Scaling up Correlation Clustering through Parallelism and Concurrency Control

Xinghao Pan
EECS, UC Berkeley

Dimitris Papailiopoulos
EECS, UC Berkeley

Benjamin Recht
EECS and Statistics, UC Berkeley

Kannan Ramchandran
EECS, UC Berkeley

Michael I. Jordan
EECS and Statistics, UC Berkeley

Abstract

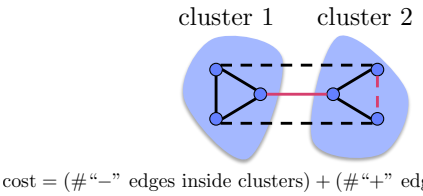
Given a similarity graph between items, correlation clustering (CC) aims to group similar items together and dissimilar ones apart. One of the most popular CC algorithms is *KwikCluster*: a simple peeling scheme that offers a 3-approximation ratio. Unfortunately, *KwikCluster* is inherently sequential and can require a large number of peeling rounds. This can be a significant bottleneck when scaling up to big graphs. Recent proposals to parallelize *KwikCluster* encounter challenges in scaling up, while sometimes they introduce a loss to the 3 approximation factor.

We present *C4*, a parallel CC algorithm that obtains a 3-approximation ratio, has limited overheads, and gracefully scales up to billion-edge graphs. The main idea behind *C4* is running multiple peeling threads concurrently, while ensuring consistency among them without many overheads. We enforce consistency through *concurrency control*, a popular paradigm in database research. We provide extensive experiments and demonstrate that *C4* can scale up to billion-edge graphs where it outputs a clustering in a few seconds.

1 Introduction

Clustering items according to some notion of similarity is a major primitive in machine learning. *Correlation clustering* is a very basic clustering variant: given a similarity measure between pairs of items, CC aims to group these items in clusters so that the number of unsatisfied pairs is minimized.

In the simplest setup, we are given a graph $\mathcal{G}$ on n vertices, $+1$ weights on edges between similar items, and -1 weights on edges for dissimilar items. Correlation clustering aims to generate clusters of items that minimize the number of erroneously clustered pairs, commonly referred to as the *number of disagreements*. In an output clustering, the number of disagreements corresponds to the number of "$+$" edges cut by the clusters plus the number of "$-$" edges inside the clusters. In Figure 1, we give a simple example of a CC instance for a graph on 6 vertices.



cost = (#"$-$" edges inside clusters) + (#"$+$" edges across clusters) = 2

Figure 1: In the above graph, solid edges denote similarity, and dashed denote dissimilarity. Let the two clusters shown above be the output of a CC algorithm. The metric of interest (i.e., the number of disagreements) for the given clustering is 2; we denote with red the edges that incur a cost.

Observe that the number of clusters *is not* given as a parameter to the problem, i.e., the algorithm can output an arbitrary number of clusters. This comes in sharp contrast to other clustering approaches such as k-means, or k-median, where the number of clusters needs to be a priori defined.

Applications. Entity deduplication is one of the traditional motivations for correlation clustering. Entity deduplication finds application in chat disentanglement, co-reference resolution, and spam detection [1, 2, 3, 4, 5, 6]. In this problem, we assume that there are some entities (say, results of a keyword search), and a pairwise classifier that indicates (with a possible error), when two entities are the same. In the context of a keyword search, two results might refer to the same item, however, if they come from different sources the results will appear slightly different. By building a similarity graph between these entities and by applying CC one hopes to cluster all duplicate entities in the same similarity class; in the context of keyword search, this would imply a more meaningful and compact list of results.

Correlation clustering is also useful in finding communities in signed networks, or classifying missing edges in opinion, or trust networks [7, 8]. Given a network of social interactions, the weights on edges can indicate likes/dislikes, or for opinion networks agreements/disagreements between individuals. In this setup, CC aims to cluster individuals in groups of agreement, while having people that disagree in different clusters. In the context of classifying missing edges in opinion networks, CC can be thought of as a way to "guess" the missing edge signs. Further applications include gene clustering according to expression pattern similarity [9], and consensus clustering [3].

A simple 3-approximation, and efforts to scale up. Arguably the simplest algorithm for CC is *KwikCluster*, a 3-approximation by [10]. *KwikCluster* works in the following way: pick a vertex v at random, create a cluster that contains v and its positive neighborhood $N^+(v)$, peel these vertices from the graph, and repeat. Apart from its theoretical guarantees, *KwikCluster* has been experimentally shown to perform reasonably well, when combined with additional local improvement heuristics [3].

The main drawback of *KwikCluster* is its number of peeling rounds: although its expected running time is $O(n + \mathsf{OPT})$ [11], the algorithm can suffer from the fact that it is inherently sequential, and can require up to $O(n)$ rounds. This can be problematic when scaling up to large graphs, or when considering implementations in distributed paradigms like MapReduce.

Recently, there have been efforts to develop scalable variants of *KwikCluster* [5, 6]. In [6] a distributed peeling algorithm was presented in the context of MapReduce. Using an elegant analysis, the authors show that their algorithm achieves a $(3 + O(\epsilon))$-approximation in $O\left(1/\epsilon \cdot \log n \cdot \log \Delta^+\right)$ rounds, where Δ^+ is the maximum positive edge degree. A potential issue is that obtaining an approximation provably close to 3 (i.e., close to that of *KwikCluster*) requires in the worst case thousands of MapReduce rounds for large graphs: the leading constants in the asymptotic analysis of the algorithm are not negligible. Unfortunately, the number of rounds has been a known bottleneck for MapReduce implementations, with reasonable numbers varying between tens to a few hundred rounds.

Additionally in [5], the rough details of a distributed algorithm were presented. This algorithm achieves the same approximation as *KwikCluster* in an expected $O(\log n)$ number of rounds. However, as this algorithm is not yet published, we cannot present a fair and meaningful comparison.

Our contributions. We present *C4*, a parallel correlation clustering algorithm that obtains the same 3-approximation as *KwikCluster*. Our algorithm has limited overheads and can gracefully scale up to billion-edge graphs. The main idea behind *C4* is that it allows multiple peeling threads that run concurrently. We show that if one enforces consistency among these peeling threads, then the output of *C4* is equivalent to the output of the serial *KwikCluster*. We enforce consistency among the peeling threads through *concurrency control*, a notion that has been extensively studied in the context of databases. Concurrency control has been recently introduced as a means for consistent parallelization of inherently sequential machine learning algorithms [12].

Using concurrency control, we are able to block peeling threads that can cause inconsistencies in the output solution. The main computational benefit of the algorithm is that blocks happen infrequently, when running on up to 16 cores. In theory, the expected number of total thread blocks is proportional to the graph's average positive degree times the number of cores. In practice, we observe that more than 99.9% of peeling threads run without being blocked. Experimentally, this translates to an almost linear speed-up on up to 16 threads, when comparing to single thread execution.

We provide an extensive experimental evaluation of *C4* and demonstrate that it can scale up to graphs with millions of vertices and more than a billion edges. We show that even in these large graphs *C4*

outputs a valid clustering of all vertices in less than five seconds, up to an order of magnitude faster than *KwikCluster*. We conclude that *C4* is suitable for large-graph cases, where the number of peeling rounds renders *KwikCluster* slow, and the graph can still fit in main memory of a few machines.

Related algorithmic state of the art Correlation clustering was formally introduced by Bansal et al. [13]. In the general case, minimizing disagreements is NP-hard and hard to approximate within an arbitrarily small constant (APX-hard) [13, 14]. There are two variations of the problem: *i)* CC on complete graphs where all edges are present and all weights are ± 1, and *ii)* CC on general graphs with arbitrary edge weights. Both problems are hard, however the general graph setup seems fundamentally harder. The best known approximation ratio for the latter is $O(\log n)$, and a reduction to the minimum multicut problem indicates that any improvement to that requires fundamental breakthroughs in theoretical algorithms [15]. In the case of complete unweighted graphs, a long series of results establishes a 2.5 approximation via a rounded linear program (LP) [10]. By avoiding the expensive LP, and by just using the rounding procedure of [10] as a basis for a greedy algorithm yields *KwikCluster*: a 3 approximation for CC on complete unweighted graphs.

2 Concurrency Control for Machine Learning

In this section, we briefly present the key ideas behind concurrency control and its application in parallelizing sequential machine learning algorithms.

In many cases, machine learning algorithms iteratively process data points and transform some global state (e.g. model parameters) giving the illusion of serial dependencies between iterations. More precisely, each iteration is a transformation $T_i : (v_i, S) \mapsto S'$ that produces a new global state S' given a data point v_i and the current global state S. The algorithm itself is a sequence of local updates on the global state using one data point per update. In the context of *KwikCluster*, the state S corresponds to an assignment of vertices to clusters, and each $T_{\pi(v)}(v, S)$ creates a new cluster around a vertex v, if that vertex is not yet assigned to an existing cluster in S.

In this paper, we take a transactional view of the *KwikCluster* algorithm, and explore parallelization through the lens of parallel database transaction processing systems. Specifically, we cast each transformation T_i as a *database transaction*, and apply ideas from database systems research to parallelizing execution of the algorithm. Parallel execution of transactions has been a focus of the database research community for decades, with the dual objectives of ensuring *serializability* – the outcome of the parallel execution must be equivalent to that of the serial algorithm – and maximizing concurrency. Concurrency control mechanisms have been developed as a result to allow transactions to execute concurrently as long as they do not conflict with one another.

The transactional approach to concurrent machine learning prescribes designing parallel algorithms to guarantee serializability through the application of concurrency control mechanisms. As a consequence, theoretical properties about the serial algorithm extend to the parallel algorithms without modification. The analysis of the parallel algorithms focuses on demonstrating concurrency, for example, by bounding the overheads in terms of the number of blocked or re-executed transactions.

3 *C4*: Parallel Correlation Clustering with Concurrency Control

The serial *KwikCluster* algorithm (Alg. 1) is a sequence of iterative operations that *i)* checks if a vertex v has been assigned to a cluster, and *ii)* if v is unassigned, then set it as a cluster center and assign its (unassigned) neighbors to its cluster. To do this, the algorithm maintains $\gamma_{ser}(v)$, an indicator of whether v is a CENTER, SPOKE or UNASSIGNED, and $\kappa_{ser}(v)$, the id of v's cluster. The order in which *KwikCluster* visits vertices is dictated by a random permutation π of $\{1, \ldots, n\}$. In the following we denote by V the set of vertices, and by E^+, the set of positive edges.

We apply both blocking and optimistic concurrency control mechanisms in *C4* (Alg. 2), our parallel version of *KwikCluster*. Each *C4* transaction $T_{\pi(v)}$ in essence performs the same two operations as an iteration in serial *KwikCluster*, and maintains analogous data $\gamma_{C4}(v)$ and $\kappa_{C4}(v)$. To verify that a vertex v should become a new cluster center, transaction $T_{\pi(v)}$ checks that none of v's neighbors that have an earlier order is a cluster center. That is, it verifies that for all u such that $\pi(u) < \pi(v)$ we have that $((u, v) \notin E^+ \vee \gamma_{C4}(v) = \text{SPOKE})$. The verification process (Alg 3) employs the use of waits to ensure $T_{\pi(v)}$ reads the correct value of $\gamma_{C4}(u)$ for all neighbors u such that $\pi(u) < \pi(v)$.

If v is created as a CENTER, $T_{\pi(v)}$ then optimistically assigns all unassigned neighbors u of v to v's cluster. This may, however, create a conflict if u has a neighbor w that is also a CENTER, but has

Algorithm 1: *KwikCluster*: serial peeling

```
1  Init ∀v ∈ V, κ_ser(v) = ∞
2  Init ∀v ∈ V, γ_ser(v) = UNASSIGNED
3  for i = 1 to n do
4      Let v be vertex such that π(v) = i.
5      if γ_ser(v) == UNASSIGNED then
6          γ_ser(v) = CENTER
7          κ_ser(v) = π(v)
8          for u : (u, v) ∈ E+ do
9              if γ_ser(u) == UNASSIGNED then
10                 γ_ser(u) = SPOKE
11                 κ_ser(u) = π(v)
```

Algorithm 2: *C4*: Parallel peeling

```
1  Init ∀v ∈ V, κ_C4(v) = ∞
2  Init ∀v ∈ V, γ_C4(v) = UNASSIGNED
3  for p ∈ {1, ..., P} do in parallel
4      for i = p, p + P, ..., p + ⌊n/P⌋P do
           // Transaction T_v
5          Let v be vertex such that π(v) = i.
6          if γ_C4(v) == UNASSIGNED then
               // Check concurrent neighbors
7              isCenter = verifyIsCenter(v)
               // Create cluster
8              if isCenter then createCluster(v)
```

Algorithm 3: `verifyIsCenter(v)`

```
1  for u : (u, v) ∈ E+ do
2      if π(u) < π(v) then
3          wait until γ_C4(u) ≠ UNASSIGNED
4          if γ_C4(u) == CENTER then
5              return false
6  return true
```

Algorithm 4: `createCluster(v)`

```
1  γ_C4(v) = CENTER
2  κ_C4(v) = π(v)
3  for u : (u, v) ∈ E+ do
       // Atomic check & set
4      if κ_C4(u) > π(v) then
5          γ_C4(u) = SPOKE;   κ_C4(u) = π(v)
```

$\pi(w) < \pi(v)$, since u should have rightly been assigned to w's cluster instead. *KwikCluster* resolves this conflict by allowing $T_{\pi(w)}$ to reassign u to w's cluster. We show that the use of concurrency control in *KwikCluster* guarantees serializability, and establish the following result. The proofs are omitted due to lack of space.

Theorem 3.1. *C4 achieves a 3-approximation ratio (in expectation) with respect to the optimal clustering assignment for the metric of disagreements.*

Concurrency of *C4* and wait times. *C4* examines the edges of each cluster center twice – first to determine that it should in fact be made a center, and again to put its neighbors into the cluster. Although this amounts to twice the edges scanned by serial KwikCluster, the work can be parallelized and does not constitute a serial overhead.

In addition, some of C4's transactions need to block to wait for concurrent transactions to complete, essentially forcing the conflicting transactions to be serially executed. We argue that such events are uncommon and thus not a serious obstacle to attaining high concurrency.

Let $T_v.begin$ and $T_v.end$ be the start and end times of T_v, the C4 transaction on v. Let $\tau = \max_{v \in V}(\pi(v) - \min\{\pi(u) : \pi(u) < \pi(v) \wedge T_u.end > T_v.begin\})$. Intuitively, τ is the maximum discrepancy between the progress of the multiple processors in executing transactions. In general, we expect τ to be in the order of number of processors.

Theorem 3.2. *The expected number of blocked transactions is upper bounded by* $2\tau|E^+|/|V|$.

We expect that real graphs will have low average degree $2|E^+|/|V|$, and thus the expected *fraction* of blocked transactions $2\tau|E^+|/|V|^2$ should be insignificant, and most transactions need not wait.

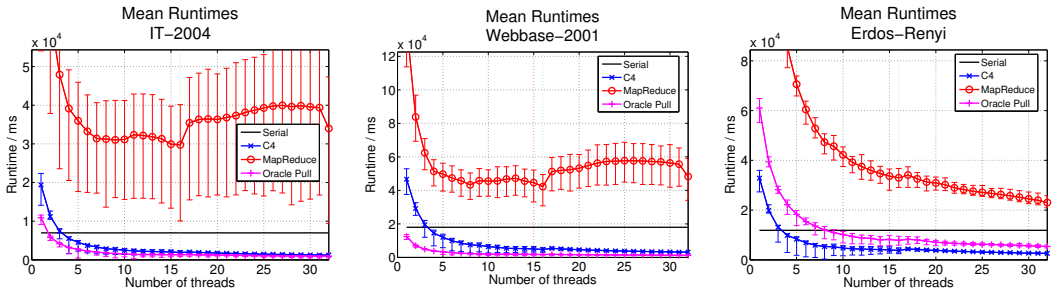
4 Experiments and Preliminary Results

We implemented our algorithms in Scala / Java and ran our experiments on Amazon EC2 r3.8xlarge instances (2 Intel Xeon E5-2670 v2 CPUs, 2.50GHz, 8 cores per CPU, 2 threads per core, 244 GiB memory). We tested on 3 graphs (Table 1), with 10 runs each of 10 random permutation, for a total of 100 runs per graph.

Graph	# vertices	# edges	Description
Erdos-Renyi	100,000,000	$\approx 1 \times 10^9$	Each edge is included with probability 2×10^{-7}.
IT-2004	41,291,594	1,135,718,909	2004 crawl of the .it domain [16, 17, 18].
WebBase-2001	118,142,155	1,019,903,190	2001 crawl by WebBase crawler [16, 17, 18].

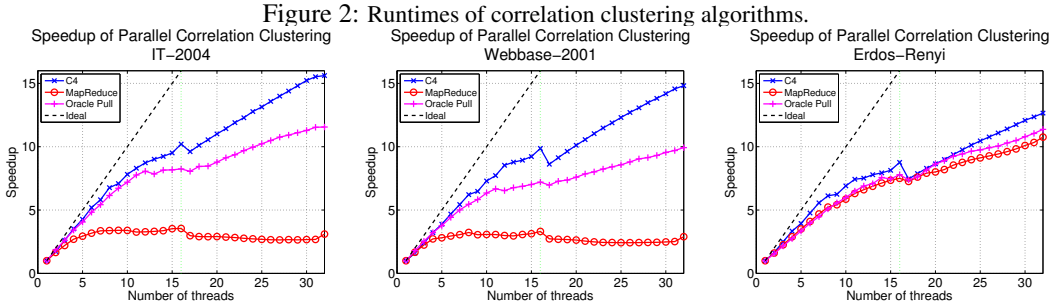
Table 1: Synthetic and real graphs used in the evaluation of C4.

Algorithms for comparison. We implemented serial KwikCluster on non-fenced memory. We also compare against an oracle 'pull assignment' approach, where the cluster centers are assumed to be known, and each spoke then assigns itself to the lowest ordered adjacent cluster center: $\kappa_{pull}(v) = \min\{\kappa_{pull}(u) : (u, v) \in E^+ \wedge \gamma_{pull}(u) = \text{CENTER}\}$. This operation trivially parallelizes over vertices, but requires a scan of all edges incident to the spokes. Conversely, our approach depends on centers to push assignments to spokes, but resolves conflicts at the spokes.



(a) IT-2004 (b) WebBase-2001 (c) Erdos-Renyi

Figure 2: Runtimes of correlation clustering algorithms.



(a) IT-2004 (b) WebBase-2001 (c) Erdos-Renyi

Figure 3: Speedups of correlation clustering algorithms.

We also considered an alternative approach using MapReduce based on ideas in [5]. In each round t, each vertex v checks if $\exists u : (u, v) \in E^+ \wedge \pi(u) < \pi(v) \wedge \gamma_{MR}(u) = \text{CENTER}$, and if so, sets $\gamma_{MR}(v) = \text{SPOKE}$. However, if $\gamma_{MR}(u) = \text{SPOKE}$ for all u such that $(u, v) \in E^+ \wedge \pi(u) < \pi(v)$, then we set $\gamma_{MR}(v) = \text{CENTER}$. Updates are sent to the fenced memory (simulating a key-value store) to efficiently inform neighbors of changes in cluster assignments. Finally, after cluster centers are found, κ_{MR} is assigned by an oracle pull. We emphasize that we tested our own multicore implementation, and not the distributed implementation of [5].

Results. On all 3 graphs, *C4* was always faster than serial *KwikCluster* with 4 threads (Fig 2). We also measured the speedups – the ratio of runtime on 1 thread to the runtime on p threads. For C4, we were able to achieve up to 10x speedup with 16 threads, and up to 15x speedup with 32 threads (Fig 3). With 32 threads, *C4* is 4-6x faster than serial KwikCluster (Fig 4c). The percentage of blocked C4 transactions (Fig 4a) is always below 0.02%, i.e. 99.98% of transactions proceed without waiting.

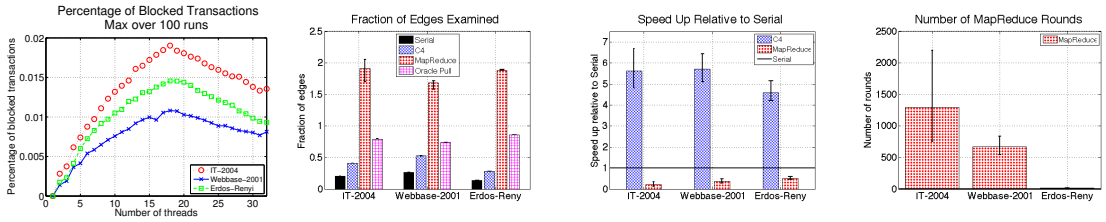
We note that *C4*'s runtime is competitive with the oracle pull assignment, and faster on the Erdos-Renyi graph. This is because the oracle pull assignment examines more edges than C4 (Fig. 4b).

Our MapReduce implementation is always slower than *C4*, and also slower than serial *KwikCluster* even on 32 threads. Admittedly, our implementation may not be fully optimized. However, we observe that the MapReduce approach takes hundreds of rounds (Fig 4d) on the real graphs (compared to about 20 rounds on Erdos-Renyi), which tends to negatively impact scalability (Fig 3). Furthermore, this approach performs redundant work processing vertices and edges multiple times (Fig 4b) until the vertices are assigned to a cluster. Hence, we believe that our asynchronous approach in *C4* is more scalable than the MapReduce one.

5 Conclusion and Future Work

We proposed *C4*, a parallel correlation clustering algorithm that employs concurrency control mechanisms. Our approach preserves the 3-approximation of serial *KwikCluster*, and has low expected overheads. We also demonstrated empirically that *C4* is scalable on real and synthetic graphs with billion edges, outperforming serial *KwikCluster* by 4-6x on 32 threads.

While we have presented this work in a multicore setting, we recognize that there are situations where distributed solutions are preferable, such as when the graph is too large to fit on a single machine, or if the graph is already loaded in distributed memory. *C4* extends easily to the distributed setting and we are currently working on distributed implementation and analysis.



(a) (b) (c) (d)

Figure 4: (a) Maximum percentage of blocked *C4* transactions over 100 runs is less than 0.02%. (b) Fraction of edges examined by various algorithms. (c) Speedup of *C4* and MapReduce over serial *KwikCluster*. (d) Number of MapRedue rounds used to identify cluster centers.

We also note that *C4* incurs some overheads to ensure serializability, particularly in resolving conflicts between adjacent vertices concurrently creating new clusters. Removing this overhead leads to faster runtimes and better scalability, but could introduce additional errors. We are currently studying the approximation ratio and empirical scalability of this lock-free style approach.

References

[1] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(1):1–16, 2007.

[2] Arvind Arasu, Christopher Ré, and Dan Suciu. Large-scale deduplication with constraints using dedupalog. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 952–963. IEEE, 2009.

[3] Micha Elsner and Warren Schudy. Bounding and comparing methods for correlation clustering beyond ilp. In *Proceedings of the Workshop on Integer Linear Programming for Natural Langauge Processing*, pages 19–27. Association for Computational Linguistics, 2009.

[4] Bilal Hussain, Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J Miller. An evaluation of clustering algorithms in duplicate detection. Technical report, 2013.

[5] Francesco Bonchi, David Garcia-Soriano, and Edo Liberty. Correlation clustering: from theory to practice. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1972–1972. ACM, 2014.

[6] Flavio Chierichetti, Nilesh Dalvi, and Ravi Kumar. Correlation clustering in mapreduce. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 641–650. ACM, 2014.

[7] Bo Yang, William K Cheung, and Jiming Liu. Community mining from signed social networks. *Knowledge and Data Engineering, IEEE Transactions on*, 19(10):1333–1348, 2007.

[8] N Cesa-Bianchi, C Gentile, F Vitale, G Zappella, et al. A correlation clustering approach to link classification in signed networks. In *Annual Conference on Learning Theory*, pages 34–1. Microtome, 2012.

[9] Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. Clustering gene expression patterns. *Journal of computational biology*, 6(3-4):281–297, 1999.

[10] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):23, 2008.

[11] Nir Ailon and Edo Liberty. Correlation clustering revisited: The true cost of error minimization problems. In *Automata, Languages and Programming*, pages 24–36. Springer, 2009.

[12] Xinghao Pan, Joseph E Gonzalez, Stefanie Jegelka, Tamara Broderick, and Michael Jordan. Optimistic concurrency control for distributed unsupervised learning. In *Advances in Neural Information Processing Systems*, pages 1403–1411, 2013.

[13] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 238–238. IEEE Computer Society, 2002.

[14] Moses Charikar, Venkatesan Guruswami, and Anthony Wirth. Clustering with qualitative information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 524–533. IEEE, 2003.

[15] Erik D Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. Correlation clustering in general weighted graphs. *Theoretical Computer Science*, 361(2):172–187, 2006.

[16] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, 2004.

[17] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*. ACM Press, 2011.

[18] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.