Scalable Community Detection via Parallel Correlation Clustering

Jessica Shi
MIT
jeshi@mit.edu

Laxman Dhulipala
MIT
laxman@mit.edu

David Eisenstat
Google
eisen@google.com

Jakub Łącki
Google
jlacki@google.com

Vahab Mirrokni
Google
mirrokni@google.com

ABSTRACT

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

We develop a generalized sequential and shared-memory parallel framework based on the LambdaCC objective (introduced by Veldt et al.), which encompasses modularity and correlation clustering. Our framework consists of highly-optimized implementations that scale to large data sets of 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.

1 INTRODUCTION

As a fundamental tool in modern data mining, graph clustering, or community detection, has a wide range of applications spanning data mining [23], social network analysis [16], bioinformatics [24], and machine learning [22], and has been well-studied under many frameworks [1, 38]. 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 major 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.* [43], which provides a general objective encompassing modularity [19] and correlation clustering [4]. Veldt *et al.* show 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 within a constant factor [10], so optimizing for modularity, and by extension optimizing for the LambdaCC objective, is inherently difficult. 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 [27, 39–41].

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 state-of-the-art empirical 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 proof-of-concept implementation of Veldt *et al.* [43]. We note that for both LabmdaCC 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 [42] and SCD [30] 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 shared-memory modularity clustering implementation in NetworKit [39].

Our code is available at https://github.com/jeshi96/parallel-correlation-clustering.

Further related work. Optimization for correlation clustering has been studied empirically in the case of complete graphs, which is equivalent to LAMBDACC objective with resolution $\gamma = 0.5$ [12, 29]. In this restricted setting, several scalable parallel implementations have been obtained based on the KWIKCLUSTER algorithm [8, 15, 29]. We observe that the KWIKCLUSTER algorithm typically obtains a negative LAMBDACC objective, which significantly limits its practical applicability.

Scalable modularity clustering has been extensively studied both in the shared-memory [13, 14, 21, 27, 39, 45] and distributed memory [17, 31, 34, 37] settings. The two fastest implementations that we identify are NetworKit [39] and Grappolo [17, 21]. Both of them offer comparable performance, but we observed the NetworKit typically computes solutions with slightly larger objective, and thus we compare 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

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

We use a generalized correlation clustering objective that is equivalent to the LAMBDACC objective given by Veldt *et al.* [43]. 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 $w'_{uv} = 0$ if $u = v$, $w'_{uv} = w_{uv} - \lambda k_u k_v$ if $(u, v) \in E$, and $w'_{uv} = -\lambda k_u k_v$ otherwise.

The goal is to maximize the 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 k and setting λ appropriately. Note that Reichardt and Bornholdt [33] 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 [19]. 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 is then equivalent.

3 ALGORITHM AND OPTIMIZATIONS

3.1 Sequential Louvain Method

We begin by describing the classic sequential Louvain method from Blondel *et al.* [6], SEQUENTIAL-CC, adapted for the correlation clustering objective. We include the pseudocode in the appendix. The main idea is to repeatedly move vertices to clusters that would maximize the objective, and once no vertices can be moved, compress clusters into vertices and repeat this process on the compressed graph. In more detail, the algorithm takes as input a graph G and node weights k, and begins with singleton clusters. Then, it iterates over each vertex in a random order, and locally moves vertices to clusters that maximize the CC objective. The computation to determine the cluster that vertex v should move to can be performed by maintaining the total vertex weight K_c of each cluster c in C. We provide the formula in the appendix.

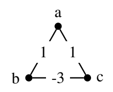

Figure 1: An example where parallel local vertex moves lowers the total objective. Assume $\lambda = 0$ and initial clusters are singletons with objective 0. If b and c are both scheduled to move at the same time, they each choose cluster $\{a\}$, leading to a single cluster $\{a, b, c\}$ with objective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

After all vertices have been moved, the algorithm repeats this step of locally moving vertices until no vertices have performed non-trivial moves. If no vertices changed clusters during this phase, then SEQUENTIAL-CC terminates. Otherwise, once a stable state has been achieved, the algorithm compresses the graph G (using a subroutine SEQUENTIAL-COMPRESS) by creating a new graph G' with vertex weights k'. Each cluster c in G corresponds to a vertex in G' with vertex weight $k'(c) = K_c$. 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 (using a subroutine SEQUENTIAL-FLATTEN). 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.

The main bottleneck in parallelizing SEQUENTIAL-CC is the sequential dependencies in moving each vertex to the cluster that maximizes the objective, and we prove a related P-completeness result about this bottleneck in the appendix (showing that the problem of obtaining a clustering equivalent to any clustering C given by moving each vertex to its best cluster, is inherently sequential to solve under standard complexity-theory assumptions).

As such, to obtain an empirically efficient implementation that achieves good parallelism, we heuristically relax the sequential dependency and allow vertices to move to clusters *concurrently*. While vertices move to clusters that would individually maximize the objective, these moves in tandem may give a lower total objective; there is no guarantee of convergence. We show an example of this in Figure 1. 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 [39]. We now discuss optimizations that can be used with this relaxation.

Algorithm 1 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 $c \in C$
3:      $D \leftarrow$ array of size $n$
4:      $V' \leftarrow V$
5:      for $i$ in range($num\_iter$) do
6:          parfor $v \in V'$ do
7:              $D[v] \leftarrow$ id($c$) such that moving $v$ to $c \in C$ would maximize CC($C$)
8:              Move $v$ to $D[v]$
9:          if no moves were made in iteration $i$ then break
10:         $V' \leftarrow \{$ neighbors of $v \mid v$ moved$\}$
11:     return $C$

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

3.2 Parallel Louvain Method and Optimizations

Algorithm 1 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 best 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.

Our parallelization uses a natural heuristic relaxation of the sequential dependency in moving vertices to their desired clusters. A more faithful parallelization would fix a random permutation of V, and move in parallel the first ℓ vertices in order for the largest ℓ such that moving these ℓ vertices would not affect each other's objectives. However, compared to a heuristic relaxation, not only does this involve greater overhead due to the prefix computation of vertices that do not conflict, but it also respects sequential dependencies that may not affect later vertex moves. Thus, in the interest of performance, we consider the parallelization given in Algorithm 1.

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.

Our main optimizations introduce symmetry breaking and work reduction techniques to improve performance while maintaining the objective. We also discuss a refinement step that improves the objective at the cost of running time and a higher memory overhead.

3.2.1 Optimization: Synchronous vs Asynchronous. The first optimization involves scheduling individual vertex moves in BEST-MOVES, on Line 8. We explore two options: *synchronous* and *asynchronous*. These options have been previously studied in parallelizing the Louvain method for the modularity objective [37, 39].

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 in parallel by aggregating vertices that move from the same clusters and vertices that move 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 perhaps surprisingly, the asynchronous setting outperforms the synchronous setting, particularly in terms of objective. Of the optimizations, the asynchronous optimization contributes most significantly towards an improvement in objective. This is because the asynchronous setting allows for symmetry breaking, whereas in the synchronous setting, vertices that are attempting to move away from each other may inadvertently move to the same cluster, since they must move in lockstep. For certain graphs, this symmetry breaking also allows the asynchronous setting to outperform the synchronous setting in terms of running time, due to fewer vertex moves required to obtain the maximal objective.

Previous uses of the Louvain method for other objectives explored different schedules for vertex moves, which give more granular trade-offs [3, 27]. We found that our asynchronous setting outperforms methods that maintain consistency guarantees in quality and speed.

3.2.2 Optimization: All Vertices vs Neighbors of Clusters vs Neighbors of Vertices. We now consider optimizations that reduce the set of vertices to consider moving in every iteration of BEST-MOVES. 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 has been previously used in work on the Louvain method for the modularity objective [3, 28]. Importantly, the BEST-MOVES subroutine takes a significant portion of total clustering time, and reducing the subset of vertices to consider offers significant performance improvements.

In more detail, isolating a vertex v which has in the 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 , and (c) vertices in c'. 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. For conciseness in describing category (b), we formally define *neighbors of clusters C* to be the union of the neighbors of each vertex in each cluster in C.

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. This is because the vertices that are most affected by moving vertices in terms of objective are neighbors of the moving vertices, and thus most of the objective obtained is from considering these neighbors. While there may be contrived scenarios in which more objective is affected due to non-neighbors of moving vertices with sufficiently large edge weights [1], we do not see these scenarios in practice, and considering a smaller subset of vertices in each iteration allows for less total work to be performed, since we save on the cost of computing best moves for other vertices. The performance improvements outweigh the marginal loss in objective from these cases.

3.2.3 Optimization: Multi-level Refinement. Finally, we consider a popular multi-level refinement optimization [35, 39]. 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; vertices in the coarsened graph correspond to clusters, or sets of vertices, in the original graph. For instance, each vertex v in the original graph is clustered into a cluster C, which corresponds to the vertex v' in the coarsened graph. We then recurse on the coarsened graph. Following the recursion, we receive a clustering on the coarsened graph, which we must translate to a clustering on the original graph. Importantly, each vertex v' in the coarsened graph now belongs to a cluster C' in the coarsened graph, and we must now assign the cluster of the original vertex v. We use a flattening procedure for this, where we simply assign each vertex v in the original graph to the corresponding cluster C'.

However, note that v did not have an opportunity to move clusters individually in successive recursive steps, and importantly, because there is no guarantee of convergence, clusters may not reach a steady state prior to compressing the graph. v may have ended up in a suboptimal cluster C when the coarsening was performed and would have been unable to change clusters after the coarsening. Now, given its new cluster C', v may desire to change clusters. The *multi-level refinement* optimization allows for v to move by performing a refinement step after each flattening step, as we traverse back up the recursive hierarchy. We simply perform a further iteration of BEST-MOVES on each individual vertex v, prior to returning the clustering.

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. This is precisely due to the lack of guarantee of convergence in Algorithm 1, where vertices may be coarsened non-optimally. The refinement step allows for these vertices to move to better clusters as we traverse back up the recursive hierarchy, resulting in better overall quality.

3.2.4 Other Optimizations. We discuss in the appendix other practical optimizations that we use, including our efficient parallelization of the subroutines SEQUENTIAL-COMPRESS and SEQUENTIAL-FLATTEN.

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.

We show that optimizations that address symmetry breaking and work reduction result in an overall faster implementation while maintaining objective, and additional refinement steps improve the objective at the cost of performance and memory usage. We also demonstrate significant speedups over state-of-the-art implementations due to our theoretically efficient parallelization of key subroutines, while obtaining high quality compared to ground truth.

Environment. We run most experiments on a c2-standard-60 Google Cloud instance, with 30 cores (with two-way hyper-threading), 3.8GHz Intel Xeon Scalable processors, and 240 GiB main memory. For experiments on large graphs we use a m1-megamem-96 Google Cloud instance, with 48 cores (with two-way hyper-threading), 2.7GHz Intel Xeon Scalable processors, and 1434 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, which, as shown in [5], provides on average a 1.43x speedup over Intel's Parallel STL library. 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) [26], 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 [25]. Details of these graphs are shown in the appendix. To show the clustering quality of our implementations, we compare with the top 5000 ground-truth communities given by SNAP. These communities may overlap, so 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.[2] This matches the methodology used by Tsourakakis *et al.* in evaluating TECTONIC [42].

We also use an approximate k-NN algorithm [20] to construct weighted graphs from pointset data, from the UCI Machine Learning repository [11]. We defer a discussion of our weighted graph data to the appendix. Additionally, we demonstrate scalability using synthetic graphs generated by the standard rMAT graph generator [7], with $a = 0.5$, $b = c = 0.1$, and $d = 0.3$.

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

[1] For instance, given a large enough star graph where each leaf has a small enough positive edge weight to the center, following the first set of moves in which every leaf clusters with the center.

[2] Any given cluster c' may be matched to multiple or no ground-truth communities c.

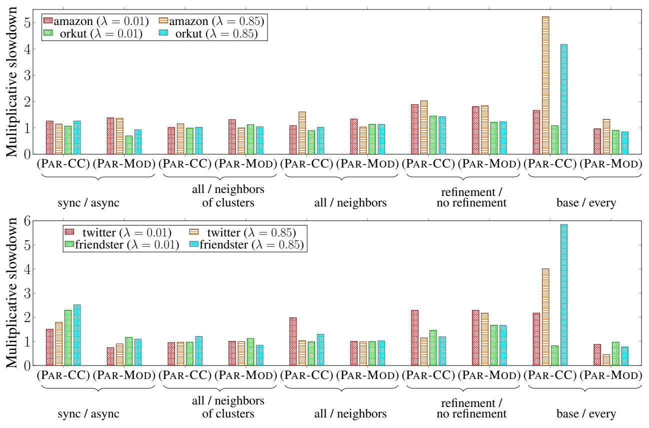

Figure 2: Multiplicative slowdown in average time of each optimization. Also shown is the slowdown for no optimizations (base) over every optimization. Running times were obtained for Par-CC and Par-Mod on the graphs amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$.

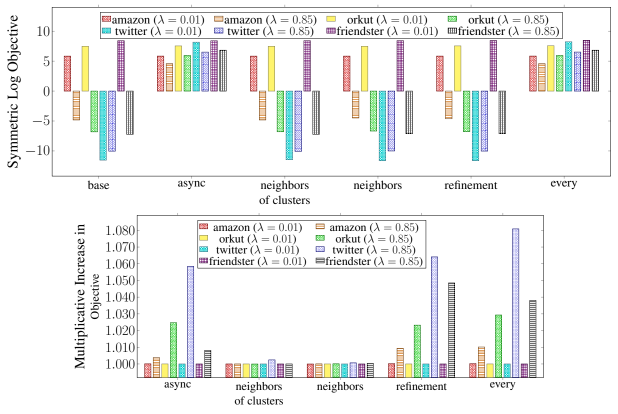

Figure 3: CC objective for Par-CC on a symmetric log scale[4] (top) and multiplicative increase in the modularity for Par-Mod over no optimizations (bottom), of each optimization and every optimization. Objectives were obtained on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$.

and λ as discussed in Section 2 to obtain modularity clustering implementations (Seq-Mod and Par-Mod). 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 to convergence (without restricting the number of iterations), and we use no superscript if we use $num\_iter = 10$. We run each experiment 10 times and report the average time and objective. [3]

We compare to two correlation clustering implementations, namely the parallel C4 and ClusterWild! by Pan *et al.* [29] (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.* [43].

We note that there is a rich body of work on pivot-based correlation clustering algorithms, both parallel and sequential, including prior work by Chierichetti *et al.* [8] and by García-Soriano *et al.* [15]. These works are based on KwikCluster (also known as Pivot), and offer faster performance with matching or worse approximation guarantees. However, in our comparison to C4, which parallelizes KwikCluster and which matches KwikCluster's approximation guarantee, we note that while C4 is much faster than Par-CC, the quality is poor compared to Par-CC in terms of both the CC objective and comparison to ground-truth communities, resulting in clusters of vertices with proportionally lower similarities to each other. Thus, we omit pivot-based work that offer the same or worse quality guarantees compared to C4.

We also compare to two state-of-the-art community detection algorithms which are based on triangle counts: the sequential Tectonic by Tsourakakis *et al.* [42], and the shared-memory parallel SCD, by Prat-Pérez *et al.* [30]. Both algorithms were shown to deliver superior quality to multiple similarly scalable baseline methods. 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 [39].

[3]The average objective is non-deterministic when using the asynchronous setting from Section 3.2.1.

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

We tuned these optimizations on the graphs amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. Note that lower resolutions produce a clustering with fewer clusters, and higher resolutions produce a clustering with more clusters; these resolutions effectively model both scenarios, where differences in the number of clusters produced may affect performance. We fix the synchronous, all vertex moves, and no refinement options, and give running times and objectives considering turning on a single optimization at a time; these are the natural settings that do not optimize the basic sequential Louvain method. Considering both Par-CC and Par-Mod, Figure 2 shows the multiplicative slowdowns of synchronous over asynchronous, all vertices over neighbors of clusters, all vertices over neighbors of vertices, multi-level refinement over no refinement (note that multi-level refinement causes slowdowns, but improves quality over the basic no refinement option), and no optimizations over every optimization. Figure 3 shows the objectives for these optimizations.

We see speedups of up to 2.50x, with a median of 1.21x, using the asynchronous over the synchronous setting across Par-CC and Par-Mod. For Par-CC, the synchronous setting often produces negative objective, whereas in the asynchronous setting, the objective is always positive, and we see a 1.29–156.01% increase in objective. The objective in the synchronous setting is negative likely due to the

[4]We take the symmetric log of x to be $\text{sign}(x) \cdot \log |x|$. We use a symmetric log scale to more accurately depict the CC objectives, because the objectives are very large positive and negative numbers.

phenomenon shown in Figure 1, which is more likely to appear for large resolutions due to the objective computation. This phenomenon has additionally been discussed in prior work in relation to the modularity objective [32, 39]. For PAR-MOD, we see a 0.0015 – 5.84% increase in modularity using the asynchronous setting over the synchronous setting. The synchronous setting often leads to very poor objectives due to a lack of symmetry breaking, compared to the asynchronous setting where there is inherent randomness.

The asynchronous setting is also often faster than the synchronous setting, because due to the symmetry breaking, fewer vertices end up making moves that decrease the objective. For PAR-MOD on orkut and twitter, the asynchronous setting is not faster than the synchronous setting, but the increase in modularity using the asynchronous setting is more significant, compared to that obtained on other graphs. Up to 1.43x more time is spent computing best moves in the asynchronous setting compared to the synchronous setting, due to an increased number of vertex moves required to obtain the higher objective. Overall, considering tradeoffs between objective and speed, the best setting in general is the asynchronous setting.

We see up to a 1.98x speedup, with a median of 1.03x, considering neighbors of vertices compared to all vertices as the subset V', and we see up to a 1.32x speedup, with a median of 1.01x, considering neighbors of clusters compared to all vertices. Moreover, the objectives obtained in all settings are comparable. This is because neighbors of previously moved vertices, and by extension neighbors of clusters of previously moved vertices, are most significantly affected in terms of objective by previously moved vertices, based on the change in objective formula. In the cases where the speedup is minimal, vertices in these classes represent a larger proportion of all vertices, which we show in the appendix; however, due to the cases with more significant speedups, the best setting in general is considering neighbors of vertices as the subset V'.

Finally, we see slowdowns of up to 2.29x, with a median of 1.67x, using multi-level refinement, compared to using no refinement. However, using multi-level refinement, we see 1.12 – 36.92% increase in the CC objective, and up to a 6.41% increase in modularity. Refinement improves objective because it allows vertices to move to better clusters following the compression steps, increasing the objective in situations where compression was not optimal. The increase in time is due to the added work in refinement, and in general, the best setting is to use refinement. Note that these results mirror prior work applying multi-level refinement for the modularity objective [35, 39]. For the modularity objective using small resolutions, the increase in objective is minimal; this is because in these cases, the objective obtained without using refinement is already very high (on average 0.99, where the maximum is 1.00).

In the remaining experiments, we fix the asynchronous setting, using neighbors of vertices, and using multi-level refinement, as the overall optimal settings, although we note that for small resolutions, multi-level refinement often offers little increase in objective considering the increase in running time. Overall, using all optimizations, we see up to a 5.85x speedup in running time and up to a 156.01% increase in objective. For PAR-MOD, there are scenarios with up to a 2.20x slowdown in running time due to contention in the asynchronous setting compared to the synchronous setting, but in these cases, we see significant increases in modularity, of 2.93% – 8.09%.

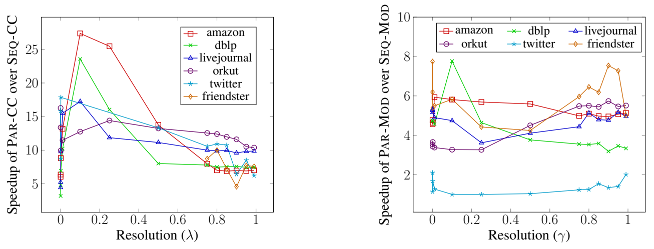

Figure 4: Speedup of PAR-CC over SEQ-CC (left) and of PAR-MOD over SEQ-MOD (right), on amazon, dblp, livejournal, orkut, twitter, and friendster, for varying resolutions. SEQ-CC timed out on twitter for $\lambda = 0.01$, 0.1, and 0.25, and on friendster for $\lambda < 0.75$.

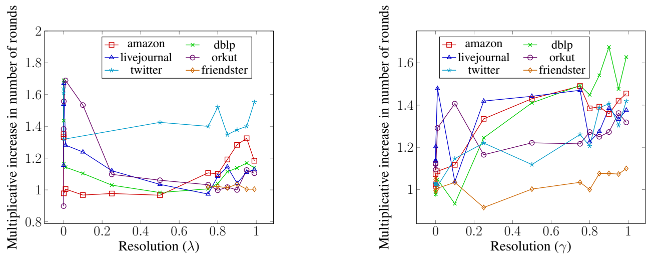

Figure 5: Multiplicative increase in the number of rounds until convergence or until the default number of iterations is reached, of PAR-CC over SEQ-CC (left) and of PAR-MOD over SEQ-MOD (right), on amazon, dblp, livejournal, orkut, twitter, and friendster, for varying resolutions. SEQ-CC timed out on twitter for $\lambda = 0.01$, 0.1, and 0.25, and on friendster for $\lambda < 0.75$.

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! [29], which are based on a maximal independent set algorithm, and the sequential Louvain-based method in LAMBDACC [43]. Our implementations significantly outperform these baselines, and we present a detailed comparison in the appendix. Notably, C4 and CLUSTERWILD! offer significant speedups of up to 428.64x over PAR-CC, but achieve poor and often negative objective, with a decrease in the objective of 273.35 – 433.31% over PAR-CC. C4 and CLUSTERWILD! also achieve poor precision and recall compared to ground truth communities, with precision between 0.44 – 0.65 and corresponding recall between 0.10 – 0.15. In comparison, on the same graphs, PAR-CC achieves recall between 0.61 – 0.98 for precision greater than 0.50. Furthermore, LAMBDACC is a MATLAB implementation that uses an adjacency matrix to represent the graph, and cannot scale to graphs of more than hundreds of vertices. Thus, 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.

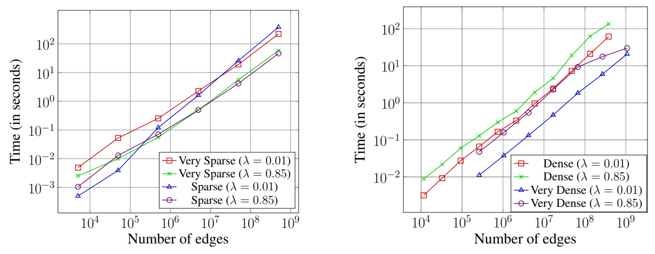

Figure 6: Scalability of PAR-CC over rMAT graphs of varying sizes.

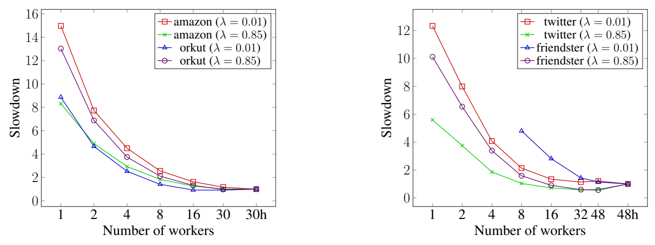

Figure 7: Scalability of PAR-CC over different numbers of threads, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. 30h and 48h indicate 30 and 48 cores respectively, with two-way hyper-threading.

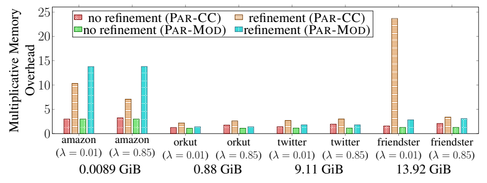

Figure 8: Multiplicative memory overhead, over the size of the input graph, of PAR-CC and PAR-MOD, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. The labels below denote the size of the input graph.

Figure 4 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 is significantly worse than that of SEQ-CC$^{\text{CON}}$, while our PAR-CC matches the average precision-recall of SEQ-CC$^{\text{CON}}$.

On the graphs amazon, dblp, livejournal, and orkut, and over varying resolutions, we see 3.19–27.38x speedups of PAR-CC over SEQ-CC, and 12.55–110.25x 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 3.18–7.76x speedups over SEQ-MOD, and 2.64–7.89x speedups over SEQ-MOD$^{\text{CON}}$, while achieving 1.00–1.06x the modularity of the serial implementations.

On the large graphs twitter and friendster, and over varying resolutions, we see 4.57–17.87x speedups of our PAR-CC over SEQ-CC, and up to 7.74x speedups of PAR-MOD over SEQ-MOD. We achieve between 0.95–1.00x the CC objective of SEQ-CC, and between 0.96–1.02x the modularity of SEQ-MOD. SEQ-CC timed out on twitter for $\lambda = 0.01$, 0.1, and 0.25, and on friendster for $\lambda < 0.75$. We observe lower speedups using PAR-MOD on twitter, which we discuss in the appendix.

Figure 5 shows the multiplicative increase in the total number of iterations required for PAR-CC and PAR-MOD over SEQ-CC and SEQ-MOD, which approximately displays the inverse of the behavior seen in the speedups shown in Figure 4 across different resolutions. We see greater speedups for resolutions where the number of iterations required in our parallel implementations match or are lower than the number of iterations required in our serial implementations. Whenever a greater number of iterations is required in parallel compared to serial, we observed lower speedups, simply due to the increased amount of work carried out by the parallel version.

Scalability. Figure 6 demonstrates the scalability of PAR-CC over rMAT graphs of varying sizes, with very sparse graphs ($m = 5n$), sparse graphs ($m = 50n$), dense graphs ($m = n^{1.5}$), and very dense graphs ($m = n^2$). We also show similar results for PAR-MOD in the appendix. We see that for both of our algorithms and across different resolutions ($\lambda = 0.01, 0.85$), the running times of our algorithms scale nearly linearly with the number of edges. Figure 7 shows the speedups of PAR-CC on amazon, orkut, twitter, and friendster, over different numbers of threads. We also show the speedups of PAR-MOD on the same graphs is given in the appendix. Overall, we see good parallel scalability, with 5.59–14.97x self-relative speedups on PAR-CC, and 1.89–14.51x self-relative speedups on PAR-MOD. Note that using fewer threads for PAR-CC times out on friendster for $\lambda = 0.01$, and we again see lower speedups for PAR-MOD on twitter, where there is increased contention in using atomic compare-and-swaps due to the very few clusters produced relative to the size of the graph. Excluding twitter, we see 5.29–14.51x self-relative speedups on PAR-MOD.

Memory Usage. Figure 8 shows the memory usage of PAR-CC and PAR-MOD on amazon, orkut, twitter, and friendster. [5] Theoretically, our memory usage is linear in the size of the input graph. We incur more memory when using multi-level refinement, particularly if more coarsening rounds are required. This is because with refinement, we store the intermediate coarsened graph from each recursive step, whereas without refinement, we discard these graphs. For instance, we see that for PAR-CC on friendster with $\lambda = 0.01$, four coarsening rounds are used, compared to $\lambda = 0.85$, where only one coarsening round is used, hence the difference in memory overhead. Overall, using refinement, we incur a 1.40–23.68x memory overhead over the size of the input graph, whereas without refinement, we incur a 1.25–3.24x overhead.

Comparisons to Other Implementations. In the special case of modularity, we compare against the highly optimized parallel modularity clustering implementation NETWORKIT [39]. NETWORKIT, like PAR-MOD, implements an asynchronous version of Louvain-based modularity clustering. We discuss in the appendix the speedups of PAR-MOD over NETWORKIT. We see up to 3.50x speedups (1.89x on average), primarily due to our optimization of the graph compression step, which we discuss below. We also obtain between

[5]The size of the input graph provided in Figure 8 is the total size in CSR format [36], which uses approximately 8 bytes per undirected edge.

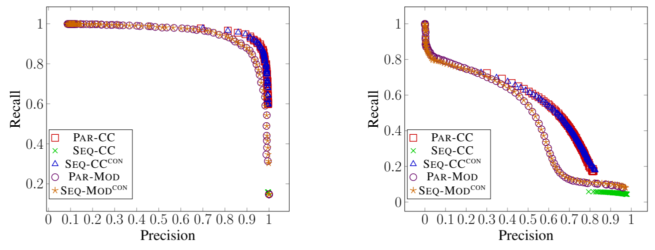

Figure 9: Average precision and recall compared to ground truth communities on amazon (left) and orkut (right) of the clusters obtained from PAR-CC and PAR-MOD, compared to SEQ-CC and SEQ-MOD, using varying resolutions.

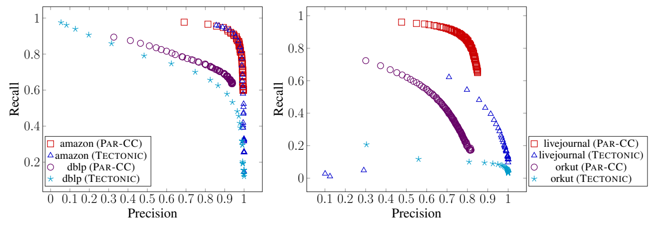

Figure 10: Average precision and recall compared to ground truth communities on amazon and dblp (left), and on livejournal and orkut (right), of the clusters obtained from PAR-CC using varying resolutions, and from TECTONIC using varying θ.

0.99 – 1.00x the modularity given by NETWORKIT's implementation, where some variance appears due to the asynchronous nature of both implementations.

Our speedups over NETWORKIT are primarily because NETWORKIT does not efficiently parallelize the graph compression step between rounds of best vertex moves. Our implementations use a work-efficient algorithm to parallelize this step, where intra-cluster edges are aggregated in polylogarithmic depth with an efficient parallel sort, whereas no such guarantee is made in NETWORKIT.

Additionally, we compare to the sequential TECTONIC implementation [42], which clusters based on the idea of triangle conductance and provides 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 [30], a parallel triangle-based community detection implementation. PAR-CC gives up to 2.89x speedups over SCD for resolutions that give comparable or better quality than SCD, in terms of average precision and recall. We discuss further details in the appendix.

4.3 Quality Compared to Ground Truth

Figure 9 shows the average precision-recall curves obtained by PAR-CC and PAR-MOD, varying resolutions, compared to the top 5000 ground truth communities on 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 limiting the number of iterations displays precisely the same average precision-recall curve as the version running to convergence.

Overall, 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. This is likely inherent in the behavior in the asynchronous setting of our implementations, where the consistency guarantees are relaxed in a way such that vertices can more easily move better clusters. Overall, PAR-CC offers a better precision-recall trade-off compared to PAR-MOD, which shows the benefits of using the CC objective. We also show in the appendix the same behavior on dblp and livejournal.

Figure 10 shows the average precision-recall curves for TECTONIC [42], 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 much better precision-recall on dblp, livejournal, and orkut. Notably, TECTONIC degrades significantly on the larger graphs livejournal and orkut compared to PAR-CC.

We also conducted quality experiments on weighted graphs, but we defer the details to the appendix.

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 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.

[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.

[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.

[4] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. 2004. Correlation Clustering. *Mach. Learn.* 56, 1–3 (June 2004), 89–113.

[5] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA)*. Association for Computing Machinery, New York, NY, USA, 507–509.

[6] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast Unfolding of Communities in Large Networks. *J. Stat. Mech. Theory Exp.* 2008, 10 (Oct. 2008), 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] Flavio Chierichetti, Nilesh Dalvi, and Ravi Kumar. 2014. Correlation Clustering in MapReduce. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD)*. Association for Computing Machinery, New York, NY, USA, 641–650.

[9] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM, 11:1–11:8.

[10] T. N. Dinh, X. Li, and M. T. Thai. 2015. Network Clustering via Maximizing Modularity: Approximation Algorithms and Theoretical Limits. In *IEEE International Conference on Data Mining (ICDM)*. 101–110.

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

[12] Micha Elsner and Warren Schudy. 2009. Bounding and Comparing Methods for Correlation Clustering Beyond ILP. In *Workshop on Integer Linear Programming for Natural Langauge Processing*. 19–27.

[13] Mahmood Fazlali, Ehsan Moradi, and Hadi Tabatabaee Malazi. 2017. Adaptive Parallel Louvain Community Detection on a Multicore Platform. *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)* 54 (2017), 26 – 34.

[14] Alexandre Fender, Nahid Emad, Serge Petiton, and Maxim Naumov. 2017. Parallel Modularity Clustering. *Procedia Computer Science* 108 (2017), 1793–1802.

[15] David García-Soriano, Konstantin Kutzkov, Francesco Bonchi, and Charalampos Tsourakakis. 2020. Query-Efficient Correlation Clustering. In *The Web Conference* (Taipei, Taiwan) *(WWW)*. Association for Computing Machinery, New York, NY, USA, 1468–1478.

[16] Ullas Gargi, Wenjun Lu, Vahab Mirrokni, and Sangho Yoon. 2011. Large-scale Community Detection on Youtube for Topic Discovery and Exploration. In *AAAI Conference on Weblogs and Social Media (ICWSM)*, Vol. 5. 486–489.

[17] 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 *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 885–895.

[18] 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.

[19] M. Girvan and M. E. J. Newman. 2002. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences (PNAS)* 99, 12 (2002), 7821–7826.

[20] 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.

[21] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. 2017. Scalable Static and Dynamic Community Detection using Grappolo. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[22] A. K. Jain, M. N. Murty, and P. J. Flynn. 1999. Data Clustering: A Review. *ACM Comput. Surv.* 31, 3 (Sept. 1999), 264–323.

[23] 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.

[24] 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.

[25] 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.

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

[27] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. 2015. Parallel Heuristics for Scalable Community Detection. *Parallel Comput.* 47 (2015), 19 – 37. Special Issue: Graph Analysis for Scientific Discovery.

[28] 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.

[29] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. 2015. Parallel Correlation Clustering on Big Graphs. In *International Conference on Neural Information Processing Systems* (Montreal, Canada) *(NIPS)*, Vol. 1. MIT Press, Cambridge, MA, USA, 82–90.

[30] Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. 2014. High Quality, Scalable and Parallel Community Detection for Large Real Graphs. In *International Conference on World Wide Web* (Seoul, Korea) *(WWW)*. Association for Computing Machinery, New York, NY, USA, 225–236.

[31] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels. 2015. Scalable Community Detection with the Louvain Algorithm. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 28–37.

[32] U. Raghavan, R. Albert, and S. Kumara. 2007. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.* 76 (2007).

[33] Jörg Reichardt and Stefan Bornholdt. 2006. Statistical Mechanics of Community Detection. *Phys. Rev. E* 74 (Jul 2006), 016110. Issue 1.

[34] Jason Riedy, David A Bader, and Henning Meyerhenke. 2012. Scalable Multi-Threaded Community Detection in Social Networks. In *IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPS)*. IEEE, 1619–1628.

[35] 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.

[36] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, USA.

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

[38] Satu Elisa Schaeffer. 2007. Survey: Graph Clustering. *Comput. Sci. Rev.* 1, 1 (Aug. 2007), 27–64.

[39] C. L. Staudt and H. Meyerhenke. 2016. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27, 1 (2016), 171–184.

[40] V. A. Traag. 2015. Faster Unfolding of Communities: Speeding Up the Louvain Algorithm. *Phys. Rev. E* 92 (Sept. 2015), 032801. Issue 3.

[41] 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.

[42] Charalampos E. Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. 2017. Scalable Motif-Aware Graph Clustering. In *World Wide Web Conference* (Perth, Australia) *(WWW)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1451–1460.

[43] Nate Veldt, David F. Gleich, and Anthony Wirth. 2018. A Correlation Clustering Framework for Community Detection. In *World Wide Web Conference* (Lyon, France) *(WWW)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 439–448.

[44] Wayne W. Zachary. 1977. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research* 33, 4 (1977), 452–473.

[45] J. Zeng and H. Yu. 2015. Parallel Modularity-Based Community Detection on Large-Scale Graphs. In *IEEE International Conference on Cluster Computing*. 1–10.

A SEQUENTIAL LOUVAIN METHOD

Algorithm 2 shows the pseudocode for the classic sequential Louvain method from Blondel *et al.* [6], SEQUENTIAL-CC.

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

Algorithm 2 Sequential Louvain method for correlation clustering

```
1: procedure SEQUENTIAL-CC($G$, $k$)
2:     $C \leftarrow$ singleton clusters for each $v \in V$
3:     do
4:         $\sigma \leftarrow$ random permutation of $V$
5:         for each $v = \sigma(i)$ 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:    $G'$, $k' \leftarrow$ SEQUENTIAL-COMPRESS($G$, $C$)
11:    $C' \leftarrow$ SEQUENTIAL-CC($G'$, $k'$)
12:    return SEQUENTIAL-FLATTEN($C$, $C'$)
```

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.

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'.

B 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) [9]. Overall, these primitives and the work-efficient scheduler provided in GBBS offer on average a 1.43x speedup over Intel's Parallel STL library [5]. 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 [18], from the GBBS implementation, 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.

C ADDITIONAL EXPERIMENTS

Table 1 shows the details of the SNAP graphs that we perform experiments upon.

Figure 11 shows the size of V' comparing neighbors of clusters and neighbors of vertices as the subset V', for PAR-CC on amazon and orkut, using the synchronous setting and no refinement. In particular, for $\lambda = 0.85$ on amazon, the neighbors of vertices optimization obtains 1.61x speedup over all vertices, whereas the neighbors of clusters optimization obtains 1.16x speedup over all vertices. This is reflected in Figure 11, where there is a significant difference in the size of V' in this case.

We note that we see lower speedups using PAR-MOD on twitter. We suspect that this is because twitter has a few vertices of particularly high degree (the maximum degree is 2,997,487, compared to the maximum degree in friendster of 5,214), and, across all resolutions, PAR-MOD and SEQ-MOD produce significantly fewer clusters on twitter relative to the size of the graph, compared to other graphs. In particular, for modularity clustering, the average cluster size on twitter is $2100–2.08 \times 10^7$ across all resolutions, whereas the average cluster size on friendster, a graph of similar size, is 1.11. As such, we see significantly increased contention on twitter.

Figure 12 shows the scalability of PAR-MOD over rMAT graphs of varying sizes, with very sparse graphs where $m = 5n$, sparse graphs where $m = 50n$, dense graphs where $m = n^{1.5}$, and very dense graphs where $m = n^2$. Figure 13 shows the speedups of PAR-MOD on amazon, orkut, twitter, and friendster, over different numbers of threads.

We show in Figure 14 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.

C.1 Experiments on Existing Baselines

We 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! [29], which are based on a maximal independent set algorithm, and the sequential Louvain-based method in LAMBDACC [43].

The parallel correlation clustering implementations C4 and CLUSTERWILD! [29] optimize for the same objective as our CC objective,

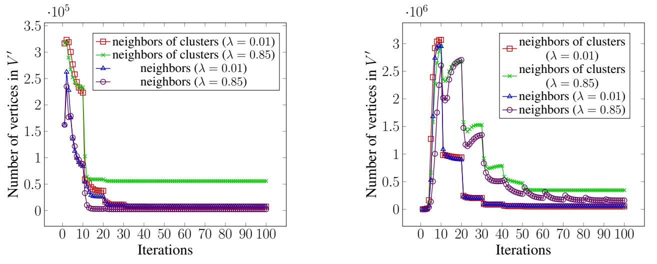

Figure 11: Number of vertices in V' per best move iteration of PAR-CC, for amazon (left) and orkut (right), considering neighbors of clusters and neighbors of vertices as the subset V'.

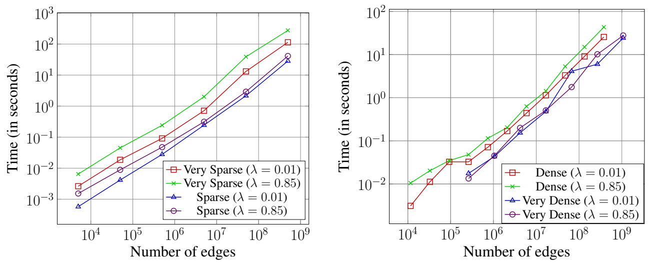

Figure 12: Scalability of PAR-MOD over rMAT graphs with varying numbers of vertices.

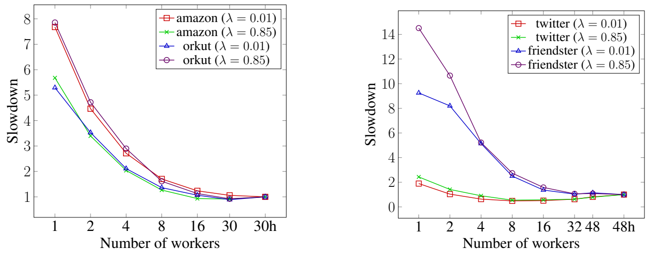

Figure 13: Scalability of PAR-MOD over different numbers of threads, on amazon, orkut, twitter, and friendster, with $\lambda = 0.01, 0.85$. Note that 30h and 48h indicate 30 and 48 cores respectively, with two-way hyper-threading.

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 they are unsuitable for optimizing for the CC objective. Moreover, compared to the top 5000 ground truth communities on these graphs, C4 and CLUSTERWILD! achieve poor precision and recall, with precision between 0.44–0.65 and corresponding recall between 0.10–0.15. In comparison, PAR-CC achieves recall between 0.61–0.98 for precision greater than 0.50.

Additionally, we compare against the Louvain-based sequential correlation clustering implementation in LAMBDACC given by Veldt *et al.* [43]. 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 [44], 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. The slowness of LAMBDACC is because the code is in MATLAB, and it uses an adjacency matrix to represent the input graph; as such, it is unable to efficiently perform sparse graph operations.

We compare to NETWORKIT [39] in the special case of modularity. 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 Figure 17 the speedups of PAR-MOD over NETWORKIT on amazon, dblp, livejournal, and orkut, for varying resolutions. We see up to 3.50x speedups, primarily due to our optimization of the graph compression step, and on average 1.89x speedups. 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.

Finally, we compare to SCD [30], 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.

C.2 Experiments on Weighted Graphs

We 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 [11], both of which also have ground truth clusters which we compare to. We use the state-of-the-art ScaNN k-NN library [20] to perform k-NN, with $k = 50$ and using cosine similarity. We symmetrize the resulting k-NN graph.

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 ^{W}. The sequential implementations SEQ-CC and SEQ-MOD give similar results to the corresponding parallel implementations, so we discuss only the parallel implementations.

Figures 15 and 16 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

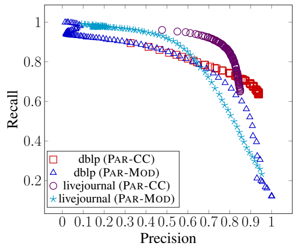

Figure 14: Average precision and recall compared to ground truth communities on dblp and livejournal of the clusters obtained from our parallel implementations Par-CC and Par-Mod, using varying resolutions.

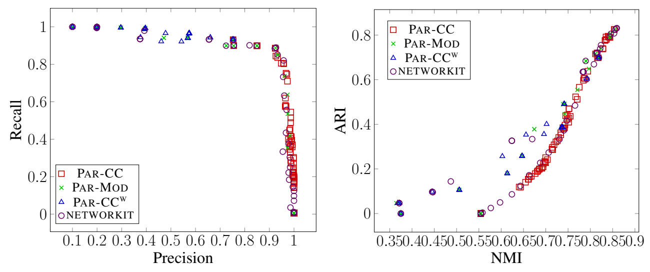

Figure 15: Average precision-recall and ARI-NMI scores for the digits graph, from Par-CC, Par-Mod, Par-CC^W, and NetworKit.

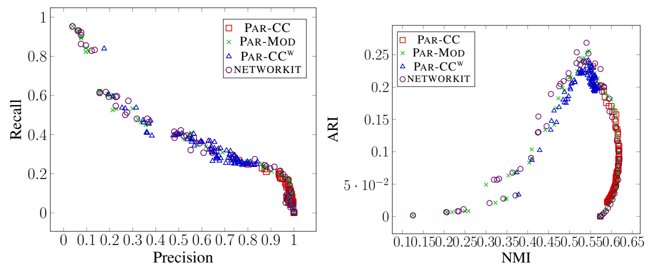

Figure 16: Average precision-recall and ARI-NMI scores for the letter graph, from Par-CC, Par-Mod, Par-CC^W, and NetworKit.

implementation, which can also take as input weighted graphs [39]. NetworKit matches our Par-Mod^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. Overall, compared to NetworKit, Par-CC^W is more robust across different resolution parameters compared to other implementations.

D 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 D.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 18 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. 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 $f(c_i) = 1 \Big/ \prod_{1 \le j < i} d_{c_j}$, 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.

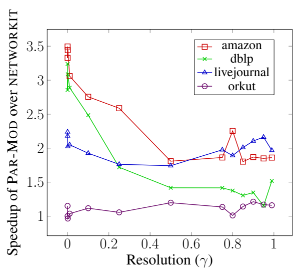

Figure 17: Speedup of PAR-MOD over NETWORKIT on amazon, dblp, livejournal, and orkut, for varying resolutions.

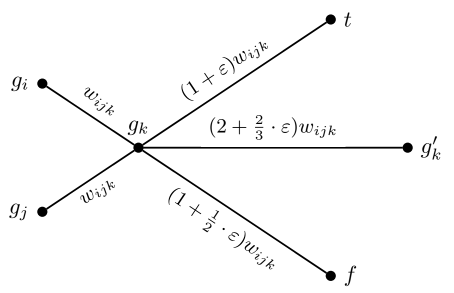
g_i
g_j
g_k
t
g'_k
f
w_{ijk}
w_{ijk}
$(1+\varepsilon)w_{ijk}$
$(2+\frac{2}{3}\cdot\varepsilon)w_{ijk}$
$(1+\frac{1}{2}\cdot\varepsilon)w_{ijk}$

Figure 18: The vertices and edges added to the graph G, given a gate $g_i \vee g_j$ which outputs to g_k.

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.

□