Parallel Correlation Clustering on Big Graphs

Xinghao Pan[α,ε], Dimitris Papailiopoulos[α,ε], Samet Oymak[α,ε]
Benjamin Recht[α,ε,σ], Kannan Ramchandran[ε], Michael I. Jordan[α,ε,σ]
[α]AMPLab, [ε]EECS at UC Berkeley, [σ]Statistics at UC Berkeley

February 2, 2022

Abstract

Given a similarity graph between items, correlation clustering (CC) groups similar items together and dissimilar ones apart. One of the most popular CC algorithms is *KwikCluster*: an algorithm that serially clusters neighborhoods of vertices, and obtains a 3-approximation ratio. Unfortunately, *KwikCluster* in practice requires a large number of clustering rounds, a potential bottleneck for large graphs.

We present *C4* and CLUSTERWILD!, two algorithms for parallel correlation clustering that run in a polylogarithmic number of rounds and achieve nearly linear speedups, provably. *C4* uses concurrency control to enforce serializability of a parallel clustering process, and guarantees a 3-approximation ratio. CLUSTERWILD! is a coordination free algorithm that abandons consistency for the benefit of better scaling; this leads to a provably small loss in the 3-approximation ratio.
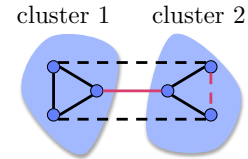
We provide extensive experimental results for both algorithms, where we outperform the state of the art, both in terms of clustering accuracy and running time. We show that our algorithms can cluster billion-edge graphs in under 5 seconds on 32 cores, while achieving a 15× speedup.

1 Introduction

Clustering items according to some notion of similarity is a major primitive in machine learning. *Correlation clustering* serves as a basic means to achieve this goal: given a similarity measure between items, the goal is to group similar items together and dissimilar items apart. In contrast to other clustering approaches, the number of clusters is not determined a priori, and good solutions aim to balance the tension between grouping all items together versus isolating them.

The simplest CC variant can be described on a complete signed graph. Our input is a graph G on n vertices, with $+1$ weights on edges between similar items, and -1 edges between dissimilar ones. Our goal is to generate a partition of vertices into disjoint sets that minimizes the number of *disagreeing edges*: this equals the number of "+" edges cut by the clusters plus the number of "−" edges inside the clusters. This metric is commonly called the *number of disagreements*. In Figure 1, we give a toy example of a CC instance.

Entity deduplication is the archetypal motivating example for correlation clustering, with applica-

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

Figure 1: In the above graph, solid edges denote similarity and dashed dissimilarity. The number of disagreeing edges in the above clustering is 2; we color these edges with red.

tions in chat disentanglement, co-reference resolution, and spam detection [1, 2, 3, 4, 5, 6]. The input is a set of entities (say, results of a keyword search), and a pairwise classifier that indicates—with some error—similarities between entities. Two results of a keyword search might refer to the same item, but might look different if they come from different sources. By building a similarity graph between entities and then applying CC, the hope is to cluster duplicate entities in the same group; in the context of keyword search, this

implies a more meaningful and compact list of results. CC has been further applied to finding communities in signed networks, classifying missing edges in opinion or trust networks [7, 8], gene clustering [9], and consensus clustering [3].

KwikCluster is the simplest CC algorithm that achieves a provable 3-approximation ratio [10], and works in the following way: pick a vertex v at random (a *cluster center*), create a cluster for v and its positive neighborhood $N(v)$ (i.e., vertices connected to v with positive edges), peel these vertices and their associated edges from the graph, and repeat until all vertices are clustered. Beyond its theoretical guarantees, experimentally *KwikCluster* performs well when combined with local heuristics [3].

KwikCluster seems like an inherently sequential algorithm, and in most cases of interest it requires many peeling rounds. This happens because a small number of vertices are clustered per round. This can be a bottleneck for large graphs. 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 establish a $(3+\epsilon)$-approximation in a polylogarithmic number of rounds. The algorithm employs a simple step that rejects vertices that are executed in parallel but are "conflicting"; however, we see in our experiments, this seemingly minor coordination step hinders scale-ups in a parallel core setting. In the tutorial of [5], a sketch of a distributed algorithm was presented. This algorithm achieves the same approximation as *KwikCluster*, in a logarithmic number of rounds, in expectation. However, it performs significant redundant work, per iteration, in its effort to detect in parallel which vertices should become cluster centers.

Our contributions   We present *C4* and CLUSTERWILD!, two parallel CC algorithms that in practice outperform the state of the art, both in terms of running time and clustering accuracy. *C4* is a parallel version of *KwikCluster* that uses concurrency control to establish a 3-approximation ratio. CLUSTERWILD! is a simple to implement, coordination-free algorithm that abandons consistency for the benefit of better scaling, while having a provably small loss in the 3 approximation ratio.

C4 achieves a 3 approximation ratio, in a poly-logarithmic number of rounds, by enforcing consistency between concurrently running peeling threads. Consistency is enforced using *concurrency control*, a notion extensively studied for databases transactions, that was recently used to parallelize inherently sequential machine learning algorithms [11].

CLUSTERWILD! is a coordination-free parallel CC algorithm that waives consistency in favor of speed. The cost that we pay is an arbitrarily small loss in CLUSTERWILD!'s accuracy. We show that CLUSTERWILD! achieves a $(3+\epsilon)\mathsf{OPT} + O(\epsilon \cdot n \cdot \log^2 n)$ approximation, in a poly-logarithmic number of rounds, with provable nearly linear speedups. Our main theoretical innovation for CLUSTERWILD! is analyzing the coordination-free algorithm as a serial variant of *KwikCluster* that runs on a "noisy" graph.

In our extensive experimental evaluation, we demonstrate that our algorithms gracefully scale up to graphs with billions of edges. In these large graphs, our algorithms output a valid clustering in less than 5 seconds, on 32 threads, up to an order of magnitude faster than *KwikCluster*. We observe how, not unexpectedly, CLUSTERWILD! is faster than *C4*, and quite surprisingly, abandoning coordination in this parallel setting, only amounts to a 1% of relative loss in the clustering accuracy. Furthermore, we compare against state of the art parallel CC algorithms, showing that we consistently outperform these algorithms in terms of both running time and clustering accuracy.

Notation G denotes a graph with n vertices and m edges. G is complete and only has ± 1 edges. We denote by d_v the positive degree of a vertex, i.e., the number of vertices connected to v with positive edges. Δ denotes the positive maximum degree of G, and $N(v)$ denotes the positive neighborhood of v; moreover, let $C_v = \{v, N(v)\}$. Two vertices u, v are neighbors in G if $u \in N(v)$ and vice versa. We denote by π a permutation of $\{1, \dots, n\}$.

2   Two Parallel Algorithms for Correlation Clustering

The formal definition of correlation clustering is given below.

Correlation Clustering. *Given a graph G on n vertices, partition the vertices into an arbitrary number k of disjoint subsets $\mathcal{C}_1, \ldots, \mathcal{C}_k$ such that the sum of negative edges within the subsets plus the sum of positive edges across the subsets is minimized:*

$$\mathsf{OPT} = \min_{1 \leq k \leq n} \min_{\substack{\mathcal{C}_i \cap \mathcal{C}_j = 0, \forall i \neq j \\ \cup_{i=1}^{k} \mathcal{C}_i = \{1, \ldots, n\}}} \sum_{i=1}^{k} E^-(\mathcal{C}_i, \mathcal{C}_i) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} E^+(\mathcal{C}_i, \mathcal{C}_j)$$

where E^+ and E^- are the sets of positive and negative edges in G.

KwikCluster is a remarkably simple algorithm that approximately solves the above combinatorial problem, and operates as follows. A random vertex v is picked, a cluster C_v is created with v and its positive neighborhood, then the vertices in C_v are peeled from the graph, and this process is repeated until all vertices are clustered. *KwikCluster* can be equivalently executed, as noted by [5], if we substitute the random choice of a vertex per peeling round, with a random order π preassigned to vertices, (see Alg. 1). That is, select a random permutation on vertices, then peel the vertex indexed by $\pi(1)$, and its neighbors. Remove from π the vertices in C_v and repeat this process. Having an order among vertices makes the discussion of parallel algorithms more convenient.

2.1 C_4: Parallel CC using Concurrency Control

Suppose we now wish to run a parallel version of *KwikCluster*, say on two threads: one thread picks vertex v indexed by $\pi(1)$ and the other thread picks u indexed by $\pi(2)$, concurrently. Can both vertices be cluster centers? They can, if and only if they are not neighbors in G. If v and u are connected with a positive edge, then the vertex with the smallest order wins. This is our *concurrency rule no. 1*. Now, assume that v and u are not neighbors in G, and

---

Algorithm 1 *KwikCluster* with π

1: π = a random permutation of $\{1, \ldots, n\}$
2: **while** $V \neq \emptyset$ **do**
3:     select the vertex v indexed by $\pi(1)$
4:     $C_v = \{v, N(v)\}$
5:     Remove clustered vertices from G and π
6: **end while**

---

both v and u become cluster centers. Moreover, assume that v and u have a common, unclustered neighbor, say w: should w be clustered with v, or u? We need to follow what would happen with *KwikCluster* in Alg. 1: w will go with the vertex that has the smallest permutation number, in this case v. This is *concurrency rule no. 2*. Following the above simple rules, we develop C_4, our serializable parallel CC algorithm. Since, C_4 constructs the same clusters as *KwikCluster* (for a given ordering π), it inherits its 3 approximation by design. The above idea of identifying the cluster centers in rounds was first used in [12] to obtain a parallel algorithm for maximal independent set (MIS).

C_4, shown as Alg. 2, starts by assigning a random permutation π to the vertices, it then samples an active set $\mathcal{A}$ of $\frac{n}{\Delta}$ unclustered vertices; this sample is taken from the prefix of π. After sampling $\mathcal{A}$, each of the P threads picks a vertex with the smallest order in $\mathcal{A}$, then checks if that vertex can become a cluster center. We first enforce *concurrency rule no. 1*: adjacent vertices cannot be cluster centers at the same time. C_4 enforces it by making each thread check the neighbors of the vertex, say v, that is picked from $\mathcal{A}$. A thread will check in `attemptCluster` whether its vertex v has any preceding neighbors (according to π) that are cluster centers. If there are none, it will go ahead and label v as cluster center, and proceed with creating a cluster. If a preceding neighbor of v is a cluster center, then v is labeled as not being a cluster center. If a preceding neighbor of v, call it u, has not yet received a label (i.e., u is currently being processed and is not yet labeled as cluster center or not), then the thread processing v, will wait on u to receive a label. The major technical detail is in showing that this wait time is bounded; we show that no more than $O(\log n)$ threads can be in conflict at the same time, using a new subgraph sampling lemma [13]. Since C_4 is serializable, it has to respect *concurrency rule no. 2*: if a vertex u is adjacent to two cluster centers, then it gets assigned to the one with smaller permutation order. This is accomplished in `createCluster`. After processing all vertices in $\mathcal{A}$, all threads are synchronized in bulk, the clustered vertices are removed, a new

active set is sampled, and the same process is repeated until everything has been clustered. In the following section, we present the theoretical guarantees for *C4*.

Algorithm 2 *C4* & CLUSTERWILD!
1: **Input**: G, ϵ
2: clusterID(1) = ... = clusterID(n) = ∞
3: π = a random permutation of $\{1, \ldots, n\}$
4: **while** $V \neq \emptyset$ **do**
5: $\quad \Delta$ = maximum vertex degree in $G(V)$
6: $\quad \mathcal{A}$ = the first $\epsilon \cdot \frac{n}{\Delta}$ vertices in $V[\pi]$.
7: $\quad$ **while** $\mathcal{A} \neq \emptyset$ **do in parallel**
8: $\quad\quad v$ = first element in $\mathcal{A}$
9: $\quad\quad \mathcal{A} = \mathcal{A} - \{v\}$
10: $\quad\quad$ **if** *C4* **then**
11: $\quad\quad\quad$ attemptCluster(v)
12: $\quad\quad$ **else if** CLUSTERWILD! **then**
13: $\quad\quad\quad$ createCluster(v)
14: $\quad\quad$ **end if**
15: $\quad$ **end while**
16: $\quad$ Remove clustered vertices from V and π
17: **end while**
18: **Output:** $\{\text{clusterID}(1), \ldots, \text{clusterID}(n)\}$.

---

createCluster(v):
$\quad$ clusterID(v) = $\pi(v)$
$\quad$ **for** $u \in \Gamma(v) \setminus \mathcal{A}$ **do**
$\quad\quad$ clusterID(u) = min(clusterID(u), $\pi(v)$)
$\quad$ **end for**

---

attemptCluster(v):
$\quad$ **if** clusterID(u) = ∞ **and** isCenter(v) **then**
$\quad\quad$ createCluster(v)
$\quad$ **end if**

---

isCenter(v):
$\quad$ **for** $u \in \Gamma(v)$ **do** //check friends (in order of π)
$\quad\quad$ **if** $\pi(u) < \pi(v)$ **then** //if they precede you, wait
$\quad\quad\quad$ **wait** until clusterID(u) $\neq \infty$ //till clustered
$\quad\quad\quad$ **if** isCenter(u) **then**
$\quad\quad\quad\quad$ **return** 0 //a friend is center, so you can't be
$\quad\quad\quad$ **end if**
$\quad\quad$ **end if**
$\quad$ **end for**
$\quad$ **return** 1 //no earlier friend are centers, so you are

2.2 ClusterWild!: Coordination-free Correlation Clustering

CLUSTERWILD! speeds up computation by ignoring the first concurrency rule. It uniformly samples unclustered vertices, and builds clusters around *all of them*, without respecting the rule that cluster centers cannot be neighbors in G. In CLUSTERWILD!, threads bypass the attemptCluster routine; this eliminates the "waiting" part of *C4*. CLUSTERWILD! samples a set $\mathcal{A}$ of vertices from the prefix of π. Each thread picks the first ordered vertex remaining in $\mathcal{A}$, and using that vertex as a cluster center, it creates a cluster around it. It peels away the clustered vertices and repeats the same process, on the next remaining vertex in $\mathcal{A}$. At the end of processing all vertices in $\mathcal{A}$, all threads are synchronized in bulk, the clustered vertices are removed, a new active set is sampled, and the parallel clustering is repeated. A careful analysis along the lines of [6] shows that the number of rounds (i.e., bulk synchronization steps) is only poly-logarithmic.

Quite unsurprisingly, CLUSTERWILD! is faster than *C4*. Interestingly, abandoning consistency does not incur much loss in the approximation ratio. We show how the error introduced in the accuracy of the solution can be bounded. We characterize this error theoretically, and show that in practice it only translates to only a relative 1% loss in the objective. The main intuition of why CLUSTERWILD! does not introduce too much error is that the chance of two randomly selected vertices being neighbors is small, hence the concurrency rules are infrequently broken.

3 Theoretical Guarantees

In this section, we bound the number of rounds required for each algorithms, and establish the theoretical speedup one can obtain with P parallel threads. We proceed to present our approximation guarantees. We would like to remind the reader that—as in relevant literature—we consider graphs that are complete, signed, and unweighted. The omitted proofs can be found in the Appendix.

3.1 Number of rounds and running time

Our analysis follows those of [12] and [6]. The main idea is to track how fast the maximum degree decreases in the remaining graph at the end of each round.

Lemma 1. *$C4$ and* ClusterWild! *terminate after $O\left(\frac{1}{\epsilon}\log n \cdot \log \Delta\right)$ rounds w.h.p.*

We now analyze the running time of both algorithms under a simplified BSP model. The main idea is that the the running time of each "super step" (i.e., round) is determined by the "straggling" thread (i.e., the one that gets assigned the most amount of work), plus the time needed for synchronization at the end of each round.

Assumption 1. *We assume that threads operate asynchronously within a round, and synchronize at the end of a round. A memory cell can be written/read concurrently by multiple threads. The time spent per round of the algorithm is proportional to the time of the slowest thread. The cost of thread synchronization at the end of each batch takes time $O(P)$, where P is the number of threads. The total computation cost is proportional to the sum of the time spent for all rounds, plus the time spent during the bulk synchronization step.*

Under this simplified model, we show that both algorithms obtain nearly linear speedup, with ClusterWild! being faster than $C4$, precisely due to lack of coordination. Our main tool for analyzing $C4$ is a recent graph-theoretic result (Theorem 1 in [13]), which guarantees that if one samples an $O(n/\Delta)$ subset of vertices in a graph, the sampled subgraph has a connected component of size at most $O(\log n)$. Combining the above, in the appendix we show the following result.

Theorem 2. *The theoretical running time of $C4$, on P cores and $\epsilon = 1/2$, is upper bounded by*

$$O\left(\left(\frac{m + n \log^2 n}{P} + P\right) \log n \cdot \log \Delta\right)$$

as long as the number of cores P is smaller than $\min_i \frac{n_i}{2\Delta_i}$, where $\frac{n_i}{2\Delta_i}$ is the size of the batch in the i-th round of each algorithm. The running time of ClusterWild! *on P cores is upper bounded by*

$$O\left(\left(\frac{m + n}{P} + P\right) \frac{\log n \cdot \log \Delta}{\epsilon^2}\right)$$

for any constant $\epsilon > 0$.

3.2 Approximation ratio

We now proceed with establishing the approximation ratios of $C4$ and ClusterWild!.

3.2.1 $C4$ is serializable

It is straightforward that $C4$ obtains precisely the same approximation ratio as *KwikCluster*. One has to simply show that for any permutation π, *KwikCluster* and $C4$ will output the same clustering. This is indeed true, as the two simple concurrency rules mentioned in the previous section are sufficient for $C4$ to be equivalent to *KwikCluster*.

Theorem 3. *$C4$ achieves a 3 approximation ratio, in expectation.*

3.2.2 ClusterWild! as a serial procedure on a noisy graph

Analyzing ClusterWild! is a bit more involved. Our guarantees are based on the fact that ClusterWild! can be treated *as if* one was running a peeling algorithm on a "noisy" graph. Since adjacent active vertices can still become cluster centers in ClusterWild!, one can view the edges between them as "deleted," by a somewhat unconventional adversary. We analyze this new, noisy graph and establish our theoretical result.

Theorem 4. CLUSTERWILD! *achieves a* $(3 + \epsilon) \cdot \mathsf{OPT} + O(\epsilon \cdot n \cdot \log^2 n)$ *approximation, in expectation.*

We provide a sketch of the proof, and delegate the details to the appendix. Since CLUSTERWILD! ignores the edges among active vertices, we treat these edges as deleted. In our main result, we quantify the loss of clustering accuracy that is caused by ignoring these edges. Before we proceed, we define *bad triangles*, a combinatorial structure that is used to measure the clustering quality of a peeling algorithm.

Definition 1. *A bad triangle in G is a set of three vertices, such that two pairs are joined with a positive edge, and one pair is joined with a negative edge. Let $\mathcal{T}_b$ denote the set of bad triangles in G.*

To quantify the cost of CLUSTERWILD!, we make the below observation.

Lemma 5. *The cost of any greedy algorithm that picks a vertex v (irrespective of the sampling order), creates $\mathcal{C}_v$, peels it away and repeats, is equal to the number of bad triangles adjacent to each cluster center v.*

Lemma 6. *Let $\hat{G}$ denote the random graph induced by deleting all edges between active vertices per round, for a given run of CLUSTERWILD!, and let τ_{new} denote the number of additional bad triangles that $\hat{G}$ has compared to G. Then, the expected cost of CLUSTERWILD! can be upper bounded as*

$$\mathbb{E}\left\{\sum_{t \in \mathcal{T}_b} \mathbf{1}_{\mathcal{P}_t} + \tau_{new}\right\},$$

where $\mathcal{P}_t$ is the event that triangle t, with end points i, j, k, is bad, and at least one of its end points becomes active, while t is still part of the original unclustered graph.

We provide the proof for the above two lemmas in the Appendix. We continue with bounding the second term $\mathbb{E}\{\tau_{\text{new}}\}$ in the bound of Lemma 6, by considering the number of new bad triangles $\tau_{\text{new},i}$ created at each round i (in the following $\mathcal{A}_i$, denotes the set of active vertices at round i):

$$\mathbb{E}\{\tau_{\text{new},i}\} \leq \sum_{(u,v) \in E_i^+} \mathbb{P}(u,v \in \mathcal{A}_i) \cdot |N_i(u) \cup N_i(v)| \leq \sum_{(u,v) \in E_i^+} \left(\frac{\epsilon}{\Delta_i}\right)^2 \cdot 2 \cdot \Delta_i \quad \leq \quad 2 \cdot \epsilon^2 \cdot \frac{E_i}{\Delta_i} \quad \leq \quad 2 \cdot \epsilon^2 \cdot n$$

where E_i^+ is the set of remaining positive and $N_i(v)$ the neighborhood of vertex v at round i, the second inequality is due to the fact that the size of the neighborhoods is upper bounded by Δ_i, the maximum positive degree at round i, and the probability bound is true since we are sampling $\frac{n_i}{\Delta_i}$ vertices without replacement from a total of n_i, the number of unclustered vertices at round i; the final inequality is true since $E_i \leq n \cdot \Delta_i$. Using the result that CLUSTERWILD! terminates after at most $O(\frac{1}{\epsilon} \log n \log \Delta)$ rounds, we get that[1]

$$\mathbb{E}\{\tau_{\text{new}}\} \leq O(\epsilon \cdot n \cdot \log^2 n).$$

We are left to bound

$$\mathbb{E}\left\{\sum_{t \in \mathcal{T}_b} \mathbf{1}_{\mathcal{P}_t}\right\} = \sum_{t \in \mathcal{T}_b} p_t.$$

To do that we use the following lemma.

Lemma 7. *If p_t satisfies*

$$\forall e, \quad \sum_{t:e \subset t \in \mathcal{T}_b} \frac{p_t}{\alpha} \leq 1,$$

then,

$$\sum_{t \in \mathcal{T}_b} p_t \leq \alpha \cdot OPT.$$

---

[1]We skip the constants to simplify the presentation; however they are all smaller than 10.

Proof. Let $\mathcal{B}_*$ be one (of the possibly many) sets of edges that attribute a $+1$ in the cost of an optimal algorithm. Then,

$$\mathsf{OPT} = \sum_{e \in \mathcal{B}^*} 1 \geq \sum_{e \in \mathcal{B}^*} \sum_{t:e \subset t \in \mathcal{T}_b} \frac{p_t}{\alpha} = \sum_{t \in \mathcal{T}_b} \underbrace{|B_* \cap t|}_{\geq 1} \frac{p_t}{\alpha} \geq \sum_{t \in \mathcal{T}_b} \frac{p_t}{\alpha}.$$

$\square$

Now, as with [6], we will simply have to bound the expectation of the bad triangles, adjacent to an edge (u, v):

$$\sum_{t:\{u,v\} \subset t \in \mathcal{T}_b} \mathbf{1}_{\mathcal{P}_t}.$$

Let $\mathcal{S}_{u,v} = \bigcup_{\{u,v\} \subset t \in \mathcal{T}_b} t$ be the union of the sets of nodes of the bad triangles that contain both vertices u and v. Observe that if some $w \in S \backslash \{u, v\}$ becomes active before u and v, then a cost of 1 (i.e., the cost of the bad triangle $\{u, v, w\}$) is incurred. On the other hand, if either u or v, or both, are selected as pivots in some round, then $\mathcal{C}_{u,v}$ can be as high as $|S| - 2$, i.e., at most equal to all bad triangles containing the edge $\{u, v\}$. Let $A_{uv} = \{u \text{ or } v \text{ are activated before any other vertices in } S_{u,v}\}$. Then,

$$\mathbb{E}\left[\mathcal{C}_{u,v}\right] = \mathbb{E}\left[\mathcal{C}_{u,v} \mid A_{u,v}\right] \cdot \mathbb{P}(A_{u,v}) + \mathbb{E}\left[\mathcal{C}_{u,v} \mid A_{u,v}^C\right] \cdot \mathbb{P}(A_{u,v}^C)$$
$$\leq 1 + (|S| - 2) \cdot \mathbb{P}(\{u, v\} \cap \mathcal{A} \neq \emptyset | \mathcal{S} \cap \mathcal{A} \neq \emptyset)$$
$$\leq 1 + 2|S| \cdot \mathbb{P}(v \cap \mathcal{A} \neq \emptyset | \mathcal{S} \cap \mathcal{A} \neq \emptyset)$$

where the last inequality is obtained by a union bound over u and v. We now bound the following probability:

$$\mathbb{P}\{v \cap \mathcal{A} \neq \emptyset | \mathcal{S} \cap \mathcal{A} \neq \emptyset\} = \frac{\mathbb{P}\{v \in \mathcal{A}\} \cdot \mathbb{P}\{\mathcal{S} \cap \mathcal{A} \neq \emptyset | v \in \mathcal{A}\}}{\mathbb{P}\{\mathcal{S} \cap \mathcal{A} \neq \emptyset\}} = \frac{\mathbb{P}\{v \in \mathcal{A}\}}{\mathbb{P}\{\mathcal{S} \cap \mathcal{A} \neq \emptyset\}} = \frac{\mathbb{P}\{v \in \mathcal{A}\}}{1 - \mathbb{P}\{\mathcal{S} \cap \mathcal{A} = \emptyset\}}.$$

Observe that $\mathbb{P}\{v \in \mathcal{A}\} = \frac{\epsilon}{\Delta}$, hence we need to upper bound $\mathbb{P}\{\mathcal{S} \cap \mathcal{A} = \emptyset\}$. The probability, per round, that no positive neighbors in $\mathcal{S}$ become activated is upper bounded by

$$\frac{\binom{n-|\mathcal{S}|}{P}}{\binom{n}{P}} = \prod_{t=1}^{|\mathcal{S}|}\left(1 - \frac{P}{n - |\mathcal{S}| + t}\right) \leq \left(1 - \frac{P}{n}\right)^{|\mathcal{S}|}$$
$$= \left[\left(1 - \frac{P}{n}\right)^{n/P}\right]^{|\mathcal{S}|n/P} \leq \left(\frac{1}{e}\right)^{|\mathcal{S}|n/P}.$$

Hence, we obtain the following bound

$$|\mathcal{S}|\mathbb{P}\{v \cap \mathcal{A} \neq \emptyset | \mathcal{S} \cap \mathcal{A} \neq \emptyset\} \leq \frac{\epsilon \cdot |\mathcal{S}|/\Delta}{1 - e^{-\epsilon \cdot |\mathcal{S}|/\Delta}}.$$

We now know that $|\mathcal{S}| \leq 2 \cdot \Delta + 2$ and also $\epsilon \leq 1$. Then,

$$0 \leq \epsilon \cdot \frac{|\mathcal{S}|}{\Delta} \leq \epsilon \cdot \left(2 + \frac{2}{\Delta}\right) \leq 4.$$

Hence, we have

$$\mathbb{E}(\mathcal{C}_{u,v}) \leq 1 + 2 \cdot \frac{4\epsilon}{1 - \exp\{-4\epsilon\}}.$$

The overall expectation is then bounded by

$$\mathbb{E}\left\{\sum_{t \in \mathcal{T}_b} \mathbf{1}_{\mathcal{P}_t} + \tau_{\mathrm{new}}\right\} \leq \left(1 + 2 \cdot \frac{4 \cdot \epsilon}{1 - e^{-4 \cdot \epsilon}}\right) \cdot \mathsf{OPT} + O(\epsilon \cdot n \cdot \log^2 n) \leq (3 + \epsilon) \cdot \mathsf{OPT} + O(\epsilon \cdot n \cdot \log^2 n)$$

which establishes our approximation ratio for CLUSTERWILD!.

3.3 BSP Algorithms as a Proxy for Asynchronous Algorithms

7

We would like to note that the analysis under the BSP model can be a useful proxy for the performance of completely asynchronous variants of our algorithms. Specifically, see Alg. 3, where we remove the synchronization barriers.

The only difference between the asynchronous execution in Alg. 3, compared to Alg. 2, is the complete lack of bulk synchronization, at the end of the processing of each active set $\mathcal{A}$. Although the analysis of the BSP variants of the algorithms is tractable, unfortunately analyzing precisely the speedup of the asynchronous $C4$ and the approximation guarantees for the asynchronous CLUSTERWILD! is challenging. However, in our experimental section we test the completely asynchronous algorithms against the

---

Algorithm 3 $C4$ & CLUSTERWILD!
(asynchronous execution)

1: **Input**: G
2: clusterID$(1) = \ldots =$ clusterID$(n) = \infty$
3: $\pi = $ a random permutation of $\{1, \ldots, n\}$
4: **while** $V \neq \emptyset$ **do**
5:     $v = $ first element in V
6:     $V = V - \{v\}$
7:     **if** $C4$ **then** // concurrency control
8:        attemptCluster(v)
9:     **else if** CLUSTERWILD! **then** // coordination free
10:        createCluster(v)
11:    **end if**
12:    Remove clustered vertices from V and π
13: **end while**
14: **Output:** $\{$clusterID$(1), \ldots,$ clusterID$(n)\}$.

BSP algorithms of the previous section, and observe that they perform quite similarly both in terms of accuracy of clustering, and running times.

4   Related Work

Correlation clustering was formally introduced by Bansal et al. [14]. In the general case, minimizing disagreements is NP-hard and hard to approximate within an arbitrarily small constant (APX-hard) [14, 15]. 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 [16].

In the case of complete unweighted graphs, a long series of results establishes a 2.5 approximation via a rounded linear program (LP) [10]. A recent result establishes a 2.06 approximation using an elegant rounding to the same LP relaxation [17]. 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.

Variations of the cost metric for CC change the algorithmic landscape: maximizing agreements (the dual measure of disagreements) [14, 18, 19], or maximizing the difference between the number of agreements and disagreements [20, 21], come with different hardness and approximation results. There are also several variants: chromatic CC [22], overlapping CC [23], or CC with small number of clusters and added constraints that are suitable for biology applications [24].

The way $C4$ finds the cluster centers can be seen as a variation of the MIS algorithm of [12]; the main difference is that in our case, we "passively" detect the MIS, by locking on memory variables, and by waiting on preceding ordered threads. This means, that a vertex only "pushes" its cluster ID and status (cluster center/clustered/unclustered) to its neighbors, versus "pulling" (or asking) for its neighbors' cluster status. This saves a substantial amount of computational effort. A sketch of the idea of using parallel MIS algorithms for CC was presented in [5], where the authors suggest using Luby's algorithm for finding an MIS, and then using the MIS vertices as cluster centers. However, a closer look on this approach reveals that there is fundamentally more work need to be done to cluster the vertices.

5   Experiments

Our parallel algorithms were all implemented in Scala—we defer a full discussion of the implementation details to Appendix C. We ran all our experiments on Amazon EC2's r3.8xlarge (32 vCPUs, 244Gb memory)

instances, using 1-32 threads. The real graphs listed in Table 1 were each tested with 100 different random π

Graph	# vertices	# edges	Description
DBLP-2011	986,324	6,707,236	2011 DBLP co-authorship network [25, 26, 27].
ENWiki-2013	4,206,785	101,355,853	2013 link graph of English part of Wikipedia [25, 26, 27].
UK-2005	39,459,925	921,345,078	2005 crawl of the .uk domain [25, 26, 27].
IT-2004	41,291,594	1,135,718,909	2004 crawl of the .it domain [25, 26, 27].
WebBase-2001	118,142,155	1,019,903,190	2001 crawl by WebBase crawler [25, 26, 27].

Table 1: Graphs used in the evaluation of our parallel algorithms.

orderings. We measured the runtimes, speedups (ratio of runtime on 1 thread to runtime on p threads), and objective values obtained by our parallel algorithms. For comparison, we also implemented the algorithm presented in [6], which we denote as CDK for short[2]. Values of $\epsilon = 0.1, 0.5, 0.9$ were used for $C4$ BSP, CLUSTERWILD! BSP and CDK. In the interest of space, we present only representative plots of our results; full results are given in our appendix.

5.1 Runtimes

$C4$ and CLUSTERWILD! are initially slower than serial, due to the overheads required for atomic operations in the parallel setting. However, all our parallel algorithms outperform serial *KwikCluster* with 3-4 threads. As more threads are added, the asychronous variants become faster than their BSP counterparts as there are no synchronization barrriers. The difference between BSP and asychronous variants is greater for smaller ϵ. CLUSTERWILD! is also always faster than $C4$ since there are no coordination overheads.

5.2 Speedups

The asynchronous algorithms are able to achieve a speedup of 13-15x on 32 threads. The BSP algorithms have a poorer speedup ratio, but nevertheless achieve 10x speedup with $\epsilon = 0.9$.

5.3 Synchronization rounds

The main overhead of the BSP algorithms lies in the need for synchronization rounds. As ϵ increases, the amount of synchronization decreases, and with $\epsilon = 0.9$, our algorithms have less than 1000 synchronization rounds, which is small considering the size of the graphs and our multicore setting.

5.4 Blocked vertices

Additionally, $C4$ incurs an overhead in the number of vertices that are blocked waiting for earlier vertices to complete. We note that this overhead is extremely small in practice—on all graphs, less than 0.2% of vertices are blocked. On the larger and sparser graphs, this drops to less than 0.02% (i.e., 1 in 5000) of vertices.

5.5 Objective value

By design, the $C4$ algorithms also return the same output (and thus objective value) as serial *KwikCluster*. We find that CLUSTERWILD! BSP is at most 1% worse than serial across all graphs and values of ϵ. The behavior of asynchronous CLUSTERWILD! worsens as threads are added, reaching 15% worse than serial for one of the graphs. Finally, on the smaller graphs we were able to test CDK on, we find that CDK returns a worse median objective value than both CLUSTERWILD! variants.

---

[2] CDK was only tested on the smaller graphs of DBLP-2011 and ENWiki-2013, because CDK was prohibitively slow, often 2-3 orders of magnitude slower than $C4$, CLUSTERWILD!, and even serial *KwikCluster*.
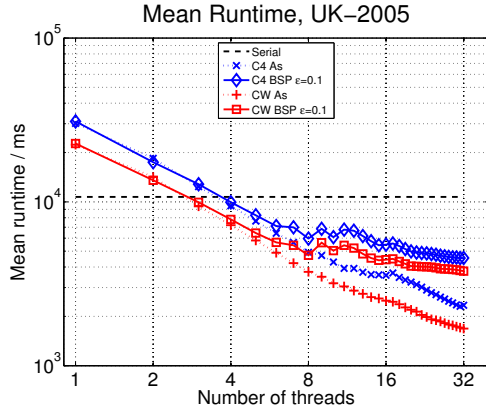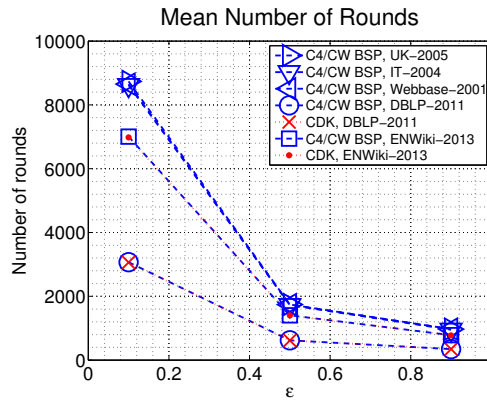
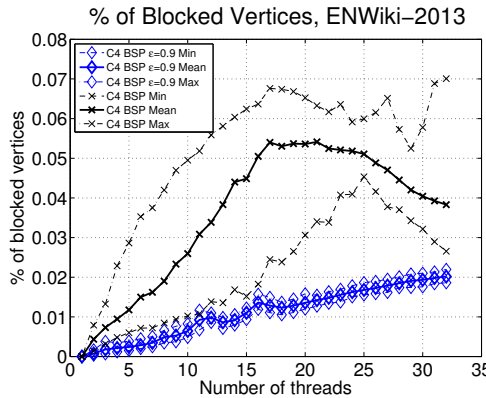(a) Mean runtimes, UK-2005, $\epsilon = 0.1$
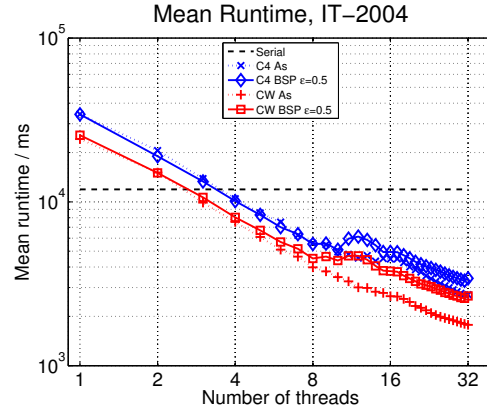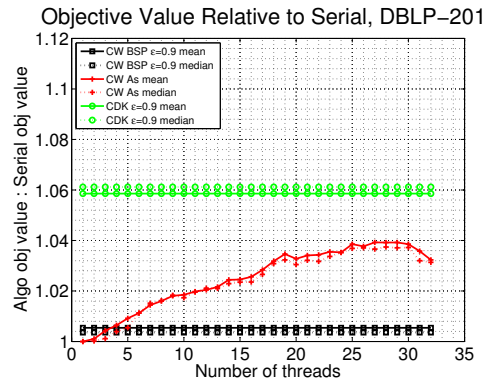
(b) Mean runtimes, IT-2004, $\epsilon = 0.5$
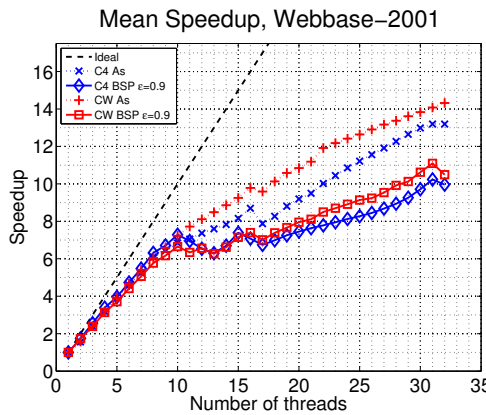
(c) Mean speedup, WebBase, $\epsilon = 0.9$

(d) Mean number of synchronization rounds for BSP algorithms

(e) Percent of blocked vertices for C4, ENWiki-2013. BSP run with $\epsilon = 0.9$.

(f) Median objective values, DBLP-2011. CW BSP and CDK run with $\epsilon = 0.9$

Figure 2: In the above figures, 'CW' is short for CLUSTERWILD!, 'BSP' is short for the bulk-synchronous variants of the parallel algorithms, and 'As' is short for the asynchronous variants.

6   Conclusions and Future Directions

We presented two parallel algorithms for correlation clustering that admit provable nearly linear speedups and approximation ratios. Our algorithms can cluster billion-edge graphs in under 5 seconds on 32 cores, while achieving a 15× speedup. The two approaches complement each other: when *C4* is fast relative to CLUSTERWILD!, we may prefer it for its guarantees of accuracy; and when CLUSTERWILD! is accurate relative to *C4*, we may prefer it for its speed.

Both *C4* and CLUSTERWILD! are well-suited for a distributed setup since they run for at most a polylogarithmic number of rounds. In the future, we intend to implement our algorithms in a distributed environment, where synchronization and communication often account for the highest cost.

Acknowledgments

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] 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.

[12] Guy E Blelloch, Jeremy T Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 308–317. ACM, 2012.

[13] Michael Krivelevich. The phase transition in site percolation on pseudo-random graphs. *arXiv preprint arXiv:1404.5731*, 2014.

[14] 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.

[15] 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.

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

[17] Shuchi Chawla, Konstantin Makarychev, Tselil Schramm, and Grigory Yaroslavtsev. Near optimal LP rounding algorithm for correlation clustering on complete and complete k-partite graphs. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, STOC '15, pages 219–228, 2015.

[18] Chaitanya Swamy. Correlation clustering: maximizing agreements via semidefinite programming. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 526–527. Society for Industrial and Applied Mathematics, 2004.

[19] Ioannis Giotis and Venkatesan Guruswami. Correlation clustering with a fixed number of clusters. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1167–1176. ACM, 2006.

[20] Moses Charikar and Anthony Wirth. Maximizing quadratic programs: extending grothendieck's inequality. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 54–60. IEEE, 2004.

[21] Noga Alon, Konstantin Makarychev, Yury Makarychev, and Assaf Naor. Quadratic forms on graphs. *Inventiones mathematicae*, 163(3):499–522, 2006.

[22] Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. Chromatic correlation clustering. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1321–1329. ACM, 2012.

[23] Francesco Bonchi, Aristides Gionis, and Antti Ukkonen. Overlapping correlation clustering. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 51–60. IEEE, 2011.

[24] Gregory J Puleo and Olgica Milenkovic. Correlation clustering with constrained cluster sizes and extended weights bounds. *arXiv preprint arXiv:1411.0547*, 2014.

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

[26] 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.

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

A Proofs of Theoretical Guarantees

A.1 Number of rounds for $C4$ and ClusterWild!

Lemma 1. *$C4$ and* CLUSTERWILD! *terminate after $O\left(\frac{1}{\epsilon} \log n \cdot \log \Delta\right)$ rounds w.h.p.*

Proof. We split our proof in two parts.

For CLUSTERWILD!, we wish to upper bound the probability

$$q_t = \mathbb{P}\left\{v \text{ not clustered by round } i + t \,\middle|\, \deg_{i+j}(v) \geq \frac{\Delta_i}{2}, 1 \leq j \leq t\right\}.$$

Observe that the above event happens either if no neighbors of v become activated by round $i + t$, or if v itself does not become activated. Hence, q_t can be upper bounded by the probability that no neighbors of v become activated by round $i + t$.

In the following, let d_{i+j} denote the degree of vertex v at roudn $i + j$; for simplicity we drop the round indices on n and P. The probability, per round, that no neighbors of v become activated is equal to[3]

$$\frac{\binom{n-d_{i+j}}{P}}{\binom{n}{P}} = \frac{(n-P)!}{(n-P-d_{i+j})!} \cdot \frac{(n-d_{i+j})!}{n!}$$

$$= \frac{\prod_{t=1}^{d_{i+j}}(n-d_{i+1}+t-P)}{\prod_{t=1}^{d_{i+j}}(n-d_{i+1}+t)} = \prod_{t=1}^{d_{i+j}} \frac{n-d_{i+1}+t-P}{n-d_{i+1}+t}$$

$$= \prod_{t=1}^{d_{i+j}} \left(1 - \frac{P}{n-d_{i+1}+t}\right) \leq \left(1 - \frac{P}{n}\right)^{d_{i+j}}$$

$$\leq \left(1 - \frac{\epsilon}{\Delta_i}\right)^{\Delta_i/2} = \left[\left(1 - \frac{\epsilon}{\Delta_i}\right)^{\Delta_i/\epsilon}\right]^{\epsilon/2} \leq e^{-\epsilon/2}.$$

where the last inequality is due to the fact that

$$(1-x)^{1/x} < e^{-1} \text{ for all } x \leq 1.$$

Therefore, the probability of vertex v failing to be clustered after t rounds is at most $q_t \leq e^{-t \cdot \epsilon/2}$. Hence, we have that for any round i, the probability that any vertex has degree more than $\Delta_i/2$ after t rounds is at most $n \cdot e^{-t \cdot \epsilon/2}$, due to a simple union bound. If we want that that probability to be smaller than δ, then

$$n \cdot e^{-t \cdot \epsilon/2} < \delta \Leftrightarrow \ln n - t \cdot \epsilon/2 < \ln(\delta) \Leftrightarrow t > \frac{2}{\epsilon} \cdot \ln(n/\delta)$$

Hence, with probability $1-\delta$, after $\frac{2}{\epsilon} \cdot \ln(n/\delta)$ rounds either all nodes of degree greater than $\Delta/2$ are clustered, or the maximum degree is decreased by half. Applying this argument $\log \Delta$ times yields the result, as the maximum degree of the remaining graph becomes 1.

For $C4$ the proof follows simply from the analogous proof of [12]. Consider any round of the algorithm, and break it into k steps (each step, for each vertex in $\mathcal{A}$ that becomes a cluster center). Let v be a vertex that has degree at most $\Delta/2$, and is not active. During step 1 of round 1, the probability that v is not adjacent to $\pi(1)$ is at most $1 - \frac{\epsilon}{2n}$. If v is not selected at step 1, then during step 2 of round 1, the probability that v is not adjacent to the next cluster center is again at most $1 - \frac{\epsilon}{2n}$. After processing all vertices in $\mathcal{A}$, during the first round, either v was clustered, or its degree became strictly less than $\Delta/2$, or the probability that neither of the previous happened is at most $(1 - \frac{\epsilon}{2n})^{\frac{\epsilon\Delta}{n}} \leq 1 - \epsilon/2$. It is easy to see that after $O(\frac{1}{\epsilon} \log n)$ rounds vertex v will have either been clustered or its degree would be smaller than $\Delta/2$. Union bounding for n vertices and all rounds, we get that the max degree of the remaining graph gets halved after $O(\frac{1}{\epsilon} \log n)$ rounds, hence the total number of rounds needed is at most $O(\frac{1}{\epsilon} \log n \log \Delta)$, with high probability. □

---

[3]This follows from a simple calculation on the pdf of the hypergeometric distribution.

A.2 Running times

In this section, we prove the running time theorem for our Algorithms. We first present the following recent graph-theoretic result.

Theorem A.1 (Theorem 1 in [13])**.** *Let G be an undirected graph on n vertices, with maximum vertex degree Δ. Let us sample each vertex independently with probability $p = \frac{1-\epsilon}{\Delta}$ and define as G' the induced subgraph on the activated vertices. Then, the largest connected component of the resulting graph G' has size at most $O(\frac{1}{\epsilon^2} \log n)$ with high probability.*

To apply Theorem A.1, we first need to convert it into a result for sampling without replacement (instead of i.i.d. sampling).

Lemma A.2. *Let us define two sequences of binary random variables $\{X_i\}_{i=1}^n, \{Y_i\}_{i=1}^n$. The first sequence comprises n i.i.d. Bernoulli random variables with probability p, and the second sequence a random subset of B random variables is set to 1 without replacement, where B is integer that satisfies*

$$(n+1) \cdot p - 1 \leq B < (n+1) \cdot p.$$

Let us now define $\rho_X = \mathbb{P}\left(f(X_1, \ldots, X_n) > C\right)$ for some f (in our case this will be the largest connected component of a subgraph defined on the sampled vertices) and some number C, and similarly define ρ_Y. Let us further assume that we have an upper bound on the above probability $\rho_X \leq \delta$. Then, $\rho_Y \leq n \cdot \delta$.

Proof. By expanding ρ_X using law of total probability we have

$$\rho_X = \sum_{b=0}^n \mathbb{P}\left(f(X_1, \ldots, X_n) > C \,\middle|\, \sum_{i=1}^n X_i = b\right) \cdot \mathbb{P}\left(\sum_{i=1}^n X_i = b\right)$$
$$= \sum_{b=0}^n q_b \cdot \mathbb{P}\left(\sum_{i=1}^n X_i = b\right) \tag{1}$$

where q_b is the probability that $f(X_1, \ldots, X_n) > C$ given that a uniformly random subset of b variables was set to 1. Moreover, we have

$$\rho_Y = \sum_{b=0}^n \mathbb{P}\left(f(Y_1, \ldots, Y_n) > C \,\middle|\, \sum_{i=1}^n Y_i = b\right) \cdot \mathbb{P}\left(\sum_{i=1}^n Y_i = b\right)$$
$$\overset{(i)}{=} \sum_{b=0}^n q_b \cdot \mathbb{P}\left(\sum_{i=1}^n Y_i = b\right)$$
$$\overset{(ii)}{=} q_B \cdot 1 \tag{2}$$

where (i) comes form the fact that $\mathbb{P}\left(f(Y_1, \ldots, Y_n) > C \,\middle|\, \sum_{i=1}^n Y_i = b\right)$ is the same as the probability that that $f(X_1, \ldots, X_n) > C$ given that a uniformly random subset of b variables where set to 1, and (ii) comes from the fact that since we sample without replacement in Y, we have that $\sum_i^n Y_i = B$ always.

If we just keep the $b = B$ term in the expansion of ρ_X we get

$$\rho_X = \sum_{b=0}^n q_b \cdot \mathbb{P}\left(\sum_{i=1}^n X_i = b\right) \geq q_B \cdot \mathbb{P}\left(\sum_{i=1}^n X_i = B\right) = \rho_Y \cdot \mathbb{P}\left(\sum_{i=1}^n X_i = B\right) \tag{3}$$

since all terms in the sum are non-negative numbers. Moreover, since X_is are Bernoulli random variables, then $\sum_{i=1}^n X_i$ is Binomially distributed with parameters n and p. We know that the maximum of the Binomial pmf with parameters n and p occurs at $\mathbb{P}\left(\sum_i X_i = B\right)$ where B is the integer that satisfies $(n+1) \cdot p - 1 \leq$

$B < (n + 1) \cdot p$. Furthermore we know that the maximum value of the Binomial pmf cannot be less than $\frac{1}{n}$, that is

$$\mathbb{P}\left(\sum_{i=1}^{n} X_i = B\right) \geq \frac{1}{n}. \tag{4}$$

If we combine (3) and (4) we get $\rho_X \geq \rho_Y / n \Leftrightarrow \rho_Y \leq n \cdot \delta$. $\qquad \square$

Corollary A.3. *Let G be an undirected graph on n vertices, with maximum vertex Δ. Let us sample $\epsilon \cdot \frac{n}{\Delta}$ vertices without replacement, and define as G' the induced subgraph on the activated vertices. Then, the largest connected component of the resulting graph G' has size at most*

$$O\left(\frac{\log n}{\epsilon^2}\right)$$

with high probability.

We use this in the proof of our theorem that follows.

Theorem 2. *The theoretical running time of C4, on P cores and $\epsilon = 1/2$, is upper bounded by*

$$O\left(\left(\frac{m + n \log^2 n}{P} + P\right) \log n \cdot \log \Delta\right)$$

as long as the number of cores P is smaller than $\min_i \frac{n_i}{2\Delta_i}$, where $\frac{n_i}{2\Delta_i}$ is the size of the batch in the i-th round of each algorithm. The running time of CLUSTERWILD! on P cores is upper bounded by

$$O\left(\left(\frac{m + n}{P} + P\right) \frac{\log n \cdot \log \Delta}{\epsilon^2}\right)$$

for any constant $\epsilon > 0$.

Proof. We start with analyzing *C4*, as the running time of CLUSTERWILD! follows from a similar, and simpler analysis. Observe, that we operate on Bulk Synchronous Parallel model: we sample a batch of vertices, P cores asynchronously process the vertices in the batch, and once the batch is empty there is a bulk synchronization step. The computational effort spent by *C4* can be split in three parts: i) computing the maximum degree, ii) creating the clusters, per batch, iii) syncronizing at the end of each batch.

Computing Δ and synchronizing cost Computing Δ_i at the beginning of each batch, can be implemented in time $\frac{m_i}{P} + \log P$, where each thread picks n_i/P vertices and computes locally their degrees, and inserts it to a sorted data structure (*e.g.*, a B-tree that admits parallel operations), and then we get the largest item in logarithmic time. Moreover, the third part of the computation, i.e., synchronization among cores, can be done in $O(P)$. A little more involved argument is needed for establishing the running time of the second part, where the algorithms create the clusters.

Clustering cost For a single vertex v sampled by a thread, the time required by the thread to process that vertex is the sum of the time needed to 1) wait inside the attemptCluster for preceding neighbors (by the order of π), 2) "send" its $\pi(v)$ to its neighbors, if v is a cluster center, 3) if v is a cluster center, then for each u neighbors it will attempt to update clusterID(u); however, this thread potentially competes with other threads that are attempting to write in clusterID(u) at the same time.

Using Corollary A.3, we can show that no more than $O(\log n)$ threads compete with each other at the same time, with high probability. Observe, that in our sampling scheme of batches of vertices, we are taking the first $B_i = \frac{\epsilon}{\Delta_i} \cdot n_i$ elements of a random prefix π. This is equivalent to sampling B_i vertices without replacement from the graph G_i of the current round. The result in Corollary A.3, asserts that the largest connected component in the sampled subgraph is at most $O(\log n)$, with high probability. This directly

implies that a thread cannot be waiting for more than $O(\log n)$ other threads inside `attemptCluster(v)`. Therefore, the time spent by each thread to wait on other threads in `attemptCluster(v)` is upper bounded by the number of maximum threads that it can be neighbors with (which assuming ϵ is set to $1/2$) is at most $O(\log n)$, times the time it takes each of these threads to be done with their execution, which is at most $\Delta_i \log n$ (even assuming the worst case conflict pattern when updating at most Δ_i entries in the clusterID array). Hence, for *C4* the processing time of a single vertex is upper bounded by $O(\Delta_i \cdot \log^2 n)$.

Job allocation Now, observe that when each thread is done processing vertex, it picks the next vertex from $\mathcal{A}$ (if $\mathcal{A}$ is not empty). This process essentially models a classical greedy task allocation to cores, that leads to a 2 approximation in terms of the optimum weight allocation; here the optimum allocation leads to a max weight among cores that is at most equal to $\max(\Delta_i, B_i \Delta_i / P)$. This implies that the running time on P asynchronous threads of a single batch, is upperbounded by

$$O \left(\max \left(\Delta_i \log n, \frac{B_i \Delta_i \log^2 n}{P} \right) \right) = O \left(\max \left(\Delta_i \log n, \frac{n_i \log^2 n}{P} \right) \right).$$

Assuming, that the number of cores, is always less than the batch size (a reasonable assumption, as more cores, would not lead to further benefits), we obtain that the time for a single batch is

$$O \left(\frac{E_i}{P} + \frac{n_i \log^2 n}{P} + P \right).$$

Observe that a difference in CLUSTERWILD!, is that waiting is avoided, hence, the running time, per batch of CLUSTERWILD! is

$$O \left(\frac{E_i}{P} + \frac{n_i}{P} + P \right).$$

Multiplying the above, with the number of rounds given by Lemma 1, we obtain the theorem.

□

A.3   Approximation Guarantees

One can view the execution of ClusterWild! on G as having *KwikCluster* run on a "noisy version" of G. A main issue is that *KwikCluster* never allows two neighbors in the original graph to become cluster centers. Hence, since ClusterWild! ignores these edges among active vertices, one can view these edges as "adverserially" deleted. The major technical contribution of this work is to quantify how these "ignored" edges affect the quality of the output solution. The following simple lemma presented in our main text, is useful in quantifying the cost of the output clustering for *any* peeling algorithm.

Lemma 5. *The cost of any greedy algorithm that picks a vertex v (irrespective of the sampling order), creates $\mathcal{C}_v$, peels it away and repeats, is equal to the number of bad triangles adjacent to each cluster center v.*

Proof. Consider the first step of the algorithm, for simplicity, and without loss of generality. Let us define as T_{in} the number of vertex pairs inside $\mathcal{C}_v$ that are not neighbors (i.e., they are joined by a negative edge). Moreover, let T_{out} denote the number of vertices outside $\mathcal{C}_v$ that are neighbors with vertices inside $\mathcal{C}_v$. Then, the number of disagreements (i..e, number of misplaced pairs of vertices) generated by cluster $\mathcal{C}_v$, is equal to $T_{\text{in}} + T_{\text{out}}$.

Observe that all the T_{in} edges are negative, and all T_{out} are positive ones. Let for example (u, w) be one of the T_{in} negative edges inside $\mathcal{C}_v$, hence both u, w belong to $\mathcal{C}_v$ (i.e., are neighbors with v). Then, (u, v, w) forms a *bad triangle*. Similarly, for every edge that is incident to a vertex in $\mathcal{C}_v$, with one end point say $u' \in \mathcal{C}_v$ and one $w' \in V \setminus v$, the triangle formed by (v, u', w'), is also a bad triangle.

Hence, all edges that are accounted for in the final cost of the algorithm (i..e, total number of disagreements) are equal to the $T_{\text{in}} + T_{\text{out}}$ bad triangles that include these edges and each cluster center per round. □

Let us now consider the set of all cluster centers generated by ClusterWild!; call these vertices $\mathcal{C}_{\mathrm{CW}}$. Then, consider the graph G' that is generated by deleting all edges between $\mathcal{C}_{\mathrm{CW}}$. Observe that this is a random graph, since the set of edges deleted depends on the specific random sampling that is performed in ClusterWild!. We will use the following simple technical proposition to quantify how many more bad triangles G' has compared to G.

Proposition A.4. *Given any graph G with positive and negative edges, then let us obtain a graph G_e where we have removed a single edge, e from G. Then, the G_e has at most Δ more bad triangles compared to G.*

Proof. Let (i, j, k) be a bad triangle in G but not in G_e. Then it must be the case that $e \in t$. WLOG let $e = (i, j)$, and so $k \in N(i) \cup N(j)$. Since $|N(i) \cup N(j)| \leq 2 \max(\deg_i, \deg_j) \leq 2\Delta$, there can be at most Δ new bad triangles in G_e. $\square$

The above proposition is used to establish the τ_{new} bound for Lemma 6. Now, assume a random permutation π for which we run ClusterWild!, and let $\hat{\mathcal{A}} = \cup_{r=1}^{R} \mathcal{A}_r$ denote the union of all active sets of vertices, for each round r of the algorithm. Moreover, let $\hat{G}$, denote the graph that is missing all edges between the vertices in the sets $\mathcal{A}_r$. A simple way to bound the clustering error of ClusterWild!, is splitting it in to two terms: the number of old bad triangles of G adjacent to active vertices (i.e., we need to bound the expectation of the event that an active vertex is adjacent to an "old" triangle), plus the number of *all* new triangles induced by ignoring edges. Observe that this bound can be loose, since not all "new" bad triangles of $\hat{G}$ count towards the clustering error, and some "old" bad triangles can disappear. However, this makes the analysis tractable. Lemma 6 then follows.

Lemma 6. *Let $\hat{G}$ denote the random graph induced by deleting all edges between active vertices per round, for a given run of ClusterWild!, and let τ_{new} denote the number of additional bad triangles that $\hat{G}$ has compared to G. Then, the expected cost of ClusterWild! can be upper bounded as*

$$\mathbb{E}\left\{\sum_{t \in \mathcal{T}_b} \mathbf{1}_{\mathcal{P}_t} + \tau_{new}\right\},$$

where $\mathcal{P}_t$ is the event that triangle t, with end points i, j, k, is bad, and at least one of its end points becomes active, while t is still part of the original unclustered graph.

B Implementation Details

Our implementation is highly optimized in our effort to have practically scalable algorithms. We discuss these details in this section.

B.1 Atomic and non-atomic variables in Java/Scala

In Java/Scala, processors maintain their own local cache of variable values, which could lead to spinlocks in *C4* or greater errors in ClusterWild!. It is necessary to enforce a consistent view across all processors by the use of synchronization or AtomicReferences, but doing so will incur high overheads that render the algorithm not scalable.

To mitigate this overhead, we exploit a monoticity property of our algorithms—the clusterID of any vertex is a non-increasing value. Thus, many of the checks in *C4* and ClusterWild! may be sufficiently performed using only an outdated version of clusterID. Hence, we may maintain both an inconsistent but cheap clusterID array as well as an expensive but consistent atomic clusterID array. Most reads can be done using the cheap inconsistent array, but writes must propagate to the consistent atomic array. Since each clusterID is written a few times but read often, this allows us to minimize the cost of synchronizing values without any substantial changes to the algorithm itself.

We point out that the same concepts may be applied in a distributed setting to minimize communication costs.

B.2 Estimating but not computing Δ

As written, the BSP variants require a computation of the maximum degree Δ at each round. Since this effectively involves a scan of all the edges, it can be an expensive operation to perform at each iteration. We instead use a proxy $\hat{\Delta}$ which is initialized to Δ in the first round, and halved every $\frac{2}{\epsilon} \ln(n \log \Delta/\delta)$ rounds.

With a simple modification to Lemma 1, we can see that w.h.p. any vertex with degree greater than $\hat{\Delta}$ will either be clustered or have its degree halved after $\frac{2}{\epsilon} \ln(n \log \Delta/\delta)$ rounds, so $\hat{\Delta}$ upper-bounds Δ and our algorithms complete in logarithmic number of rounds.

B.3 Lazy deletion of vertices and edges

In practice, we do not remove vertices and edges as they are clustered, but simply skip over them when they are encountered later in the process. We find that this approach decreases the runtimes and overall complexity of the algorithm. (In particular, edges between vertices adjacent to cluster centers may never be touched in the lazy deletion scheme, but must nevertheless be removed in the proactive deletion approach.) Lazy deletions also allow us to avoid expensive mutations of internal data structures.

B.4 Binomial sampling instead of fixed-size batches

Lazy deletion does introduce an extra complication, namely it is now more difficult to sample a fixed-size batch of $\epsilon n_i/\Delta_i$ vertices, where n_i is the number of remaining unclustered vertices. This is because we do not maintain a separate set of n_i unclustered vertices, nor explicitly compute the value of n_i.

We do, however, maintain a set of *unprocessed* vertices, that is, a suffix of π containing n_i unclustered vertices and m_i clustered vertices that have not been passed through by the algorithm. We may therefore resort to an i.i.d. sampling of these vertices, choosing each with probability ϵ/Δ_i. Since processing an unprocessed but clustered vertex has no effect, we effectively simulate an i.i.d. sampling of the n_i unclustered vertices.

Furthermore, we do not have to actually sample each vertex—because π is a uniform random permutation, it suffices to draw $B \sim Bin(n_i + m_i, \epsilon/\Delta_i)$ and extract the next B elements from π for processing, reducing the number of random draws from $n_i + m_i$ Bernoullis to a single Binomial.

All of our theorems hold in expectation when using i.i.d. sampling instead of fixed-size batches.

B.5 Comment on CDK Implementation

A crucial difference between the CDK algorithm and our algorithms lies in the fact that CDK might reject vertices from the active set, which are then placed back into the set of unclustered vertices for potential selection at later rounds. Conversely, our algorithms ensure that the active set is always completely processed, so any vertex that has been selected will no longer be selected in an active set again. We are therefore able to exploit a single random permutation π and use the tricks with lazy deletions and binomial sampling that are not available to CDK, which instead has to perform the complete i.i.d. sampling. We believe that this accounts for the largest difference in runtimes between CDK and our algorithms.
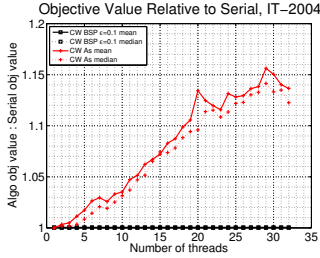
C Full experiment results
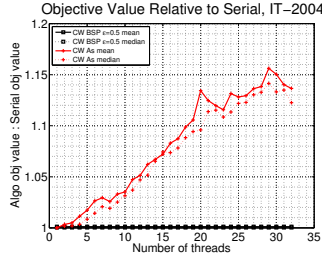
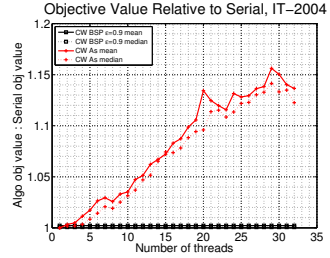(a) UK-2005, $\epsilon = 0.1$ (b) UK-2005, $\epsilon = 0.5$ (c) UK-2005, $\epsilon = 0.9$
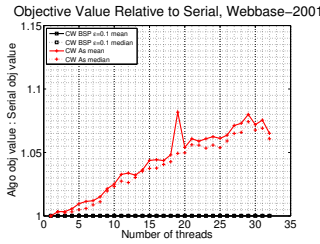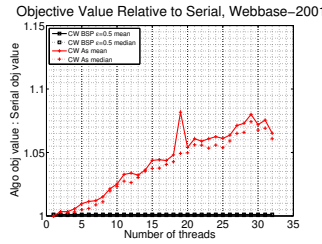
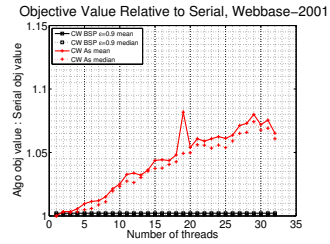(d) IT-2004, $\epsilon = 0.1$ (e) IT-2004, $\epsilon = 0.5$ (f) IT-2004, $\epsilon = 0.9$
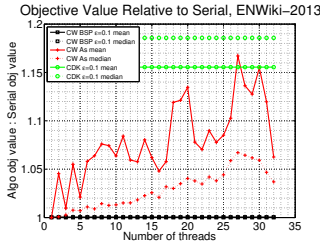
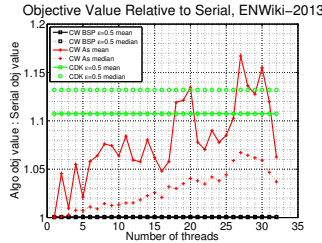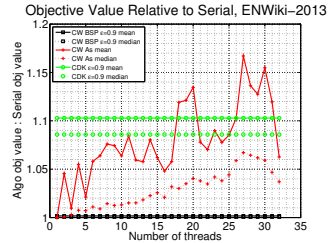(g) Webbase-2001, $\epsilon = 0.1$ (h) Webbase-2001, $\epsilon = 0.5$ (i) Webbase-2001, $\epsilon = 0.9$
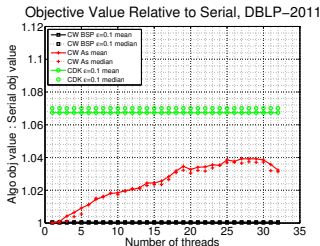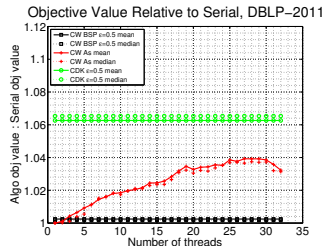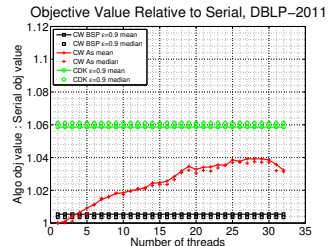
(j) ENWiki-2013, $\epsilon = 0.1$ (k) ENWiki-2013, $\epsilon = 0.5$ (l) ENWiki-2013, $\epsilon = 0.9$
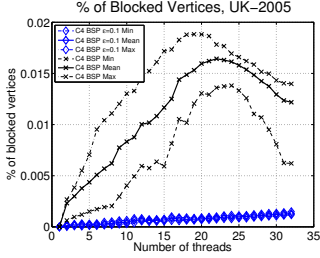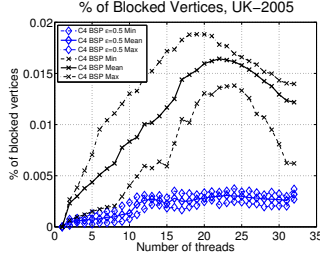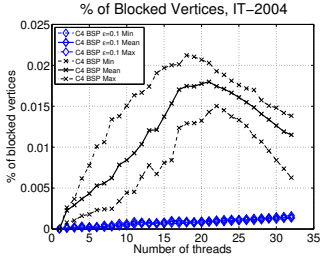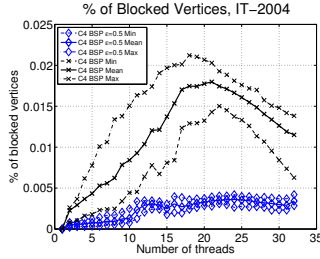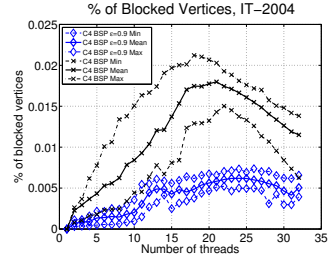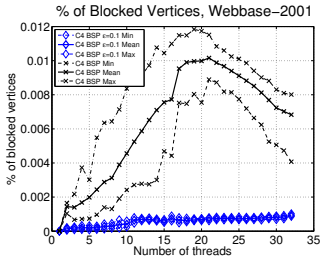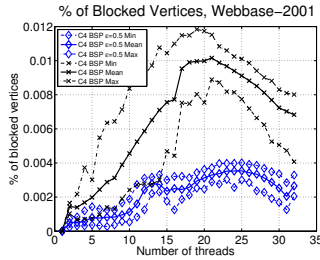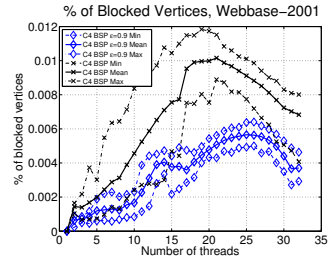
(m) DBLP-2011, $\epsilon = 0.1$ (n) DBLP-2011, $\epsilon = 0.5$ (o) DBLP-2011, $\epsilon = 0.9$
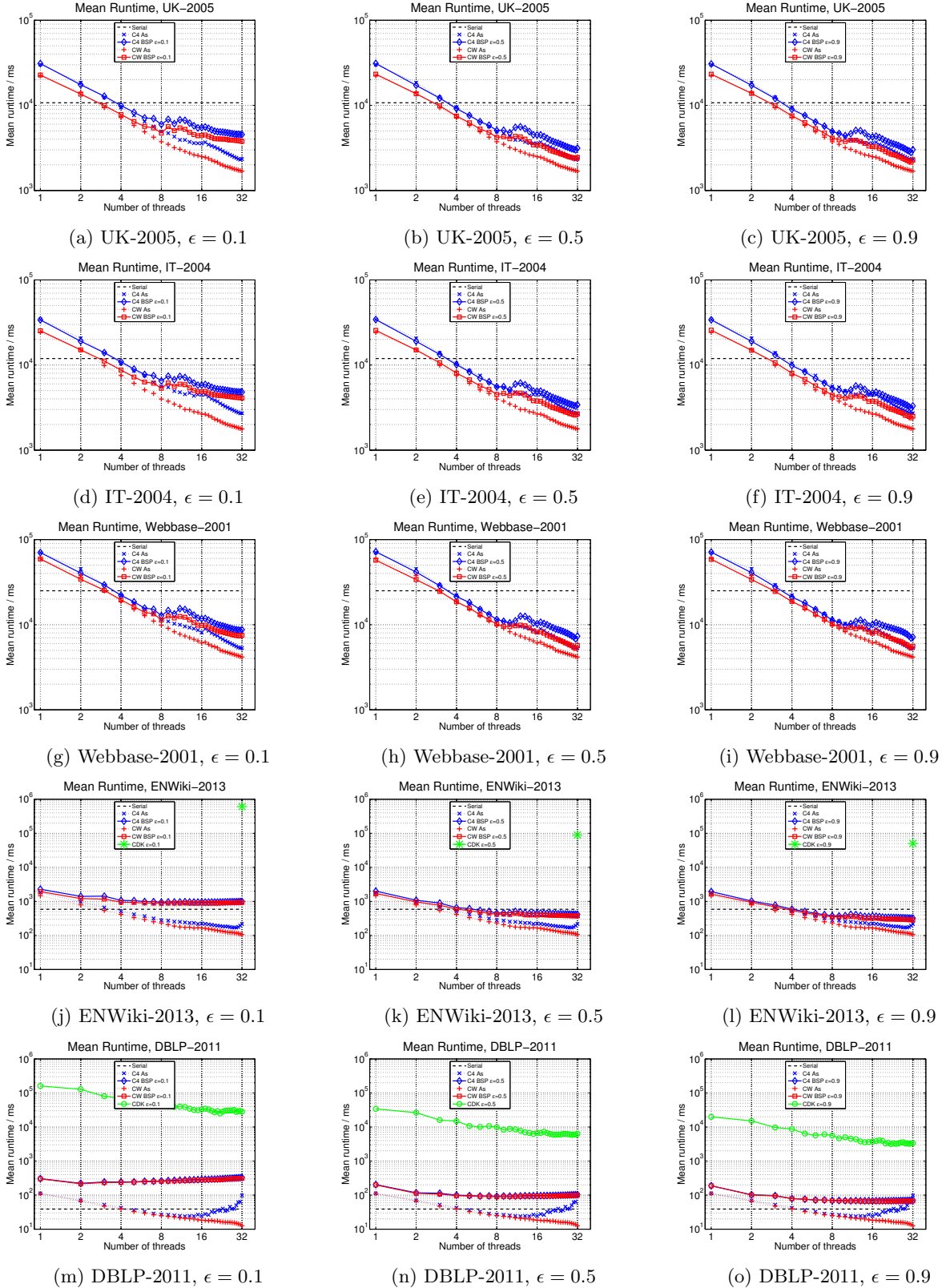
Figure 3: Empirical mean runtimes. For short, 'CW' is CLUSTERWILD! and 'As' refers to the asynchronous variants. On larger graphs, our parallel algorithms on 3-4 threads are faster than serial *KwikCluster*. On the smaller graphs, the BSP variants have expensive synchronization barriers (relative to the small amount of actual work done) and do not necessary run faster than serial *KwikCluster*; the asynchronous variants do outperform serial *KwikCluster* with 4-5 threads. We were only able to run CDK on the smaller graphs, for which CDK was 2-3 orders of magnitude slower than serial. Note also that the BSP variants have improved runtimes for larger ϵ.

19
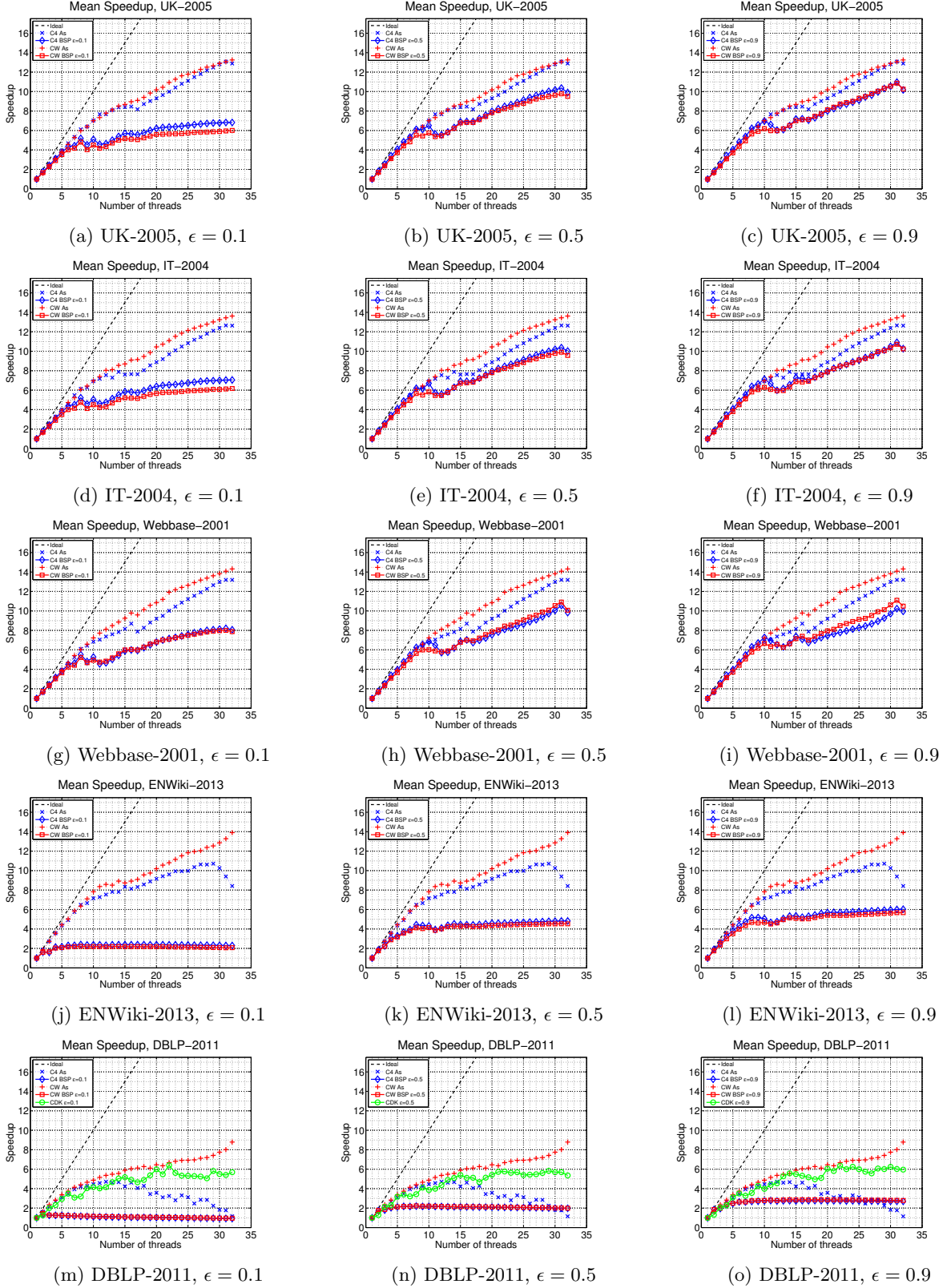
Figure 4: Empirical mean speedups. The best speedups (14x on large graphs) are achieved by asynchronous CLUSTERWILD! which has the least coordination, followed by asynchronous $C4$ (13x on large graphs). The BSP variants achieve up to 10x speedups on large graphs, with better speedups as ϵ increases. On small graphs we obtain poorer speedups as the cost of any contention is magnified as the actual work done is comparatively small. There are a couple of kinks at 10 and 16 threads, which we postulate is due to NUMA and hyperthreading effects—the EC2 r3.8xlarge instances are equipped with 10-core Intel Xeon E5-2670 v2 (Ivy Bridge) processors with 32 vCPUs and hyperthreading.
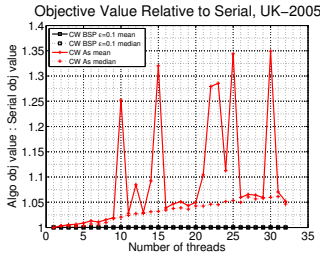
(a) UK-2005, $\epsilon = 0.1$ (b) UK-2005, $\epsilon = 0.5$ (c) UK-2005, $\epsilon = 0.9$
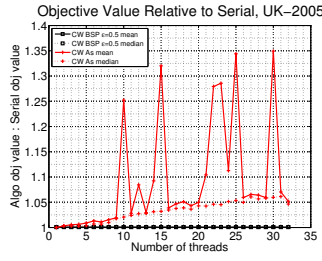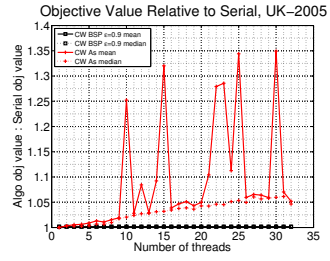
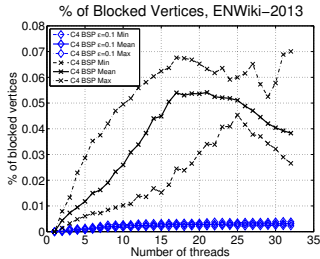(d) IT-2004, $\epsilon = 0.1$ (e) IT-2004, $\epsilon = 0.5$ (f) IT-2004, $\epsilon = 0.9$
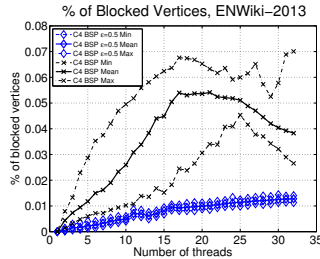
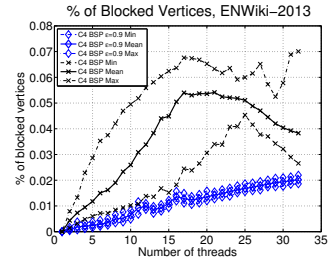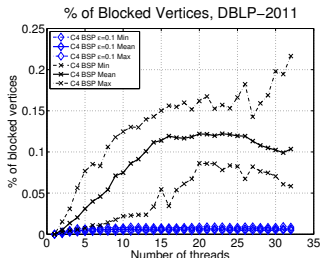(g) Webbase-2001, $\epsilon = 0.1$ (h) Webbase-2001, $\epsilon = 0.5$ (i) Webbase-2001, $\epsilon = 0.9$
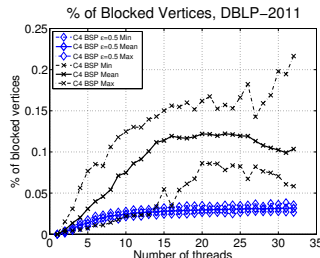
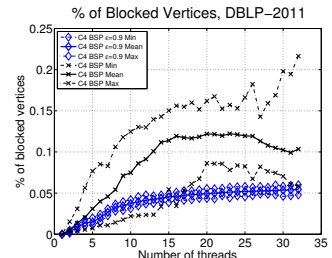(j) ENWiki-2013, $\epsilon = 0.1$ (k) ENWiki-2013, $\epsilon = 0.5$ (l) ENWiki-2013, $\epsilon = 0.9$

(m) DBLP-2011, $\epsilon = 0.1$ (n) DBLP-2011, $\epsilon = 0.5$ (o) DBLP-2011, $\epsilon = 0.9$

Figure 5: Empirical objective values relative to mean objective value obtained by serial algorithm.
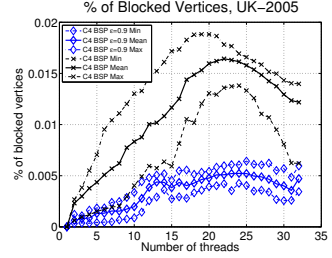
Figure 6: Empirical percentage of blocked vertices. Generally the number of blocked vertices increases with the number of threads and larger ϵ values. *C4* BSP has fewer blocked vertices than asynchronous *C4*, but at the cost of more synchronization barriers. We point out that across all 100 runs of every graphs, the maximum percentage of blocked vertices is less than 0.25%; for large sparse graphs, the maximum percentage is less than 0.025%, i.e., 1 in 4000.