is necessary due to the lack of guarantee of convergence. Moreover, in order to allow each vertex to efficiently compute its best move, we maintain the total vertex weights K_c of each cluster c, which is not shown in the pseudocode for simplicity.

3.2.1 Optimization: Synchronous vs Asynchronous. The first optimization involves appropriately scheduling individual vertex moves in BEST-MOVES, on Line 8. We explore two main options: *synchronous* and *asynchronous* vertex moves. Note that these options have been previously studied in the context of parallelizing the Louvain method for the modularity objective [33, 35].

In the synchronous setting, instead of moving vertices on Line 8 immediately after the computation of their desired cluster, we move all vertices in parallel to their desired cluster $D[v]$ after the parallel for loop on Line 6. This can be efficiently performed by in parallel aggregating vertices that move from and to the same clusters.

In the asynchronous setting, we perform vertex moves on Line 8 as highlighted in blue. Note that moving a vertex v in this manner potentially interferes with the computation that each vertex performs on Line 7, where other vertices' computations of their desired cluster may depend on v's current cluster, and the total vertex weight of v's prior and v's new cluster. Instead of using locks or other methods of synchronization, we relax consistency guarantees, and we perform separate atomic operations to update the cluster that v moves to and to update the total vertex weight of the cluster that v moves to. Thus, there is no guarantee that the stored total vertex weights of clusters represent the actual total vertex weights of the clusters.

We show in Section 4.1 that the asynchronous setting outperforms the synchronous setting with comparable objective obtained. Note that previous uses of the Louvain method for other objectives explored different schedules for local vertex moves, which offer more granular trade-offs than the two options discussed [3, 25]. We found that our asynchronous setting outperforms certain variations that maintain consistency guarantees.

3.2.2 Optimization: All Vertices vs Neighbors of Clusters vs Neighbors of Vertices. We now consider optimizations that reduce the space of vertices to consider moving in every iteration of BEST-MOVES. Specifically, when considering vertices on Lines 6 – 7, we note that following a set of vertex moves in the previous iteration, we can reduce the number of vertices that would be likely to be induced to change clusters by the vertex moves in the previous iteration. This idea applies to both the sequential and the parallel algorithms, and has been previously used in work on the Louvain method for the modularity objective [3, 26]. Importantly, as we show in Section 4.1, performing the BEST-MOVES subroutine takes a significant portion of total clustering time, and reducing the space of vertices to consider offers significant performance improvements.

In more detail, isolating a vertex v which has in a previous iteration moved from cluster c to cluster c', the vertices that would be affected by this move in the next iteration belong to three categories: (a) neighbors of v, (b) neighbors of any vertex in c that are not neighbors of v, and (c) vertices in c' that are not neighbors of v. Any vertex u that is neither a neighbor of v nor a neighbor of c, and that is not in c', is not induced to move clusters due to v's move. This is due to the change in objective formula from Section 3.1.

In our algorithm, we consider three options for this optimization: restricting considered vertices to *neighbors of vertices* moved in the previous iteration, restricting considered vertices to *neighbors of clusters* that vertices have moved to in the previous iteration, and considering *all vertices* in each iteration. The first option corresponds to the update on V' in Line 10, highlighted in blue. The second option would instead replace this line with setting $V' \leftarrow \{$ neighbors of the current cluster $C[i] \mid v$ moved from cluster $C[i]$ to cluster $c\}$, and the final option would set $V' \leftarrow V$.

We show in Section 4.1 that restricting V' to the set of neighbors of vertices that have moved in the previous iteration outperforms both other options while maintaining comparable objective.

3.2.3 Optimization: Multi-level Refinement. Finally, we consider a popular multi-level refinement optimization [32, 35]. Note that the first phase of our parallel algorithm and the classic Louvain method involves what can be viewed as successive coarsening steps, in which we perform best vertex moves and compress the resulting clustering into a coarsened graph, and recurse on the coarsened graph. Then, following each coarsening step, we flatten the clustering obtained from the coarsened graph into the current graph. The *multi-level refinement* optimization performs a refinement step following each flattening, where the flattened clustering is further refined by performing another iteration of BEST-MOVES prior to returning the clustering to the previous recursive call.

This refinement optimization is shown on Line 9, highlighted in blue. Omitting this line removes the optimization. The optimization increases the space usage and the amount of time required for our implementation, since it requires each compressed graph to be maintained throughout and since it adds an additional subroutine, but it non-trivially improves quality, as we show in Section 4.1.

3.2.4 Other Optimizations. We make use of other practical optimizations in our parallel implementation of PARALLEL-CC. First, we use the theoretically efficient parallel primitives available in the Graph Based Benchmark Suite (GBBS) [8]. Importantly, we use the EDGEMAP primitive from GBBS to maintain the frontier of neighbors of moved vertices or of modified clusters in each step of BEST-MOVES. EDGEMAP takes a vertex subset and applies a user-defined function to generate a new vertex subset – in our case, generated from specified neighbors. The primitive switches between a sparse and a dense representation of the subset depending on size, and the implementation of EDGEMAP similarly changes depending on the size of the input subset and the number of outgoing edges.

We also efficiently parallelize the sequential graph compression and cluster flattening subroutines, SEQUENTIAL-COMPRESS and SEQUENTIAL-FLATTEN respectively. Flattening a given clustering C to the clustering C' from the coarsened graph can be parallelized by maintaining a set of cluster IDs for each vertex (given by the index in $[0, n]$ of the cluster in C), and assigning for each vertex, the cluster ID in C' corresponding to the cluster containing its cluster ID in C. Moreover, we parallelize graph compression by aggregating in parallel the edges in the original graph by the cluster IDs of

their endpoints, and using parallel reduces to combine edges whose endpoints correspond to the same cluster ID.

Furthermore, in computing each vertex's desired cluster on Line 7, we make use of a parallel and a sequential subroutine, which we choose heuristically depending on the degree of the vertex. The change in objective for moving a vertex v to other clusters can be efficiently parallelized by iterating through v's neighbors in parallel and using a parallel hash table to maintain the sum of edge weights to neighbors in the same cluster. However, for vertices of small degree and with large V', the parallel overhead of maintaining such a hash table for each vertex is too costly. On the other hand, for vertices of large degree and with small V', the parallel overhead is negligible compared to the improved depth in utilizing a parallel hash table. We use a fixed threshold to choose between using the sequential subroutine versus using the parallel subroutine.

4 EXPERIMENTS

In this section, we present a comprehensive evaluation of our algorithms, demonstrating significant speedups over state-of-the-art implementations and high-quality clusters compared to ground truth.

Environment. We run most of our experiments on a c2-standard-60 Google Cloud instance, with 30 cores (with two-way hyper-threading), 3.8GHz Intel Xeon Scalable (Cascade Lake) processors, and 240 GiB main memory. We use for experiments on large graphs a m1-megamem-96 Google Cloud instance, with 48 cores (with two-way hyper-threading), 2.7GHz Intel Xeon Scalable (Skylake) processors, and 1433.6 GiB main memory. We compile our programs with g++ (version 7.3.1) and the `-O3` flag, and we use an efficient work-stealing scheduler [5]. We also terminate any experiment that takes over 7 hours.

Graph Inputs. We test our implementations on real-world undirected graphs from the Stanford Network Analysis Project (SNAP) [24], namely com-dblp, com-amazon, com-livejournal, com-orkut, and com-friendster. We also use twitter, a symmetrized version of the Twitter graph representing follower-following relationships [23]. The details of these graphs are shown in Table 1.

To show the clustering quality of our implementations, we compare against the top 5000 ground-truth communities given by SNAP for the graphs amazon and orkut. These ground-truth communities may overlap, so in order to compute average precision and recall, for each ground-truth community c, we match c to the cluster c' with the largest intersection to c. This matches the metric used by Tsourakakis *et al.* in the evaluation of TECTONIC [38].

We also use an approximate k-NN algorithm to construct weighted graphs from pointset data. Specifically, we use the Optical Recognition of Handwritten Digits (digits) dataset (1,797 instances) and the Letter Recognition (letter) dataset (20,000 instances) from the UCI Machine Learning repository [10], both of which also have ground truth clusters which we compare to. We use the state-of-the-art ScaNN k-NN library [18] to perform k-NN, with $k = 50$ and using cosine similarity. We symmetrize the resulting k-NN graph.

Finally, we show the scalability of our implementations using synthetic graphs generated by the standard rMAT graph generator [7], with $a = 0.5$, $b = c = 0.1$, and $d = 0.3$. We generate two sets of rMAT graphs: sparse graphs, where the number of edges $m = 50n$, and very sparse graphs, where the number of edges $m = 5n$.

Graphs	Num. Vertices	Num. Edges
amazon	334,863	925,872
dblp	317,080	1,049,866
livejournal	3,997,962	34,681,189
orkut	3,072,441	117,185,083
twitter	41,652,231	1,202,513,046
friendster	65,608,366	1,806,067,135

Table 1: Sizes of graph inputs.

All experiments are run on the c2-standard-60 instances, except for experiments on the twitter and friendster graphs, which are run on the m1-megamem-96 instances due to the higher memory requirement, particularly when using multi-level refinement.

Implementations. We test the Louvain-based implementations of our sequential and parallel correlation clustering algorithms (SEQ-CC and PAR-CC respectively). We also redefine vertex weights and λ as discussed in Section 2 to obtain sequential and parallel modularity clustering implementations (SEQ-MOD and PAR-MOD respectively). For our parallel implementations, we use $num\_iter = 10$ unless otherwise specified. For our sequential implementations, we use the superscript $^{\text{CON}}$ if we run the implementation to convergence (without restricting the number of iterations), and we use no superscript if we run the implementation with $num\_iter = 10$.

Also, excepting experiments on the large graphs twitter and friendster, we run each experiment 4 times and report the average time. When we use the asynchronous setting from Section 3.2.1, the algorithm is inherently non-deterministic and the objective can vary; in this case, we report the average objective obtained over all 4 runs.

Finally, our algorithms run seamlessly on weighted graphs, and unweighted graphs are treated as weighted graphs with unit weights. For weighted graphs G, we test both our implementations treating G as an unweighted graph (with unit weight edges), and our implementations treating G as a weighted graph. We denote the former with no superscript, and the latter with the superscript $^{\text{W}}$.

We compare our implementations to two correlation clustering implementations, namely the parallel C4 and CLUSTERWILD! given by Pan *et al.* [27], which are based on the sequential correlation clustering algorithm KWIKCLUSTER [2], and the sequential Louvain-based implementation in the correlation clustering framework LAMBDACC, by Veldt *et al.* [39]. We also compare to two state-of-the-art community detection algorithms which form communities based on the triangle counts: TECTONIC by Tsourakakis *et al.* [38], and SCD, by Prat-Pérez *et al.* [28]. Both these algorithms were shown to deliver superior quality to multiple similarly scalable baseline methods. We note that TECTONIC implementation is sequential, while SCD takes advantage of shared-memory parallelism. In the special case of modularity, we compare to the parallel Louvian-based modularity clustering implementation in the NetworKit toolkit (NETWORKIT), by Staudt and Meyerhenke [35].

4.1 Tuning Optimizations

We evaluated the effectiveness of the different optimizations discussed in Section 3.2, namely, considering *synchronous* versus *asynchronous* vertex moves, considering *all vertices* versus *neighbors of clusters* that vertices have moved to versus *neighbors of vertices* that have moved as the vertex subset V' to iterate over, and considering *multi-level refinement* versus *no refinement*. We establish here that the optimizations that offer reasonable trade-offs between speed and

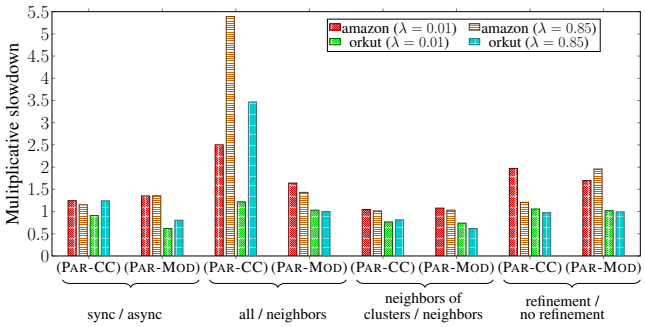

Figure 2: Multiplicative slowdown in average time of the optimizations synchronous over asynchronous, all vertices over neighbors of vertices, neighbors of clusters over neighbors of vertices, and multi-level refinement over no refinement, fixing all other optimizations in each comparison. Running times were obtained for PAR-CC and PAR-MOD on the graphs amazon and orkut, with $\lambda = 0.01, 0.85$ for PAR-CC and $\gamma = 0.01, 0.85$ for PAR-MOD.

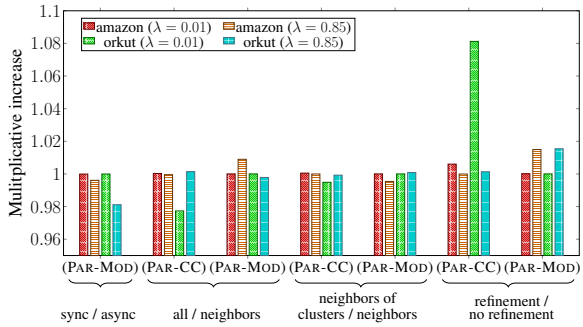

Figure 3: Multiplicative increase in the CC objective and modularity for PAR-CC and PAR-MOD respectively, of the optimizations synchronous over asynchronous, all vertices over neighbors of vertices, neighbors of clusters over neighbors of vertices, and multi-level refinement over no refinement, fixing all other optimizations in each comparison. Running times were obtained on the graphs amazon and orkut, with $\lambda = 0.01, 0.85$ for the CC objective and $\gamma = 0.01, 0.85$ for the modularity objective. Note that the increase in the CC objective for synchronous over asynchronous is omitted because the objective is negative in the synchronous setting for $\lambda = 0.85$.

quality are asynchronous vertex moves, considering neighbors of vertices that have moved as V', and using multi-level refinement.

Note that we tuned these optimizations on the graphs amazon and orkut, with $\lambda = 0.01, 0.85$ and $\gamma = 0.01, 0.85$. For each set of optimizations, fixing all of the other optimizations and considering both PAR-CC and PAR-MOD, Figure 2 shows the multiplicative slowdowns of synchronous over asynchronous, all vertices over neighbors of vertices, neighbors of clusters over neighbors of vertices, and multi-level refinement over no refinement. Figure 3 similarly shows the multiplicative increase in objective for these optimizations. Figure 4 shows the timing breakdown between the subroutines BEST-MOVES, PARALLEL-COMPRESS, and the multi-level refinement step, across each set of optimizations, using PAR-CC on orkut and $\lambda = 0.85$.

Fixing all vertices and no refinement, we see speedups of up to 1.35x using the asynchronous over the synchronous setting across both graphs and across PAR-CC and PAR-MOD. For PAR-CC, the objective obtained is negative in the synchronous setting for $\lambda = 0.85$ (and as such is omitted from Figure 3), and using the asynchronous setting, we see a 1.27–155.97% increase in objective. For PAR-MOD,

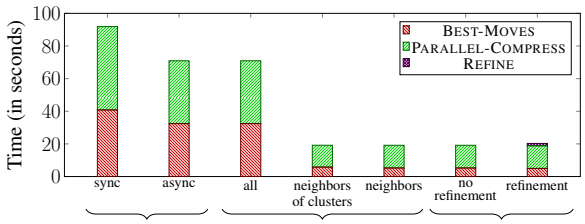

Figure 4: For each set of optimizations, fixing all other optimizations, the running times of PAR-CC on orkut using $\lambda = 0.85$, with the breakdowns for the subroutines BEST-MOVES, PARALLEL-COMPRESS, and the multi-level refinement step.

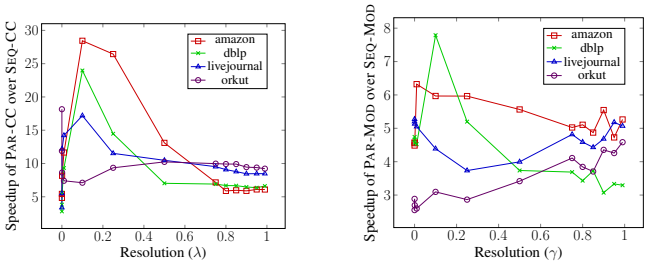

Figure 5: Speedup of PAR-CC over SEQ-CC (left) and of PAR-MOD over SEQ-MOD (right), on amazon, dblp, livejournal, and orkut, for varying resolutions.

we see a 0.0035 – 1.92% increase in modularity using the asynchronous setting over the synchronous setting. Because of the scenarios where the objective obtained is poor in the synchronous setting, we fix the asynchronous setting for the rest of our experiments.

Fixing the asynchronous setting and no refinement, we see up to a 5.39x speedup considering neighbors of vertices compared to all vertices as the subset V', and we see up to a 1.08x speedup considering neighbors of vertices compared to neighbors of clusters. Note that for orkut, neighbors of clusters does outperform neighbors of vertices by up to 1.62x. However, there is variance in the objective obtained due to the asynchronous setting, and the standard deviation in the running time is significantly higher in the neighbors of clusters setting compared to the neighbors of vertices setting. The standard deviation in the neighbors of clusters setting represents up to 40.01% of the mean, whereas the standard deviation in the neighbors of vertices setting represents up to 4.32% of the mean. Moreover, the objectives obtained in both settings are comparable, with objectives between 1.00 – 1.02x the objective obtained using all vertices in PAR-CC and PAR-MOD. We see in Figure 4 that using neighbors of clusters and neighbors of vertices offer significant time savings in both the BEST-MOVES steps and the PARALLEL-COMPRESS steps. As such, given the comparable performance of both the neighbors of clusters and neighbors of vertices settings, we fix the neighbors of vertices setting for the remainder of our experiments.

Finally, fixing the asynchronous setting and using neighbors of vertices, we see slowdowns of up to 1.97x using multi-level refinement, compared to using no refinement. However, using multi-level refinement, we see up to a 1.08x increase in objective across both the CC objective and modularity; thus, we fix multi-level refinement for the remainder of our experiments.

4.2 Speedups and Scalability

Speedups. We first note that there exist no prior scalable correlation clustering baselines that offer high quality in terms of objective. The existing implementations are the parallel C4 and ClusterWild! [27], which are based on a maximal independent set algorithm, and the sequential Louvain-based method in LambdaCC [39].

The parallel correlation clustering implementations C4 and ClusterWild! [27] optimize for the same objective as our CC objective, if we set $\lambda = 0.5$; importantly, they do not generalize to other resolution parameters, and they do not take into account weighted graphs. We test the asynchronous versions of C4 and ClusterWild!, which also outperform the synchronous versions while maintaining the objective, on the graphs amazon, dblp, livejournal, and orkut. Both implementations offer significant speedups over Par-CC, of up to 139.43x and 428.64x respectively. However, rescaling the objectives given by C4 and ClusterWild! to match the CC objective, we see that C4 and ClusterWild! decrease the objective by 273.35 – 433.31% over Par-CC. Notably, the objectives given by C4 and ClusterWild! are often negative, meaning that it is unsuitable for optimizing for the CC objective.

Additionally, we compare against the Louvain-based sequential correlation clustering implementation in LambdaCC given by Veldt *et al.* [39]. Unfortunately, this implementation does not scale to large graphs of more than hundreds of vertices. We were able to test LambdaCC on the karate graph [40], which consists of 34 vertices and 78 edges. For $\lambda = 0.01$, LambdaCC takes 0.057 seconds to cluster the karate graph, whereas our Par-CC takes 0.0002 seconds.

As such, we demonstrate the speedups of our parallel implementations primarily against our own sequential implementations, which include the applicable optimizations discussed in Section 4.1, namely considering only neighbors of vertices when computing best local moves, and multi-level refinement.

Figure 5 shows the speedup of Par-CC and Par-Mod over Seq-CC and Seq-Mod respectively. We also compared to Seq-CC$^{\text{CON}}$ and Seq-Mod$^{\text{CON}}$; we note that running to convergence generally increases the running time while improving the objective, although the improvements are not always significant. However, as we show later in Section 4.3, the average precision-recall of Seq-CC diverges significantly from that of Seq-CC$^{\text{CON}}$, while our Par-CC matches the average precision-recall of Seq-CC$^{\text{CON}}$.

On the unweighted graphs amazon, dblp, livejournal, and orkut, and over varying resolutions, we see 2.80–28.44x speedups of our Par-CC over Seq-CC, and 13.42–124.10x speedups of Par-CC over Seq-CC$^{\text{CON}}$. Our parallel implementations also achieve between 0.98–1.08x the CC objective of our serial implementations, demonstrating high performance while maintaining the CC objective. For Par-Mod, we see between 2.87–7.79x speedups over Seq-Mod, and 2.64–7.81x speedups over Seq-Mod$^{\text{CON}}$, while achieving 1.00–1.06x the modularity of the serial implementations.

Over the large unweighted graphs twitter and friendster, considering $\lambda = 0.01, 0.85$ and $\gamma = 0.01, 0.85$, we see 8.83–14.45x speedups of Par-CC over Seq-CC, and 2.65–12.04x speedups of Par-Mod over Seq-Mod respectively. For $\lambda = 0.01$, Seq-CC times out on both graphs, while Par-CC finishes in 1,042.70 seconds for twitter and 23,608.77 seconds for friendster. Moreover, Par-CC and Par-Mod match the objectives obtained by their sequential counterparts,

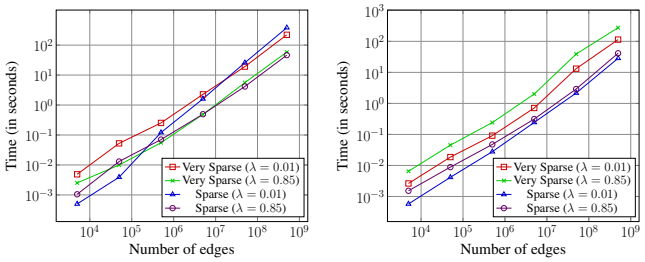

Figure 6: Scalability of Par-CC (left) and Par-Mod (right) over sparse and very sparse rMAT graphs with varying numbers of edges.

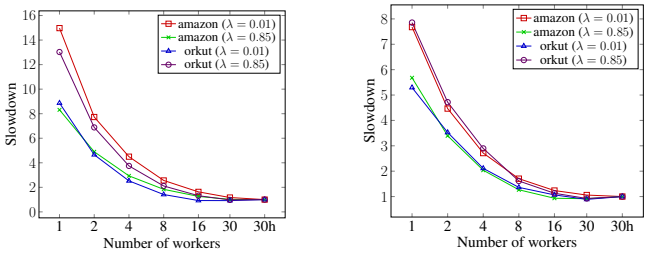

Figure 7: Scalability of Par-CC (left) and Par-Mod (right) over different numbers of threads, on amazon and orkut with $\lambda = 0.01, 0.85$. Note that 30h indicates 30 cores with two-way hyper-threading.

of between 0.96–0.99x the CC objective and between 0.93–1.08x the modularity of the sequential implementations respectively.

Scalability. Figure 6 demonstrates the scalability of Par-CC and Par-Mod over rMAT graphs of varying sizes, with very sparse graphs where $m = 5n$ and with sparse graphs where $m = 50n$. We see that for both of our algorithms and across different resolution parameters ($\lambda = 0.01, 0.85$), the running times of our algorithms scale nearly linearly with the number of edges. Figure 7 shows the speedups of Par-CC and Par-Mod on amazon and orkut over different numbers of threads. Overall, we see good parallel scalability, with 8.31–14.97 self-relative speedups on Par-CC, and 5.29–7.96x self-relative speedups on Par-Mod.

Comparisons to Other Implementations. In the special case of modularity, we compare against the highly optimized parallel modularity clustering implementation NetworKit [35]. Note that NetworKit, like Par-Mod, implements an asynchronous version of Louvain-based modularity clustering, and requires a parameter *num_iter* to guarantee completion. By default NetworKit sets *num_iter* = 32, so to compare with Par-Mod, we similarly set *num_iter* = 32. We show in the appendix the speedups of Par-Mod over NetworKit considering a range of resolutions on the graphs amazon, dblp, livejournal, and orkut. We see up to 3.50x speedups, primarily due to our optimization of the graph compression step in the Louvain-based algorithm, and on average 1.89x speedups. We also obtain between 0.99 – 1.00x the modularity given by NetworKit's implementation, where some variance appears due to the asynchronous nature of both implementations.

For the twitter graph, setting $\gamma = 0.01, 0.85$, Par-Mod gives between 1.08 – 3.03x speedups over NetworKit while maintaining comparable modularity. For the friendster graph, we turn off NetworKit's turbo parameter (which offers a trade-off between memory usage and performance) due to space constraints, and setting $\gamma = 0.01, 0.85$, Par-Mod gives between 1.23–1.26x speedups over NetworKit while maintaining comparable modularity.

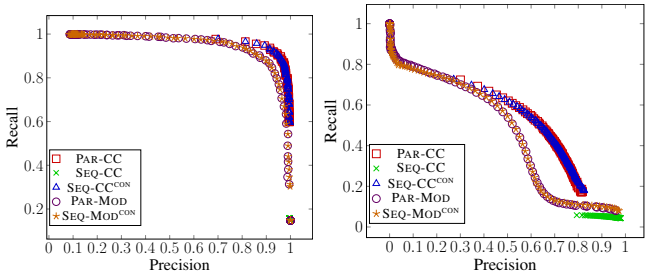

Figure 8: Average precision and recall compared to ground truth communities on amazon (left) and orkut (right) of the clusters obtained from our parallel implementations PAR-CC and PAR-MOD, compared to our sequential implementations SEQ-CC and SEQ-MOD, using varying resolutions.

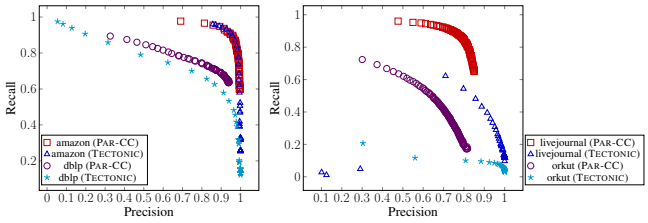

Figure 9: Average precision and recall compared to ground truth communities on amazon and dblp (left), and on livejournal and orkut (right), of the clusters obtained from PAR-CC using varying resolutions, and from TECTONIC using varying θ.

Furthermore, NETWORKIT's implementation is also able to take into account edge weights of weighted graphs. We discuss their quality with respect to weighted graphs in Section 4.3.

Additionally, we compare to the sequential TECTONIC implementation [38], which clusters based on the idea of triangle conductance and provides demonstrably good average precision-recall compared to ground truth communities on SNAP graphs. Like PAR-CC, TECTONIC uses a parameter θ that can be set to achieve a range of clusters with varying average precision and recall. We discuss in more detail in Section 4.3 the comparison between the quality of PAR-CC's and TECTONIC's clusters, but for outputs where PAR-CC either outperforms or matches TECTONIC in terms of average precision and recall considering $\lambda \in \{0.01x \mid x \in [1, 99]\}$ and $\theta \in \{0.01x \mid x \in [1, 299]\}$ respectively, we see between 2.48–67.62x speedup of PAR-CC over TECTONIC on the graphs amazon, dblp, livejournal, and orkut. Notably, PAR-CC significantly outperforms TECTONIC on large graphs, with between 34.22–67.62x speedups on orkut.

Finally, we compare to SCD [28], a parallel triangle-based community detection implementation. Note that SCD is not able to vary parameters to obtain clusters with significantly different average precision and recall. For amazon, dblp, and livejournal, our PAR-CC implementation achieves 2.00–2.89x speedups over SCD while maintaining the same average precision and recall. For orkut, SCD obtains an average precision of 0.15 and an average recall of 0.05, while PAR-CC can obtain an average precision of 0.61 and an average recall of 0.53 with 1.31x speedup over SCD.

4.3 Quality Compared to Ground Truth

Unweighted graphs. Figure 8 shows the average precision-recall curves obtained by PAR-CC and PAR-MOD, varying resolutions, compared to the top 5000 ground truth communities on the unweighted graphs amazon and orkut. For PAR-CC, we set $\lambda \in \{0.01x \mid x \in [1, 99]\}$, and for PAR-MOD, we set $\gamma \in \{0.02 \cdot (1.2)^x \mid x \in [1, 99]\}$. We compare these curves to those obtained by SEQ-CC and SEQ-MOD, running with both the same number of iterations ($num\_iter = 10$) as the parallel implementations, and to convergence. Note that we only show SEQ-MOD$^{\text{CON}}$, because the version restricting the number of iterations displays precisely the same average precision-recall curve as the version running to convergence.

Overall, we see that the average precision-recall obtained by PAR-CC and PAR-MOD match those obtained by their sequential counterparts for both amazon and orkut. Note that if we do not run SEQ-CC to convergence, we obtain relatively poor precision-recall compared to PAR-CC, suggesting that more progress is made in fewer iterations in our parallel implementation. Moreover, we see that overall, PAR-CC offers a better precision-recall trade-off compared to PAR-MOD. We also see the same behavior on dblp and livejournal, which we show in the appendix.

Figure 9 shows the average precision-recall curves for TECTONIC [38], considering $\theta \in \{0.01x \mid x \in [1, 299]\}$, which we compare to PAR-CC on amazon, dblp, livejournal, and orkut. We see that TECTONIC achieves similar precision-recall trade-offs on amazon, but PAR-CC obtains significantly better precision-recall on dblp, livejournal, and orkut. Notably, TECTONIC degrades significantly on the larger graphs livejournal and orkut compared to PAR-CC.

Weighted graphs. For weighted graphs, we use the Adjusted Rand-Index (ARI) and Normalized Mutual Information (NMI) scores, and precision and recall, to measure the quality of the clusters produced on the weighted k-NN graphs digits and letter, compared to ground truth. We defer the details to the appendix, but in summary, we see overall that taking into account edge weights leads to a higher quality clustering compared to treating these weighted graphs as unweighted, particularly for the CC objective. Furthermore, the PAR-CC$^{\text{W}}$ implementation offers higher quality over a wider range of resolutions, compared to other implementations, including NETWORKIT, where quality degrades sharply with certain resolutions. In this sense, PAR-CC$^{\text{W}}$ is more robust across different resolution parameters compared to other implementations.

5 CONCLUSION

We have designed and evaluated a comprehensive and scalable parallel clustering framework, which captures both correlation and modularity clustering. Our framework offers settings with trade-offs between performance and clustering quality. We obtained significant speedups over existing state-of-the-art implementations that scale to large datasets of up to billions of edges. Moreover, we showed that optimizing for correlation clustering objective results in higher-quality clusters with respect to ground truth data, compared to other methods used in highly-scalable clustering implementations. This shows the significance of the correlation clustering objective for community detection. Finally, we proved the P-completeness of the Louvain-like algorithms for parallel modularity and correlation clustering.

ACKNOWLEDGMENTS

This research was supported by NSF Graduate Research Fellowship #1122374. We thank Christian Sohler for insightful discussions on evaluating clustering quality.

REFERENCES

[1] Charu C. Aggarwal and Haixun Wang. 2010. *A Survey of Clustering Algorithms for Graph Data*. Springer US, Boston, MA, 275–301. https://doi.org/10.1007/978-1-4419-6045-0_9

[2] Nir Ailon, Moses Charikar, and Alantha Newman. 2008. Aggregating Inconsistent Information: Ranking and Clustering. *J. ACM* 55, 5, Article 23 (Nov. 2008), 27 pages. https://doi.org/10.1145/1411509.1411513

[3] Seung-Hee Bae, Daniel Halperin, Jevin D. West, Martin Rosvall, and Bill Howe. 2017. Scalable and Efficient Flow-Based Community Detection for Large-Scale Graph Analysis. *ACM Trans. Knowl. Discov. Data* 11, 3, Article 32 (March 2017), 30 pages. https://doi.org/10.1145/2992785

[4] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. 2004. Correlation Clustering. *Mach. Learn.* 56, 1–3 (June 2004), 89–113. https://doi.org/10.1023/B:MACH.0000033116.57574.95

[5] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. Brief Announcement: ParlayLib – A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[6] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (oct 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008

[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining (SDM)*. 442–446.

[8] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*, Akhil Arora, Semih Salihoglu, and Nikolay Yakovets (Eds.). ACM, 11:1–11:8. https://doi.org/10.1145/3398682.3399168

[9] T. N. Dinh, X. Li, and M. T. Thai. 2015. Network Clustering via Maximizing Modularity: Approximation Algorithms and Theoretical Limits. In *2015 IEEE International Conference on Data Mining*. 101–110. https://doi.org/10.1109/ICDM.2015.139

[10] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[11] Micha Elsner and Warren Schudy. 2009. Bounding and comparing methods for correlation clustering beyond ILP. In *Proceedings of the Workshop on Integer Linear Programming for Natural Langauge Processing*. 19–27.

[12] Mahmood Fazlali, Ehsan Moradi, and Hadi Tabatabaee Malazi. 2017. Adaptive parallel Louvain community detection on a multicore platform. *Microprocessors and Microsystems* 54 (2017), 26 – 34. https://doi.org/10.1016/j.micpro.2017.08.002

[13] Alexandre Fender, Nahid Emad, Serge Petiton, and Maxim Naumov. 2017. Parallel Modularity Clustering. *Procedia Computer Science* 108 (2017), 1793–1802. https://doi.org/10.1016/j.procs.2017.05.198 International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[14] Ullas Gargi, Wenjun Lu, Vahab Mirrokni, and Sangho Yoon. 2011. Large-scale community detection on youtube for topic discovery and exploration. In *in Proc. of the Fifth international AAAI Conference on Weblogs and Social Media*. 486–489.

[15] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin. 2018. Distributed Louvain Algorithm for Graph Community Detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 885–895. https://doi.org/10.1109/IPDPS.2018.00098

[16] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 698–710.

[17] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826. https://doi.org/10.1073/pnas.122653799 arXiv:https://www.pnas.org/content/99/12/7821.full.pdf

[18] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *International Conference on Machine Learning (ICML)*. 3887–3896.

[19] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. 2017. Scalable static and dynamic community detection using grappolo. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[20] A. K. Jain, M. N. Murty, and P. J. Flynn. 1999. Data Clustering: A Review. *ACM Comput. Surv.* 31, 3 (Sept. 1999), 264–323. https://doi.org/10.1145/331499.331504

[21] Lucas G. S. Jeub, Prakash Balachandran, Mason A. Porter, Peter J. Mucha, and Michael W. Mahoney. 2015. Think locally, act locally: Detection of small, medium-sized, and large communities in large networks. *Phys. Rev. E* 91 (Jan 2015), 012821. Issue 1. https://doi.org/10.1103/PhysRevE.91.012821

[22] Biaobin Jiang, Jiguang Wang, Jingfa Xiao, and Yong-Cui Wang. 2009. Gene Prioritization for Type 2 Diabetes in Tissue-specific Protein Interaction Networks. *Syst Biol* 11.

[23] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.

[24] Jure Leskovec and Andrej Krevl. 2019. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[25] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel Comput.* 47 (2015), 19 – 37. https://doi.org/10.1016/j.parco.2015.03.003 Graph analysis for scientific discovery.

[26] Naoto Ozaki, Hiroshi Tezuka, and Mary Inaba. 2016. A Simple Acceleration Method for the Louvain Algorithm. *International Journal of Computer and Electrical Engineering* 8 (01 2016), 207–218. https://doi.org/10.17706/IJCEE.2016.8.3.207-218

[27] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. 2015. Parallel Correlation Clustering on Big Graphs. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 82–90.

[28] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. 2014. High Quality, Scalable and Parallel Community Detection for Large Real Graphs. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14)*. Association for Computing Machinery, New York, NY, USA, 225–236. https://doi.org/10.1145/2566486.2568010

[29] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels. 2015. Scalable Community Detection with the Louvain Algorithm. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 28–37. https://doi.org/10.1109/IPDPS.2015.59

[30] Jörg Reichardt and Stefan Bornholdt. 2006. Statistical mechanics of community detection. *Phys. Rev. E* 74 (Jul 2006), 016110. Issue 1. https://doi.org/10.1103/PhysRevE.74.016110

[31] Jason Riedy, David A Bader, and Henning Meyerhenke. 2012. Scalable multi-threaded community detection in social networks. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1619–1628.

[32] Randolf Rotta and Andreas Noack. 2011. Multilevel Local Search Algorithms for Modularity Clustering. *ACM J. Exp. Algorithmics* 16, Article 2.3 (July 2011), 27 pages. https://doi.org/10.1145/1963190.1970376

[33] N. S. Sattar and S. Arifuzzaman. 2018. Parallelizing Louvain Algorithm: Distributed Memory Challenges. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. 695–701. https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00122

[34] 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

[35] C. L. Staudt and H. Meyerhenke. 2016. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 171–184. https://doi.org/10.1109/TPDS.2015.2390633

[36] V. A. Traag. 2015. Faster unfolding of communities: Speeding up the Louvain algorithm. *Phys. Rev. E* 92 (Sep 2015), 032801. Issue 3. https://doi.org/10.1103/PhysRevE.92.032801

[37] V. A. Traag, L. Waltman, and N. J. van Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports* 9, 1 (March 2019), 5233. https://doi.org/10.1038/s41598-019-41695-z

[38] Charalampos E. Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. 2017. Scalable Motif-Aware Graph Clustering. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1451–1460. https://doi.org/10.1145/3038912.3052653

[39] Nate Veldt, David F. Gleich, and Anthony Wirth. 2018. *A Correlation Clustering Framework for Community Detection*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 439–448. https://doi.org/10.1145/3178876.3186110

[40] Wayne W. Zachary. 1977. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research* 33, 4 (1977), 452–473. http://www.jstor.org/stable/3629752

[41] J. Zeng and H. Yu. 2015. Parallel Modularity-Based Community Detection on Large-Scale Graphs. In *2015 IEEE International Conference on Cluster Computing*. 1–10. https://doi.org/10.1109/CLUSTER.2015.11

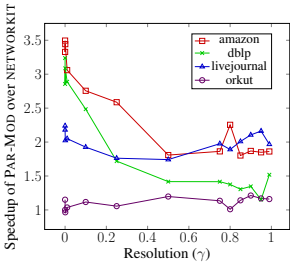

Figure 10: Speedup of PAR-MOD over NETWORKIT on amazon, dblp, livejournal, and orkut, for varying resolutions.

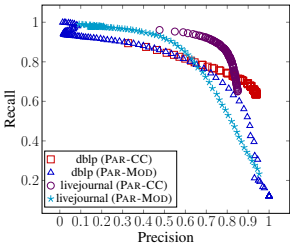

Figure 11: Average precision and recall compared to ground truth communities on dblp and livejournal of the clusters obtained from our parallel implementations PAR-CC and PAR-MOD, using varying resolutions.

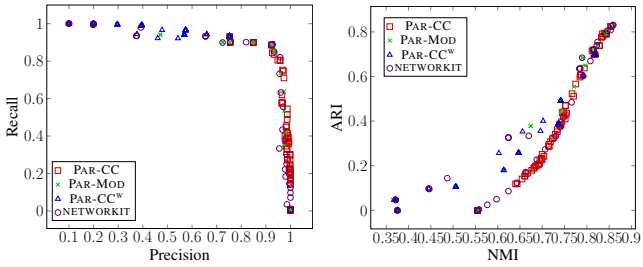

Figure 12: Average precision-recall and ARI-NMI scores for the digits graph, from PAR-CC, PAR-MOD, PAR-CC$^{\text{W}}$, and NETWORKIT.

A ADDITIONAL EXPERIMENTS

We show in Figure 10 the speedups of PAR-MOD over NETWORKIT on amazon, dblp, livejournal, and orkut, for varying resolutions.

We show in Figure 11 the average precision and recall compared to ground truth communities on dblp and lj, of the clusters obtained by PAR-CC and PAR-MOD for varying resolutions. As discussed in Section 4.3, we see that overall, PAR-CC offers a better precision-recall trade-off compared to PAR-MOD.

We also include the details of our experiments on weighted graphs here. The sequential implementations SEQ-CC and SEQ-MOD give similar results to the corresponding parallel implementations, so we discuss only the parallel implementations. Figures 12 and 13 show the average precision-recall and ARI-NMI scores for the digits and letter graph respectively. We compare our parallel implementations to NETWORKIT's modularity clustering implementation, which can also take as input weighted graphs [35]. NETWORKIT matches our PAR-MOD$^{\text{W}}$ implementation, so we omit the latter from our figures. We consider our implementations both treating the graphs as unweighted, and taking into account the edge weights. Moreover, we consider a range of resolutions, with $\lambda \in \{0.01x \mid x \in [1, 99]\}$ for the CC objective, and $\gamma \in \{0.02 \cdot (1.2)^x \mid x \in [1, 99]\}$ for the modularity objective.

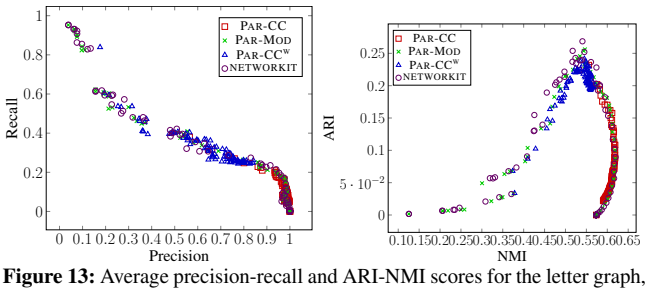

Figure 13: Average precision-recall and ARI-NMI scores for the letter graph, from PAR-CC, PAR-MOD, PAR-CC$^{\text{W}}$, and NETWORKIT.

B P-COMPLETENESS OF LOUVAIN

We prove here that the problem of obtaining the clustering given by the Louvain method maximizing for the CC objective is P-complete.

THEOREM 3.1. *The problem of obtaining a clustering equivalent to that given by the Louvain method maximizing for the* CC *objective is P-complete.*

PROOF. We set $\lambda = 0$ for the purposes of this proof. We will show that there is an NC reduction from the monotone circuit-value problem (CVP), where given an input circuit C of $\wedge$ and $\vee$ gates on n variables $(x_1, \ldots, x_n)$ and their negations, and an assignment of truth to these variables, the problem is to compute the output value of C.

The reduction works as follows. We use a fixed small $\varepsilon > 0$, and we construct a graph G. Initially, the vertices of G are the literals $(x_1, \ldots, x_n)$ and their negations, and two additional vertices t and f (representing true and false respectively). We assign an edge with a large enough constant negative weight between t and f. We also assign an edge with a large enough constant positive weight between each literal and its corresponding t or f, depending on if the literal is true or false respectively.

Then, for every gate, say $g_i \vee g_j$ which outputs to g_k, we add a corresponding vertex g_k and g'_k in G. We use a weight w_{ijk}, which we define later. We also add edges of weight w_{ijk} between (g_i, g_k) and (g_j, g_k), and an edge of weight $(2 + \frac{2}{3} \cdot \varepsilon)w_{ijk}$ between (g_k, g'_k). Finally, we add an edge of weight $(1 + \varepsilon)w_{ijk}$ between (g_k, t), and an edge of weight $(1 + \frac{1}{2} \cdot \varepsilon)w_{ijk}$ between (g_k, f). Figure 14 shows these vertices and edges.

We then define w_{ijk} as follows, for each gate $g_i \vee g_j$. We take the topological sort of the directed acyclic graph given by the circuit C, where the vertices correspond to the gates and the literals.

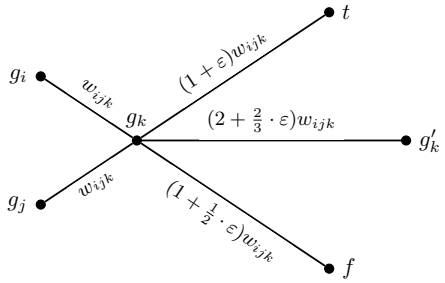

Figure 14: The vertices and edges added to the graph G, given a gate $g_i \vee g_j$ which outputs to g_k.

We call the ordered vertices of this DAG given by the topological sort $(c_1, \ldots, c_n)$, in order, and we let $c(g)$ denote the c_i that corresponds to the gate or literal g. Then, we define a function $f(c_i) = 1 \Big/ \prod_{1 \leq j < i} d_{c_j}$, where d_{c_j} denotes the degree of c_j. We note that $f(\cdot)$ can be efficiently computed for each c_i by taking a prefix product of the degrees of c_i, in order. Finally, given a gate $g_i \vee g_j$ which outputs to g_k, we set $w_{ijk} = \min(f(c(g_i)), f(c(g_j)))$.

Note that the construction for every $\wedge$ gate is performed similarly, except we swap the edge weights between (g_k, t) and (g_k, f). More explicitly, we instead add an edge of weight $(1 + \frac{1}{2} \cdot \varepsilon) w_{ijk}$ between (g_k, t), and an edge of weight $(1 + \varepsilon) w_{ijk}$ between (g_k, f).

We now claim that applying the Louvain method optimizing for the CC objective on this graph G solves the circuit C. In other words, we claim that each gate g_i will ultimately cluster with either t or f, depending on if its value in the circuit is true or false respectively. Recall that the Louvain method involves two main subroutines that iterate until convergence – a subroutine in which vertices move to their best local clusters, and a subroutine that compresses the graph. We actually prove that the clustering given by performing best local vertex moves until convergence on G produces two final clusters of this nature, one containing t and one containing f. Then, in the compression subroutine, we obtain two vertices in the compressed graph corresponding to t and f, and because the edge weight between t and f is a large enough negative constant, the two vertices in the compressed graph will not cluster with each other, terminating the algorithm.

We begin by proving a weaker statement, namely that at any given point in time throughout the best local vertex moves process, each gate g_i is clustered into either a) a singleton cluster containing only g_i, b) a two-vertex cluster containing g_i and g_i', or c) a cluster containing g_i and either t or f (but not both), depending on g_i's corresponding truth value.

We prove this statement using induction. The base case follows because we begin with singleton clusters. We also note that the literals x_i and their negations will always choose to cluster with their corresponding t or f, because of the large enough positive constant weight between the edge from x_i to its corresponding truth value. Similarly, the vertices t and f will always choose to cluster with corresponding x_i, due to the large enough positive constant weight. As such, we disregard literals and the vertices t and f in our inductive step. For our inductive step, we assume the inductive hypothesis, and show that the statement holds when we consider the local best move of a vertex g_k, originating from a gate $g_i \vee g_j$ (the argument for a gate $g_i \wedge g_j$ follows symmetrically).

Note that g_k' only has a single positive weight edge to g_k, so its local best move is always to move to the cluster that g_k is in. If neither g_i nor g_j are clustered with t or f when g_k decides its best move, then g_k will always choose to move to the cluster that g_k' is in. This follows by construction because the edge weight (g_k, g_k') of $(2 + \frac{2}{3} \cdot \varepsilon) w_{ijk}$ exceeds the edge weights from g_k to g_i. g_j, t, and f, and because the sum of the weights of the edges from g_k to its out-neighbors must be less than w_{ijk}. Moreover, since g_k' only ever decides to cluster with g_k, either g_k will remain in its current cluster, which satisfies the inductive step, or g_k will move to cluster with g_k', which must be a singleton cluster. In this case, g_k forms a two-vertex cluster with g_k'.

If at least one of g_i and g_j is clustered with t when g_k decides its best move, WLOG g_i, then g_k will always choose to move to cluster with t, which by the inductive hypothesis must correspond with g_k's truth value. This is because the sum of the weight of the edges (g_k, t) and (g_k, g_i) is given by $(2 + \varepsilon) w_{ijk}$, which exceeds the weight of the edge (g_k, g_k'), or $(2 + \frac{2}{3} \cdot \varepsilon) w_{ijk}$, thus inducing g_k to move if it was originally in either a singleton cluster or a two-vertex cluster with g_k'. As before, the sum of the weights of the edges from g_k to its out-neighbors must be less than w_{ijk}, so g_k will not consider any other cluster.

Similarly, if both of g_i and g_j are clustered with f when g_k decides its best move, then g_k will always choose to move to cluster with f, which by the inductive hypothesis must correspond with g_k's truth value. This again follows directly from the weights of the edges from g_k to g_i, g_j, t, f, and g_k'. Moreover, if exactly one of g_i and g_j is clustered with f, and neither are clustered with t, then g_k will always choose to move to the cluster that g_k' is in, by virtue of the constructed edge weights. In both of these cases, the inductive step is ultimately satisfied.

Thus, we have shown that at any given point in time throughout the best local vertex moves process, each gate g_i is clustered into either a) a singleton cluster containing only g_i, b) a two-vertex cluster containing g_i and g_i', or c) a cluster containing g_i and either t or f (but not both), depending on g_i's corresponding truth value.

To complete the proof, we must now show that the best moves process converges with each vertex clustered with either t or f, depending on its corresponding truth value. This follows from a similar argument to our inductive argument above, where for a gate $g_i \vee g_j$ that outputs to g_k, if g_i and g_k are both clustered with their corresponding truth value, then g_k will always choose to cluster with its corresponding truth value during its best move operation. Furthermore, the literals x_i and their negations necessarily choose to cluster with their corresponding truth values whenever prompted, by construction of the edge weights between the literals and t and f. Thus, it follows that the best moves process converges with each gate g_i clustered with either t or f.

This completes the reduction, since we can obtain from the final clusters the solution to the circuit C.

□