Title
Parallel Machine Learning Using Concurrency Control

Author
Pan, Xinghao

Publication Date
2017

Peer reviewed|Thesis/dissertation

Parallel Machine Learning Using Concurrency Control

by

Xinghao Pan

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

and the Designated Emphasis

in

Communications, Computation and Statistics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael I. Jordan, Chair
Professor Joseph Hellerstein
Associate Professor Benjamin Recht
Professor Thomas Griffiths

Summer 2017

Parallel Machine Learning Using Concurrency Control

Abstract

Parallel Machine Learning Using Concurrency Control

by

Xinghao Pan

Doctor of Philosophy in Computer Science
and the Designated Emphasis in
Communications, Computation and Statistics

University of California, Berkeley

Professor Michael I. Jordan, Chair

Many machine learning algorithms iteratively process datapoints and transform global model parameters. It has become increasingly impractical to serially execute such iterative algorithms as processor speeds fail to catch up to the growth in dataset sizes.

To address these problems, the machine learning community has turned to two parallelization strategies: bulk synchronous parallel (BSP), and coordination-free. BSP algorithms partition computational work among workers, with occasional synchronization at global barriers, but has only been applied to 'embarrassingly parallel' problems where work is trivially factorizable. Coordination-free algorithms simply allow concurrent processors to execute in parallel, interleaving transformations and possibly introducing inconsistencies. Theoretical analysis is then required to prove that the coordination-free algorithm produces a reasonable approximation to the desired outcome, under assumptions on the problem and system.

In this dissertation, we propose and explore a third approach by applying *concurrency control* to manage parallel transformations in machine learning algorithms. We identify points of possible interference between parallel iterations by examining the semantics of the serial algorithm. Coordination is then introduced to either avoid or resolve such conflicts, whereas non-conflicting transformations are allowed to execute concurrently. Our parallel algorithms are thus engineered to produce the same exact output as the serial machine learning algorithm, preserving the serial algorithm's theoretical guarantees of correctness while maximizing concurrency.

We demonstrate the feasibility of our approach to parallelizing a variety of machine learning algorithms, including nonparametric unsupervised learning, graph clustering, discrete optimization, and sparse convex optimization. We theoretically prove and empirically verify that our parallel algorithms produce equivalent output to their serial counterparts. We also theoretically analyze the expected concurrency of our parallel algorithms, and empirically demonstrate their scalability.

Contents

Acknowledgments

I would like to thank my advisor, Michael I. Jordan, for his support and advice through graduate school. Mike was an inspiration for conducting research in machine learning and always staying ahead of the field. He taught me to focus on the big ideas without losing track of the details, and instilled in me a mathematical rigor that I did not realize I was capable of.

I would also like to thank the members of my thesis committee: Joe Hellerstein, for constantly challenging me to sharpen my arguments and turning this thesis into its current final form; Ben Recht, for always cutting to the core of the research issue and asking the hard questions; and Tom Griffiths, for providing valuable feedback on the Bayesian nonparametrics aspects of this work.

There have been many researchers I have had the privilege to work with over the past five years. Special credit goes to my closest collaborators, Joseph Gonzalez and Dimitris Papailiopoulos. Joey was a major influence in the early days when I was still crystallizing the central theme of database-inspired parallel machine learning. The numerous discussions we had on systems and learning problems in large scale machine learning deepened my understanding of each field, and enhanced my appreciation of the intricacies of making them work together. Dimitris came on board to this project later, but was one of my greatest supporters in the effort to marry concurrency control with machine learning. His enthusiasm and determination to push through seemingly impossible difficulties — both theoretical and practical — was the driving force for the second half of this work.

Stefanie Jegelka, Tamara Broderick, Maximilian Lam, Stephen Tu, Ce Zhang, Samet Oymak, Chris Re, and Kannan Ramchandran co-authored papers which contributed to this work; this thesis would not have been possible without their expertise and talents. I have also had the honor of collaborating with Horia Mania, Evan Sparks, Ameet Talwalkar, Virginia Smith, Andre Wibisono, and Shivaram Venkataraman on research outside the scope of this thesis. Many others from SAIL and AMP Lab have at various times provided feedback and been sounding boards for my random ideas. This vibrant community of machine learning folks — Elaine Angelino, Chi Jin, Robert Nishihara, Philipp Moritz, Xin Wang — and systems and databases people — Dan Crankshaw, Neeraja Yadwadkar, Sanjay Krishnan — has made me feel at home in Berkeley.

My gratitude also goes out to Jianmin Chen and Rajat Monga who hosted me at Google Brain during my internship. Without too much evidence, they chose to trust my ability and afforded me the opportunity to work on a real-world large scale learning problem. I am thankful for their patience while I fumbled around, and for their understanding at times when personal matters took precedence over work.

This work was sponsored in large part by a DSO National Laboratories Postgraduate Scholarship. I am greatly appreciative of the support that the organization and my superiors, Khee Yin How, Gee Wah Ng, and Loo Nin Teow have provided throughout my graduate studies.

Finally, and most importantly, I am grateful to my wife, Rachel (Jue) Rui, for her endless support, leaving behind everything in Singapore to live with me in a foreign land. It has

been twelve years filled with wonderful memories and unexpected adversities. Thank you for being by my side through all the highs and lows, and for carrying on this journey with me until the end.

Chapter 1

Introduction

Many machine learning algorithms iteratively process datapoints and transform some global state (e.g. model parameters or variable assignments). Such algorithms have typically been developed and studied in the serial setting, where iterations are sequentially executed after transformations of earlier iterations have completed. However, the impending death of Moore's law combined with the advent of the big data era has rendered it impractical to execute millions to trillions of transformations serially. The desire to apply machine learning to increasingly larger datasets has pushed the machine learning community to design parallel algorithms, for both multicore and distributed settings, with the twin objectives of:

- **Correctness**: The parallel algorithm is theoretically guaranteed to produce the desired solution to the machine learning problem, or at least a quantifiable approximation to the desired solution. One method for achieving correctness is to ensure equivalence of output to the serial algorithm — theoretical guarantees of the serial algorithm then immediately translate to the parallel algorithm.
- **Concurrency**: Increasing the computational resources decreases the time required to obtain the solution; ideally, with linear speedup, i.e., doubling the computational resources should at least halve the running time.

In general, there have been two classes of approaches for the parallelization of machine learning algorithms.

Simple Bulk Synchronous Parallel (BSP). Simple BSP algorithms partition the work of *each iteration* across multiple workers, each of which computes an intermediate result for the iteration. These intermediate results are aggregated before being applied as a transformation to the global state. A global synchronization barrier prevents workers from proceeding with the computation of the following iteration before the transformation of the current iteration completes. While the simple BSP approach ensures correctness, it is limited to algorithms for which an iteration's work factors trivially[1].

[1] Also known as 'embarrassingly parallel' problems.

Coordination-free. A second class of *coordination-free* algorithms simply allows workers to each execute multiple iterations, in parallel, as they would have done in the serial algorithm. Concurrent iterations are interleaved, and workers may partially or completely overwrite each other's progress, possibly leading to inconsistent states. Such approaches, exemplified by HOGWILD! [116], appear to emphasize the maximization of concurrency with an apparent disregard for correctness. Surprisingly, recent breakthroughs have demonstrated that many such coordination-free machine learning algorithms retain theoretical guarantees of approximate correctness. Unfortunately, these analyses are tailored to individual algorithms, work under various assumptions on the computation system and / or problem, and provide approximations that worsen with increasing parallelism.

Concurrency control for parallel machine learning. In this dissertation, we explore an alternative approach to simple BSP and coordination-free methods for parallelizing machine learning algorithms. Specifically, we wish to achieve the following goals:

- *Strong guarantees of correctness.* We desire theoretical guarantees that do not worsen with increasing parallelism, and prioritize achieving correctness over maximizing concurrency.
- *Scalable.* Ideally, we should be able to prove speedups of our parallel algorithms under assumptions on the system and problem, or at minimum, demonstrate empirical speedups for the parallel algorithms.
- *Verifiable and reproducible results.* It should be possible to verify that an implementation of the parallel algorithm is producing the correct output. Ideally, multiple runs of the parallel algorithm should produce the same output. These properties aid testing and debugging, and help to ensure correct implementations.

Our proposal for achieving these goals is to employ *concurrency control* mechanisms for coordinating parallel workers, ensuring that concurrent iterations and transformations do not interfere with one another. Concurrency control has been studied by database systems researchers as a means for parallel execution of transactions. A database transaction is a set of read-compute-write operations that need to be executed atomically and free of interference from other transactions. This in turn guarantees **serializability**: the parallel execution of transactions produces an output that is equivalent to some serial execution of the transactions. If the transactions' ordering is recorded, the database output can be verified by comparing it against a serial execution obeying the same ordering.

In our context, each iteration of the machine learning algorithm will be cast as a database transaction. We will identify potential conflicts between concurrent transactions through an understanding of the algorithm's semantics. By either avoiding or resolving such conflicts through concurrency control, we guarantee that our parallel algorithms are serializable, and therefore retains the same theoretical guarantees of correctness of the serial algorithm.

In most cases, we are also able to show that the machine learning algorithms parallelized through concurrency control are **deterministic**, i.e., the parallel execution corresponds to a pre-determined order of serial execution, which grants the added advantage of reproducible and verifiable results, and ease of debugging.

A key to achieving high concurrency is the insight that the iterative transformations of many machine learning algorithms are only weakly dependent on one another. Thus, most transactions can execute concurrently without conflict. We quantify the concurrency and scalability of our parallel algorithms by analyzing the occurrence of conflict, under assumptions on the computation system and / or problem.

1.1 Contributions

Concretely, the contributions of this dissertation are as follows:

1. We present a general approach of applying concurrency control to the parallelization of machine learning algorithms.
2. We demonstrate the feasibility of our approach by application to algorithms for different classes of machine learning problems:
   a) Nonparametric unsupervised learning (Chapter 4)
   b) Graph clustering (Chapter 5)
   c) Discrete optimization (Chapter 6)
   d) Sparse convex optimization (Chapter 7)
3. We prove that our parallel algorithms are serializable (or deterministic), and therefore preserve the theoretical guarantees of the serial counterparts. Our parallel algorithms are verifiable (by comparing the results against a serial run with the corresponding ordering of transactions) and easy to study (by reduction to the serial algorithm).
4. We theoretically analyze the expected concurrency of our parallel algorithms, and empirically validated the algorithms' speedups.
5. For the correlation clustering (Chapter 5) and double greedy (Chapter 6) algorithms, we also propose coordination-free parallelizations. We analyze the theoretical approximations of the coordination-free algorithms, and examine the trade-offs of both the concurrency controlled and coordination-free algorithms.

1.2 Organization and Key Results

The remainder of this dissertation is organized as follows.

We present a model of machine learning algorithms as iterative transformations in Chapter 2, and discuss both traditional database concurrency control mechanisms and existing parallel machine learning approaches in this context.

Chapter 3 presents the general approach of applying concurrency control to the parallelization of machine learning algorithms. We also compare our approach with other parallel machine learning approaches and traditional concurrency control techniques in transactional databases.

We demonstrate concrete applications of the concurrency control approach to various machine learning algorithms in Chapters 4, 5, 6, and 7. In Chapter 4, we study three nonparametric unsupervised learning algorithms. DP-means and BP-means are respectively small-variance asymptotic approximations to Gibbs samplers for Dirichlet processes and Beta processes; the online facility location (OFL) algorithm is a clustering algorithm closely related to DP-means. All three algorithms iteratively process datapoints, and possibly propose new clusters / features if the datapoint is not well-represented by existing clusters / features. Our parallelizations utilize an optimistic concurrency control protocol to resolve conflicting proposals for new clusters / features. We empirically evaluated and demonstrated the scalability of our parallel algorithms in a distributed, cluster computing environment.

We continue in Chapter 5 with the theme of nonparametric clustering, specifically, correlation clustering, which aims to group similar items in a similarity graph together. One algorithm that achieves a 3-approximation for correlation clustering is KwikCluster, which iteratively picks an unclustered vertex, and places the vertex and its unclustered neighbors in a new cluster. Our first parallelization of KwikCluster uses a mix of locking and compensation logic to achieve a deterministic parallel algorithm C4. We also examine a coordination-free parallelization, ClusterWild!, and provide theoretical guarantees of speedups and approximations for both C4 and ClusterWild!. Empirically, our solutions outperform the previous state-of-the-art parallel correlation clustering algorithm in both speed and clustering quality.

Chapter 6 focuses on the important class of non-monotone submodular maximization problems. The randomized double greedy algorithm has been shown to obtain an optimal 1/2 approximation in expectation; however, this algorithm is particularly serial – each iteration takes the entire global state as input and always makes some transformation to the global state. Our insight for parallelization is that, under the randomization, there is a small probability that an iteration's computation is affected by other concurrent iterations. Thus, most iterations can proceed concurrently without interference, and in the minute chance that a conflict could occur, we simply force the later iteration to wait. The resultant deterministic algorithm, which we term CC-2g, preserves the optimal 1/2 approximation. We also propose a coordination-free algorithm, CF-2g, which incurs an additive error due to conflicts. We empirically evaluated CC-2g and CF-2g on large synthetic and real datasets.

In Chapter 7 we parallelize a large class of sparse stochastic optimization algorithms, which include SGD (with or without weight decay), SVRG, and SAGA. We exploit the fact that concurrent sparse updates are unlikely to share parameters that they read or write. Our parallelization, Cyclades, combines BSP with concurrency control — iterations are processed

in batches, and conflicting iterations within each batch are serialized by assigning them to the same processor. This technique obviates the use of explicit locks and provides better cache coherency. The BSP approach also allows us to theoretically analyze Cyclades' running time and prove it achieves near linear speedup. We demonstrate that for sufficiently sparse problems, Cyclades can outperform coordination-free implementations, even for notoriously robust algorithms such as SGD.

Finally, we conclude and discuss some potential future directions of the concurrency control approach to parallelization of machine learning algorithms in Chapter 8.

Chapter 2

Background

We present a general model of machine learning algorithms in this chapter, and discuss both traditional database concurrency control mechanisms and existing parallel machine learning approaches in the context of our model.

2.1 Iterative Transformations

A large number of machine learning algorithms can be viewed as iterative transformations to a global state of model parameters. Given a dataset $\mathcal{X} = \{x_1, \ldots, x_n\}$ and model parameters θ, the t-th iteration of the algorithm computes the update

$$\theta^{(t+1)} = T_t(\theta^{(t)}, \Lambda_t(\theta^{(t)}, X^{(t)})) \tag{2.1}$$

where $\theta^{(t)}$ is the model after t iterations, and $X^{(t)} \subseteq \mathcal{X}$ is the subset of data that the t-th iteration operates on. Here, the change function $\Lambda_t(\theta, X)$ computes an intermediate result based on θ and X, which is then applied to the model by the transformation function $T_t(\cdot, \cdot)$. It will often be the case that the ordering of the updates is inconsequential, in the sense that executing the updates in a different order still produces a valid, though possibly different, outcome. For some other algorithms, we will additionally require that the update ordering be a uniformly random permutation.

We will sometimes make the dependence on individual d coordinates of θ explicit by writing

$$\theta_k^{(t+1)} = T_{t,k}(\theta_k^{(t)}, \Lambda_t(\{\theta_1^{(t)}, \ldots, \theta_d^{(t)}\}, X^{(t)})). \tag{2.2}$$

We assume that the transformation factors as $[T_t(\theta, \lambda)]_k = T_{t,k}(\theta_k, \lambda)$, where the notation $[\cdot]_k$ denotes the k-th coordinate. In some cases, Λ_t will only depend on a subset of coordinates, and T_t will only transform a subset of coordinates:

$$\theta_k^{(t+1)} = \begin{cases} T_{t,k}(\theta_k^{(t)}, \Lambda_t(\{\theta_r^{(t)} : r \in R(X^{(t)})\}, X^{(t)})) & \text{if } k \in W(X^{(t)}) \\ \theta_k^{(t)} & \text{otherwise} \end{cases} \tag{2.3}$$

where R and W are the read and write sets[1] respectively.

This formulation was alluded to in [111, 110], and also recognized by others in [135]. We next present a number of example machine learning algorithms that fit into this framework.

Example 2.1 (Stochastic Gradient Descent (SGD))**.** *Given a composite loss function* $F(\theta) = \frac{1}{n}\sum_{i=1}^{n} f_i(\theta; x_i)$*, SGD minimizes* F *with respect to* θ *by computing the iteration*

$$\theta^{(t+1)} = \theta^{(t)} - \gamma_t \nabla_\theta f_{i^{(t)}}(\theta; x_{i^{(t)}})\big|_{\theta=\theta^{(t)}}, \tag{2.4}$$

where $i^{(t)} \in \{1, \ldots, n\}$ *is a random index, drawn uniformly at random with or without replacement. SGD can be represented as an iterative transformation with* $\Lambda((\eta, t), \mathcal{X}) = -\gamma_t \nabla_\theta f_{i^{(t)}}(\theta; x_{i^{(t)}})\big|_{\theta=\eta}$ *and* $T((\eta, t), \lambda) = (\eta + \lambda, t + 1)$*. Note that the updates can be reordered with a uniformly random permutation while still producing a valid outcome, without impacting any theoretical properties of the algorithm.*

Example 2.2 (Minibatch SGD)**.** *Minibatch SGD seeks to optimize the same composite loss as SGD, but computes the gradient over a random minibatch* $B^{(t)} = \{B_1^{(t)}, \ldots, B_m^{(t)}\} \subseteq \{1, \ldots, n\}$*. We consider two ways to cast minibatch SGD in the iterative transformations framework. Firstly, we can consider the application of the minibatch gradient as a single iteration:*

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\gamma_t}{|B^{(t)}|} \sum_{i \in B^{(t)}} \nabla_\theta f_i(\theta, x_i)\big|_{\theta=\theta^{(t)}} \tag{2.5}$$

with $\Lambda((\eta, t), \mathcal{X}) = -\frac{\gamma_t}{|B^{(t)}|}\sum_{i \in B^{(t)}} \nabla_\theta f_i(\theta, x_i)\big|_{\theta=\eta}$ *and* $T((\eta, t), \lambda) = (\eta + \lambda, t + 1)$*. Alternatively, a minibatch can be viewed as* m *individual iterations*

$$\theta^{(t,s+1)} = \theta^{(t,s)} - \frac{\gamma_t}{|B^{(t)}|} \nabla_\theta f_{B_s^{(t)}}(\theta, x_{B_s^{(t)}})\big|_{\theta=\theta^{(t,0)}} \tag{2.6}$$

$$\theta^{(t+1,0)} = \theta^{(t,|B^{(t)}|)} \tag{2.7}$$

with $\Lambda((\eta, t, s), \mathcal{X}) = \frac{\gamma_t}{|B^{(t)}|} \nabla_\theta f_{B_s^{(t)}}(\theta, x_{B_s^{(t)}})\big|_{\theta=\eta}$*, and* $T((\eta, t, s), \lambda) = (\eta + \lambda, t, s + 1)$ *if* $s < |B^{(t)}|$ *and* $(\eta + \lambda, t + 1, 0)$ *otherwise.*

Example 2.3 (Conditional Gibbs Sampler for Beta-Bernoulli)**.** *Consider the generative probabilistic model*

$$p \sim Beta(1, 1) \tag{2.8}$$

$$x_i \sim Bernoulli(p), \qquad i = 1, \ldots, n \tag{2.9}$$

A conditional Gibbs sampler for this model iteratively performs the sampling

$$p | x_1, \ldots, x_n \sim Beta\left(1 + \sum_{i=1}^{n} x_i, 1 + n - \sum_{i=1}^{n} x_i\right) \tag{2.10}$$

$$x_i | p, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \sim Bernoulli(p), \qquad i = 1, \ldots, n \tag{2.11}$$

[1] We assume that the read and write sets depend only on the data subset $X^{(t)}$, and not on $\theta^{(t)}$ or Λ_t, which will suffice for most of our example applications.

Note that for $i \neq i'$, x_i is conditionally independent of $x_{i'}$ given p. The above iterative transformations can be described by

$$[\Lambda_p(q, y_1, \ldots, y_n)]_p \sim Beta\left(1 + \sum_{i=1}^{n} y_i, 1 + n - \sum_{i=1}^{n} y_i\right) \tag{2.12}$$

$$[\Lambda_{x_i}(q, y_1, \ldots, y_n)]_{x_i} \sim Bernoulli(q) \tag{2.13}$$

$$T_p(\cdot, \lambda) = \lambda_p \qquad W_p = \{p\} \tag{2.14}$$

$$T_{x_i}(\cdot, \lambda) = \lambda_{x_i} \qquad W_{x_i} = \{x_i\} \tag{2.15}$$

Example 2.4 (Marginal Gibbs Sampler for Beta-Bernoulli)**.** *We consider the same generative probabilistic model as before, but with a Gibbs sampler where p is marginalized. The sampling proceeds as*

$$x_i | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \sim Bernoulli\left(\frac{1 + \sum_{i' \neq i} x_{i'}}{1 + n}\right) \tag{2.16}$$

which has the iterative transformation

$$[\Lambda_{x_i}(y_1, \ldots, y_n)]_{x_i} \sim Bernoulli\left(\frac{1 + \sum_{i' \neq i} y_{i'}}{1 + n}\right) \tag{2.17}$$

$$T_{x_i}(\cdot, \lambda) = \lambda_{x_i} \qquad W_{x_i} = \{x_i\} \tag{2.18}$$

In a parallel execution of iterative transformations, concurrent reads and writes may be interleaved and re-ordered. Without additional coordination, the change function Λ and transformation function T of the t-th iteration may access model parameters not necessarily generated by the $(t-1)$-th iteration. That is, the output of the t-th iteration is

$$\theta_k^{(t+1)} = \begin{cases} T_k(\theta_k^{(t-\sigma_{t,k})}, \Lambda(\{\theta_r^{(t-\tau_{t,r})} : r \in R(X^{(t)})\}, X^{(t)})) & \text{if } k \in W(X^{(t)}) \\ \theta_k^{(t)} & \text{otherwise} \end{cases} \tag{2.19}$$

for some arbitrary delays $\sigma_{t,k}$ and $\tau_{t,r}$. These 'delays' need not be positive, since a 'later' iteration may be interleaved with the t-th iteration, and update the coordinate θ_k before the t-th iteration reads θ_k. Note also that $\theta^{(t)}$ may never exist as a consistent state — at any point of time, there may exist coordinates k and k' such that $\theta_k^{(t)}$ and $\theta_{k'}^{(t')}$ are simultaneously stored in memory, with $t \neq t'$. The challenge of any parallelization of the iterative transformation algorithm is to show, through a mix of coordination and analysis on the robustness of the algorithm, that (2.19) produces the (approximately) correct solution despite the interactions between concurrent iterations.

2.2 Concurrency Control

Concurrency control is a well-studied area of database systems research for parallel execution of database transactions, which are sets of read-compute-write operations. The goal of concurrency control is to maximize throughput (number of transactions processed per unit time), minimize latency (total time taken to process each transaction), while ensuring that the resultant database corresponds to the state of some serial execution of the transactions.

In the context of our iterative transformation framework, we observe that since iterations may be reordered arbitrarily, we essentially have a *set* of updates that needs to be executed. We will therefore treat each iteration as a database transaction $\mathcal{T}_t = (T_t, \Lambda_t, X^{(t)})$ to be executed by the parallel algorithm; the goal is thus to ensure the parallel algorithm corresponds to some execution of the serial algorithm, under any ordering of the iterations. Formally, we can define serializability as below.

Definition 2.1 (Serializability)**.** *Let $\mathcal{A}$ be a parallel algorithm taking as input a set of transactions $\{\mathcal{T}_1, \ldots, \mathcal{T}_N\}$ where $\mathcal{T}_t = (T_t, \Lambda_t, X^{(t)})$, and let $\theta^{(t+1)}$ be the output of its t-th transaction (iteration) be defined as* (2.19)*. The algorithm $\mathcal{A}$ is* serializable *if for every parallel execution of $\mathcal{A}$, there is some permutation π such that*

$$\theta_k^{(\pi(t)+1)} = \begin{cases} T_{t,k}(\theta_k^{(\pi(t))}, \Lambda_t(\{\theta_r^{(\pi(t))} : r \in R(X^{(t)})\}, X^{(t)})) & \text{if } k \in W(X^{(t)}) \\ \theta_k^{(\pi(t))} & \text{otherwise} \end{cases} \tag{2.20}$$

or more concisely

$$\theta^{(\pi(t)+1)} = T_t(\theta^{(\pi(t))}, \Lambda_t(\theta^{(\pi(t))}, X^{(t)})) \tag{2.21}$$

and $\mathcal{A}$ generates output $\theta^{(\pi(N))}$.

Said differently, a serializable parallel algorithm may reorder iterations, but ensures that the output is equivalent to a serial execution of the reordered iterations, specifically, where $\mathcal{T}_t$ is executed as the $\pi(t)$-th transaction. The definition of serializability makes no reference to a serial algorithm, but merely to *some serial execution* of transactions[2] Note also that serializability is a statement on the parallel algorithm's output and not of its execution; in particular, the state[3] of a serializable parallel algorithm need not be consistent until it produces the final output.

Many mechanisms have been proposed for concurrency control. We review a few common techniques in this chapter, following the classification in Chapter 18 of [58]. For a more complete review, the reader is referred to [58, 16, 17].

[2] Nevertheless, this will suffice for our needs: since the serial algorithm allows for arbitrary reordering of iterations, every serial execution of transactions is a valid outcome of the serial algorithm.

[3] In the machine learning setting, the dataset $\mathcal{X}$ is given and considered to be immutable, i.e., read-only, and thus requires no coordination of access. The 'state' to which we apply concurrency control mechanism refers only to the model parameters θ.

Locking

Locking is a common concurrency control technique for enforcing mutual exclusion. Intuitively, when a transaction acquires a 'lock' on a coordinate θ_k, it receives ownership of θ_k so no other transaction may make changes to θ_k.

There are two types of locks that a transaction may acquire. A read lock gives the transaction the right to read a coordinate θ_k without θ_k changing, and must be successfully acquired and held during the transaction reading θ_k. A write lock gives the transaction the sole right to make changes to the coordinate θ_k, and must be successfully acquired and held during the transaction writing to θ_k. Read locks conflict with write locks — an existing read lock on θ_k precludes another transaction from acquiring a write lock on θ_k, and vice versa. Write locks also conflict with write locks — no two transactions may hold write locks on the same coordinate θ_k at the same time. However, read locks do not conflict with read locks — multiple transactions may hold read locks on the same coordinate simultaneously. If a transaction requests for a lock but is blocked by an existing lock, it must wait until the conflicting lock has been released.

A protocol for using locks to achieve serializability is *two-phase locking*: in the first phase, a transaction may acquire but not release locks; in the second phase, the transaction may release but not acquire locks. The transaction is logically timestamped at the point after the first phase but before the start of the second phase. Transaction timestamps can be used to provide the serialization ordering π. A simple inductive proof shows that two-phase locking is indeed serializable. Intuitively, the transaction must read all earlier conflicting writes (since earlier writes must complete before the releasing of write locks and the transaction's acquiring a read lock); the transaction's writes must be read by later conflicting reads (for the same reason). Furthermore, if two transactions have write-write conflicts on multiple coordinates, the earlier transaction must have acquired all write locks on conflicting coordinates before the later transaction acquires any write locks on these coordinates (for otherwise the protocol is not two-phase), and so writes of the later transaction to the conflicting coordinates will always overwrite writes of the earlier transaction.

A potential problem of locking arises when a set of transactions have partially acquired their requisite locks, but are blocked by acquiring the remaining locks by other transactions in the set. This situation is known as *deadlock*, and many schemes have been proposed to either avoid or detect and break deadlocks. For our purposes, we will use the simple scheme of acquiring locks in a pre-determined, globally agreed-upon lock order, which is a feasible method since we assume read and write sets are declared upfront.

Locking is a pessimistic concurrency control mechanism, in that it assumes that conflicts are likely and hence actively seeks to prevent them from happening. The other two types of concurrency control mechanisms are optimistic in comparison — they assume that nothing will go wrong, but then take actions to remedy things when conflicts do happen.

Timestamp Ordering

In two-phase locking, the timestamp of the transaction is implicitly defined. The key idea of timestamp ordering is to explicitly assign a timestamp to each transaction, and then to ensure that transactions execute in the order of their timestamps. Intuitively, a transaction's read is allowed if it has not been superseded by a later transaction's write, and a transaction's write is allowed if it does not interfere with a later transactions' read. (For clarity of exposition, we omit technicalities with transaction commits.)

Formally, each transaction $\mathcal{T}$ is assigned a timestamp $TS(\mathcal{T})$, and every coordinate θ_k is assigned a write-timestamp $WT(\theta_k)$ and a read-timestamp $RT(\theta_k)$. The write-timestamp $WT(\theta_k)$ is the timestamp of the transaction that wrote the stored value of θ_k, and the read-timestamp $RT(\theta_k)$ is the largest timestamp of transactions that have read θ_k. The rules for timestamp ordering are:

1. A read of θ_k by $\mathcal{T}$ is allowed if $TS(\mathcal{T}) \geq WT(\theta_k)$. If the read is successful, set $RT(\theta_k) \leftarrow \max\{RT(\theta_k), TS(\mathcal{T})\}$.

2. A write of θ_k by $\mathcal{T}$ is allowed if $TS(\mathcal{T}) \geq RT(\theta_k)$. If also $TS(\mathcal{T}) \geq WT(\theta_k)$, we write the new value of θ_k and set $WT(\theta_k) \leftarrow TS(\mathcal{T})$. Otherwise, if $TS(\mathcal{T}) < WT(\theta_k)$, then $\mathcal{T}$'s write has already been superseded by the write from a later transaction $\mathcal{T}'$ with $TS(\mathcal{T}') = WT(\theta_k)$, so we simply ignore $\mathcal{T}$'s write to θ_k — this is known as the Thomas Write Rule.

In case any read or write is unsuccessful, the transaction $\mathcal{T}$ must rollback all its writes to the database, and retry from scratch with a new timestamp.

Validation

The *validation* concurrency control method was first published in [86] under the name of 'optimistic concurrency control', and was the first to use the term 'optimistic' to describe non-locking techniques. At a high level, the validation method optimistically allows all reads to proceed without blocking, but any change to the database via a write must be validated to ensure that it does not conflict with other transactions.

A transaction $\mathcal{T}$ in a validation-based database goes through three phases, and is logically timestamped at the second phase. In the first read phase, it is allowed to read any coordinate θ_k from the database. In the second validation phase, the database checks that $\mathcal{T}$'s reads reflect writes of earlier transactions, and that its writes will not be overwritten by earlier transactions. If the validation completes successfully, $\mathcal{T}$ instantiates its writes into the database; otherwise, $\mathcal{T}$ must retry from scratch again.

We now present the details of the validation phase for avoiding read-write and write-write conflicts:

1. *Read-write conflicts:* If $\mathcal{T}$ reads θ_k and an earlier transaction $\mathcal{T}'$ writes to θ_k, the validation must ensure that $\mathcal{T}$ reads the write of θ_k by $\mathcal{T}'$. This is conservatively

checked by comparing the time $\mathcal{T}'$ finished its writes to the start time of $\mathcal{T}$, and confirming that the former is larger than the latter.

2. *Write-write conflicts:* If $\mathcal{T}$ and an earlier transaction $\mathcal{T}'$ both write θ_k, then $\mathcal{T}$'s write should be reflected in the database. This is conservatively checked by comparing the time $\mathcal{T}'$ finished its writes to the time $\mathcal{T}$ starts its writes (i.e. the validation time), and confirming that the former is larger than the latter.

Applying Concurrency Control to Iterative Transformations

It is straightforward to apply traditional concurrency control mechanisms, including the three reviewed above, to parallelizing iterative transformations. In particular, when the read set $R(X^{(t)})$ and write set $W(X^{(t)})$ are defined, as in (2.19), we can cast each iteration as a transaction and provide the read and write sets to the transactional database, which in turn applies some form of concurrency control based on resolving read-write and write-write conflicts to produce a serializable output.

2.3 Parallel Machine Learning Approaches

There has been a proliferation of parallel machine learning algorithms in recent years. We briefly review some of these approaches in the context of the iterative transformations framework, and highlight a couple of systems with particular relevance to our approach.

Simple Bulk Synchronous Parallel (BSP)

The BSP model [132] of computation is a method for ensuring lock-step progress of parallel workers. Specifically, there are three phases in a global BSP superstep: (1) concurrent asynchronous *computation* by workers, (2) *communication* of intermediate results by workers to data storage, and (3) a *global barrier* which synchronizes workers and prevents them from proceeding with the next superstep before all workers complete their computation and communication.

We consider two ways that the BSP model has been directly applied to computing iterative transformations; we term these approaches as 'simple BSP'. Firstly, BSP may be used if the work of computing the change function Λ can be trivially factorized. For example, if $\Lambda(A, x) = Ax$ is a matrix-vector multiplication, we may distribute the rows of A across workers and compute each coordinate of Ax separately.

The second way batches multiple iterations together into a single BSP superstep, and uses an aggregation to combine the results from the iterations. Formally, let $B^{(t)} =$

$\{B_1^{(t)}, \ldots, B_m^{(t)}\} \subseteq \{1, \ldots, n\}$ be a batch of indices. The BSP computation computes

$$\lambda_i^{(t)} = \Lambda(\theta^{(t)}, \{x_{B_i^{(t)}}\}) \qquad i = 1, \ldots, m \tag{2.22}$$

$$\theta^{(t+1)} = T(T(\ldots T(T(\theta^{(t)}, \lambda_1^{(t)}), \lambda_2^{(t)}), \ldots, \lambda_{m-1}^{(t)}), \lambda_m^{(t)}) \tag{2.23}$$

This BSP superstep corresponds to the update

$$\theta^{(t,s+1)} = T(\theta^{(t,s)}, \Lambda(\theta^{(t,0)}, \{x_{B_i^{(t)}}\})) \tag{2.24}$$

$$\theta^{(t+1,0)} = \theta^{(t,|B^{(t)}|)} \tag{2.25}$$

Since the $\lambda^{(t)}$'s are computed from the same $\theta^{(t)}$, and they do not depend on other iterations in the same batch. In general, this update may not correspond to the below iteration of

$$\theta^{(t+1)} = T(\theta^{(t)}, \Lambda(\theta^{(t)}, B^{(t)})) \tag{2.26}$$

A restricted setting where (2.26) is equivalent to (2.22), (2.23) is when T is associative, commutative, and distributed by Λ, i.e., $\Lambda(\theta, B \cup B') = T(\Lambda(\theta, B), \Lambda(\theta, B'))$ for all θ, and disjoint B, B'. Then, we have

$$\begin{aligned}\theta^{(t+1)} &= T(T(\ldots T(T(\theta^{(t)}, \lambda_1^{(t)}), \lambda_2^{(t)}), \ldots, \lambda_{m-1}^{(t)}), \lambda_m^{(t)}) \\ &= T(\theta^{(t)}, T(\lambda_1^{(t)}, T(\lambda_2^{(t)}, T(\ldots T(\lambda_{m-1}^{(t)}, \lambda_m^{(t)}) \ldots)))) \\ &= T(\theta^{(t)}, \Lambda(\theta^{(t)}, B^{(t)}))\end{aligned} \tag{2.27}$$

Examples of algorithms that can be parallelized in this manner are minibatch SGD and the conditional Gibbs sampler for Beta-Bernoulli models.

The introduction of the MapReduce [43] programming paradigm has coincided with an increased interest in the application of simple BSP to parallelizing machine learning algorithms. [39] presented 10 machine learning algorithms that can be implemented on multicore using MapReduce / simple BSP. Modern distributed machine learning frameworks such as MLI [127] on Spark, and Tensorflow [1, 112] implement BSP minibatch stochastic optimization. GraphLab [97] and Petuum [135] are respectively general graph-processing and parameter server systems that provide BSP-style execution of graph-based or parameter-based iterations.

It is important to recognize that BSP is a coordination technique that may be used in ways beyond the abovementioned direct applications. In Chapters 4, 5, 7, we will use BSP in conjunction with other concurrency control mechanisms to either guarantee correctness or to simplify our speedup analyses. However, in existing literature, BSP has only been applied to either embarrassingly parallel problems (as above) or without additional coordination and thus requiring additional analysis of correctness (see [104] for example).

Coordination-free

The coordination-free approach simply executes iterations asynchronously in parallel without any coordination. As we pointed out in (2.19), the output of the t-th iteration is

$$\theta_k^{(t+1)} = \begin{cases} T_k(\theta_k^{(t-\sigma_{t,k})}, \Lambda(\{\theta_r^{(t-\tau_{t,r})} : r \in R(X^{(t)})\}, X^{(t)})) & \text{if } k \in W(X^{(t)}) \\ \theta_k^{(t)} & \text{otherwise} \end{cases} \tag{2.28}$$

where $\sigma_{t,k}$ and $\tau_{t,r}$ are arbitrary delays that the coordination-free algorithm does not attempt to control. This interleaving of concurrent iterations' reads and writes leaves us with an execution that almost surely does not correspond to any serial execution, and hence one may not appeal to the correctness of the serial algorithm when arguing the theoretical guarantees of the parallel algorithm. Instead, one typically assumes a bounded delay, i.e., that there is some small constant c such that $|\sigma_{t,k}| \leq c$ and $|\tau_{t,r}| \leq c$, which translate to a bound on the interference that iterations can have on one another. The approximation of the parallel algorithm can then in turn be bounded, and usually worsens with larger c. Intuitively, c should be in the order of the number of parallel workers, so increasing parallelism leads to poorer approximations.

On the other hand, the complete absence of coordination gives coordination-free algorithms (theoretical) ideal linear speedup.

Interest in coordination-free algorithms in recent years was first sparked by the seminal paper HogWild! [116] which empirically demonstrated that coordination-free SGD vastly outperformed SGD with locks. Furthermore, [116] provided a template for analyzing the convergence rate of a coordination-free stochastic optimization algorithm. Since then, various coordination-free stochastic optimization algorithms have been proposed and analyzed, including stochastic coordinate descent [95, 101]; dual averaging and AdaGrad [50]; Frank-Wolfe [133]; and SVRG [101]. Recent papers [101, 41] have introduced more unified analysis of coordination-free stochastic optimization algorithms; these papers view the interference of concurrent iterations as stochastic noise, an issue for which stochastic optimization algorithms are particularly well-suited to handle.

Coordination-free algorithms have also been proposed for Gibbs samplers, particularly for topic models such as LDA and HDP [106, 3, 126]. However, understanding the correctness or approximation of such samplers have proven more difficult. [70, 69] provided bounds on the error of a single iteration of coordination-free LDA sampling, whereas [73] was restricted to Gaussian distributions. A breakthrough was achieved in [42], which bounded both the bias and mixing times of Gibbs samplers for distributions that satisfy Dobrushin's condition (which intuitively states that no variable has a strong influence on any other variable), and under the assumption of bounded delays.

Systems that provide coordination-free executions include PowerGraph [62], parameter servers [89, 65], DistBelief [44], and TensorFlow [1].

Hybrid Approaches

In this section, we review some parallel machine learning approaches that do not fit nicely into our above dichotomy of either simple BSP or coordination-free.

Some of these approaches have used the BSP computation model but without the same guarantees of serializability. CoCoA [71, 98, 125] is a distributed primal-dual solver, where machines solve local subproblems in the compute phase, communicate their approximate solutions for aggregation, and then wait on the global synchronization barrier for the next BSP superstep. Distributed ADMM solvers [27] also solve local subproblems on subsets of data before a global consensus update. Splash [142] proposes running (reweighted) SGD locally on a subset of data in the compute phase before globally averaging the local parameters. In all three cases, the aggregate of local solutions is not equivalent to the solution one would have obtained from optimizing over the entire dataset. This dependence on the parallelism can be observed by the worsened rates of CoCoA and Splash with larger number of machines.

A different parallelization strategy begins with the coordination-free algorithm, but adds cheap coordination mechanisms to reduce interference between iterations. SoftSync [141] batches gradients from multiple workers together before performing a minibatch update, but does so without the BSP global synchronization barrier. Instead, gradients are aggregated whenever they are sent by workers, regardless of whether the worker had read an outdated and stale parameter. Stale Synchronous Parallel parameter servers [65] enforce workers to not drift apart by too many iterations; this turns the assumption of bounded delays into a systems guarantee. Such parallel algorithms serve as a middle ground between coordination-free and serializable methods. Nevertheless, because they are not in fact serializable, separate analysis is still required to prove correctness.

Next, we take a closer examination of two parallel machine learning systems that are particularly relevant to our approach.

GraphLab and PowerGraph

GraphLab [97] and PowerGraph [62] are parallel machine learning frameworks that exploit sparse structure common in machine learning problems. Iterative transformations are centered on individual vertices, and may read and write to data at the vertex, its edges, and its neighboring vertices. In particular, the change function Λ in PowerGraph is restricted to be commutative and associative, and acts only on an edge and its adjacent vertices. Both PowerGraph and GraphLab provide different modes of execution, including simple BSP, coordination-free, and asynchronous serializable (through either locking or BSP with supersteps over maximal independent sets of the graph).

Petuum

Petuum [135] is a general framework for parallel machine learning based on parameter servers. Petuum supports both data parallelism, where data $\mathcal{X}$ is distributed for computation across machines, and model parallelism, where the model parameters θ are partitioned. Roughly speaking, data parallelism corresponds to our presentation of simple BSP in (2.22), (2.23), (2.27), where λ's are computed across workers and aggregated for a single transformation. Model parallelism roughly corresponds to the iterative transformation defined by (2.19), with workers holding the parameters in the read sets $R(X^{(t)})$ and write sets $W(X^{(t)})$ of iterations which they are responsible for. However, serializability is not guaranteed by Petuum, as it implements Stale Synchronous Parallel [65] consistency, which guarantees bounded but non-zero delays. Petuum also tries to exploit 'non-uniform convergence' by scheduling iterations that it believes will lead to greatest improvements.

Chapter 3

Approach

The objective of this dissertation is to present and evaluate a general approach for parallelizing machine learning algorithms, expressible as iterative transformations, with the following characteristics:

- *Strong guarantees of correctness:* Theoretical guarantees of accuracy, approximation, etc. of our parallel machine learning algorithm are independent of the degree of parallelism, and no worse than those of the serial algorithm.
- *Scalable:* Our parallel machine learning algorithms demonstrate good empirical speedups. Ideally, we will provide guaranteed bounds on the speedups or coordination overheads of the parallel algorithms, under assumptions on the computational system and problem.
- *Verifiable and repeatable:* The output of our parallel machine learning algorithms can be checked. Ideally, multiple runs of the parallel algorithms generate the same output. Verifiability and repeatability are desirable properties for software engineering, testing, etc. and is important for the scientific process.

Our approach may be succinctly stated as applying concurrency control mechanisms for parallelizing iterative transformation algorithms, by coordinating concurrency iterations.

Specifically, we view the serial machine learning algorithm through the iterative transformation framework, and treat each iteration as a database transaction. By examining and understanding the semantics of the serial machine learning algorithm, we identify potential conflicts and interference between concurrent transactions, i.e., situations where the output of a transaction may influence the computation and output of another concurrent transaction. We then design concurrency control mechanisms for avoiding or resolving such conflicts; we choose techniques to incur the least expected coordination overheads for the algorithm and problem. In doing so, our parallel algorithms are always, by design, guaranteed to be *serializable*.

Any execution of our serializable parallel algorithm produces an outcome that is equivalent to some serial execution of the iterations, specifically, where $\mathcal{T}_t$ is executed as the $\pi(t)$-th

transaction. Since our serial algorithm admits arbitrary reordering of iterations as well, the output of the parallel algorithm is equivalent to a valid output of the serial algorithm under some ordering of iterations. Hence, serializability achieves the first objective of strong guarantees of correctness — theoretical properties of the parallel algorithms' outputs directly translate from that of the serial algorithms, independently of the degree of parallelism. Furthermore, if we log the transaction ordering of a serializable algorithm, we can always verify that an implementation of the algorithm is producing the correct output by comparing it against a serial run of the same transaction ordering.

In addition, most of the algorithms we parallelize using concurrency control will also be deterministic.

Definition 3.1 (Determinism)**.** *Let $\mathcal{A}$ be a parallel algorithm taking as input a permutation π and a set of non-random*[1] *transactions $\{\mathcal{T}_1, \ldots, \mathcal{T}_N\}$ where $\mathcal{T}_t = (T_t, \Lambda_t, X^{(t)})$, and let $\theta^{(t+1)}$ be defined as*

$$\theta_k^{(\pi(t)+1)} = \begin{cases} T_{t,k}(\theta_k^{(\pi(t))}, \Lambda_t(\{\theta_r^{(\pi(t))} : r \in R(X^{(t)})\}, X^{(t)})) & \text{if } k \in W(X^{(t)}) \\ \theta_k^{(\pi(t))} & \text{otherwise} \end{cases} \tag{3.1}$$

or more concisely

$$\theta^{(\pi(t)+1)} = T_t(\theta^{(\pi(t))}, \Lambda_t(\theta^{(\pi(t))}, X^{(t)})) \tag{3.2}$$

The parallel algorithm $\mathcal{A}$ is deterministic *if every execution of $\mathcal{A}$ generates output $\theta^{(\pi(N))}$.*

Like serializability, determinism is a statement about the parallel algorithm's output, and puts no constraint on the intermediate states during execution. However, determinism is a stronger condition than serializability, since it enforces that the algorithm produces an output according to the desired ordering. Determinism is also a sufficient (though not necessary) condition for repeatability — running the algorithm twice with the same transactions and ordering will reproduce the same output.

We illustrate our approach with the examples of sparse SGD and the Beta-Bernoulli marginal Gibbs sampler.

Example 3.1 (Concurrency Control for Sparse Stochastic Gradient Descent (SGD))**.** *We consider a specialization of Example 2.1 where the gradient $\nabla_\theta f_i(\theta; x_i)$ is dependent only on the non-zero support of x_i, which we denote $S_i = \{k \ : \ x_{i,k} \neq 0\}$. Letting $\theta_{S_i} = \{\theta_k \ : \ k \in S_i\}$, we can write the gradient as $\nabla_\theta f_i(\theta_{S_i}; x_i)$ to emphasize this dependence. Examples of problems with such structure include linear models such as linear regression and logistic regression. The sparse SGD update is given by*

$$\theta_k^{(t+1)} = \begin{cases} \theta_k^{(t)} - \gamma_t \nabla_\theta f_{i^{(t)}}(\theta_{S_{i^{(t)}}}; x_{i^{(t)}})\big|_{\theta_{S_{i^{(t)}}} = \theta^{(t)}_{S_{i^{(t)}}}} & \text{if } k \in S_{i^{(t)}} \\ \theta_k^{(t)} & \text{otherwise} \end{cases} \tag{3.3}$$

[1] Randomness of stochastic algorithms can be encapsulated as a seed for a psuedo-random generator within T_t, Λ_t, or $X^{(t)}$.

with read and write sets $R(\{x_{i^{(t)}}\}) = W(\{x_{i^{(t)}}\}) = S_{i^{(t)}}$.

A simple approach for parallelizing sparse SGD would be to apply locking mechanisms. Specifically, when a processor starts executing transaction t*, it begins by requesting locks on every coordinate in* $S_{i^{(t)}}$*. After the successful acquisition of all locks, the gradient is computed and the update is made to* $\theta_{S_{i^{(t)}}}$*. Finally, the transaction is completed when all locks are released.*

We point out that the abovementioned simple locking scheme has been previously investigated in [116] and shown to be heavy-handed in comparison with the proposed coordination-free approach. In Chapter 7 we propose a different concurrency control approach that amortizes the coordination overhead, and empirically outperforms the coordination-free approach in sparse settings.

Example 3.2 (Concurrency Control for Marginal Beta-Bernoulli Gibbs Sampler)**.** *The Bernoulli sampling of* (2.16) *can be expressed as a two-step process*

$$u_i \sim Uniform(0,1) \tag{3.4}$$

$$x_i | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n = \begin{cases} 0 & \text{if } u_i > \frac{1+\sum_{i' \neq i} x_{i'}}{1+n} \\ 1 & \text{otherwise} \end{cases} \tag{3.5}$$

Each sampling (3.5) *has read set* $R(\{x_i\}) = \mathcal{X} \backslash x_i$ *and write set* $W(\{x_i\}) = \{x_i\}$*, which could cause transactions to be executed serially if a naive locking technique were used. However, we observe that there is a weak dependence on each* $x_{i'}$*, in the sense that if we were to flip the value of* $x_{i'}$*, there is only a* $\frac{1}{1+n}$ *probability of the sampling of* x_i *being affected. If we have* p *other concurrent transactions, then the probability of sampling* x_i *being affected is* $\frac{p}{1+n}$*, which is small for* $p << n$*.*

We will keep the sum $N \stackrel{def}{=} \sum_{i' \neq i} x_{i'}$ *in our parallel algorithm to be read at the start of each transaction* $\mathcal{T}_t$*. While we cannot be sure that the read value of* N *will be equal to the true value of* N *at the point of* $\mathcal{T}_t$*'s serialization, we assume that the true value is bounded as* $\underline{N} \leq N \leq \overline{N}$*. Hence, we can assign*

$$x_i | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n = \begin{cases} 0 & \text{if } u_i(1+n) > 1 + \overline{N} \\ 1 & \text{if } u_i(1+n) < 1 + \underline{N} \\ UNK & \text{otherwise} \end{cases} \tag{3.6}$$

where UNK *is a placeholder token indicating that a decision cannot be made about* x_i*.*

A single process is then designated as the validator, which serializes the transactions and determines the true value of N*. As the validator processes each transaction, it checks that its assumption on the bounds hold, and that its proposed value of* x_i *is not* UNK*. If either of those conditions fail, the transaction is rejected and retried, by either the proposal process or by the validator itself; otherwise, the proposal is accepted and the transaction is committed, whereupon the true values of* x_i *and* N *are updated.*

The above two examples illustrate our approach for parallelizing iterative transformation machine learning algorithms. In both cases, we first turn the updates into a set of transactions; we then examine the semantics of the transactions' operations to identify potential conflicts. Depending on the nature of the conflicts and workload, we design concurrency control mechanisms to produce a serializable parallel algorithm. We emphasize that there is no single concurrency control mechanism that works for all iterative transformation algorithms. The validation approach of Example 3.2, for instance, is heavily tied to the sampling equations of the marginal Beta-Bernoulli Gibbs sampler. This trick could possibly be applied to other algorithms with discrete sampling (see CC-2G in Chapter 6, for example), but is not applicable to a problem such as the sparse SGD of Example 3.1.

By design, our parallel algorithms are serializable (or deterministic) and do not require further analysis of correctness. Work is required, however, to quantify the theoretical overheads of coordination and prove bounds on speedup. Assumptions on the system and / or problem are typically made for the sake of the concurrency analysis, but are not necessary for demonstrating correctness.

3.1 Comparison With Existing Parallel Machine Learning Approaches

We now compare our approach of using concurrency control with the existing approaches to parallel machine learning.

Simple BSP approaches

The simple BSP parallelization does not provide serializability except in easy cases, such as when the change function Λ can be trivially factorized, or when the transformation function T is commutative, associative, and distributed by Λ. While this covers many important algorithms [39] including minibatch gradient descent for stochastic optimization, the BSP abstraction fails when there are computational dependencies between iterations. Furthermore, despite being 'embarrassingly parallel', BSP solutions often do not provide linear speedups in practice. This failure may be due to poor systems efficiency (as processors have to wait for the slowest stragglers [112]) or poor statistical efficiency (e.g. larger batch sizes in minibatch SGD provides diminishing speedups [46, 88]).

Coordination-free approaches

The coordination-free and concurrency control approaches are dual to each other: while coordination-free approaches are engineered to be fast and then shown to be correct under assumptions, our concurrency control approach is engineered to be correct and then demonstrated to be fast under assumptions. We reiterate that correctness and concurrency are important key objectives of both approaches; the two approaches differ in the

choice of how the two objectives are traded-off against each other when dependencies in the problem and algorithm make it impossible to perfectly attain both simultaneously. The interactions between transactions leading to poorer approximations of the coordination-free approach are also points where our concurrency control approach needs to coordinate to ensure serializability. Therefore, the assumptions needed to demonstrate correctness of the coordination-free approach are often related to the assumptions we make to prove scalability. In the worst case, our parallel algorithms reduce to serial executions but maintain correctness through serializability, whereas coordination-free approaches remain fast but produce poor approximations.

We remark that despite the 'naive' parallelization, coordination-free approaches may not achieve linear speedups in practice, due to a combination of poor systems efficiencies (e.g. cache contention [108]) and poorer approximations with greater parallelism. Also, because coordination-free algorithms are non-deterministic and non-serializable, they are difficult to debug and test, and challenging to analyze.

GraphLab and PowerGraph

GraphLab [97] and PowerGraph [62] were among the first general machine learning systems to employ concurrency control ideas from databases. These systems explicitly capture transaction dependencies in advance through graph structure and then enforce these dependencies through read / write locks on vertices and edges. In contrast, we typically employ a more semantic notion of conflict, and generalize beyond simple read-write and write-write conflicts. Operations in PowerGraph are also restricted to only commutative and associative updates. Scalability and speedups in both systems are demonstrated empirically without theoretical analysis.

Petuum

Petuum implements Stale Synchronous Parallel consistency instead of serializability, and therefore suffers form the same qualitative issues as coordination-free approaches, namely, results are non-repeatable, and approximations that require separate analysis for each algorithm. While it is possible to enforce serializability by using a staleness setting of zero, this reduces Petuum to either BSP or simple locking (as no worker is allowed to read a parameter if another concurrent worker is computing an update for the parameter). Like GraphLab and PowerGraph, Petuum uses a read / write notion of conflict.

3.2 Comparison With Transactional Databases

Unlike the traditional concurrency control mechanisms implemented in databases, our work explores more general notions of conflict than read-write and write-write conflicts. For example, we are able to parallelize the marginal Beta-Bernoulli Gibbs sampler of Example

3.2 because we examined the semantics of the algorithm and exploited the weak dependencies implicit in the marginal Gibbs sampler. One would not achieve any parallelism if one were to plug this sampler directly into a standard transactional database.

Specializing the concurrency control approach to machine learning algorithms also affords us several advantages over traditional transactional databases.

1. *Throughput, not latency.* We are only concerned with minimizing the total runtime to complete all transactions, i.e., maximizing throughput, and not with the latency of each individual transaction. Thus, our algorithms may choose to defer transactions for later computation if it leads to a faster processing of other transactions.

2. *Durability is secondary.* The durability of individual transactions is unimportant, since the input itself serves as a checkpoint from which we may regenerate the algorithm's output. This is particularly true for deterministic algorithms, but also for serializable algorithms, because every transaction ordering leads to a valid outcome.

3. *Known workload.* All information about the workload, including data, computation, and read / write sets, are known upfront. Hence, we can choose the concurrency control technique that caters best to the algorithm's parallelism. Furthermore, one could implement algorithm- and data-specific optimizations, such as reordering transactions, or finding optimal distributions for the computation and storage.

Chapter 4

Nonparametric Unsupervised Learning

4.1 Introduction

Much of existing literature on distributed machine learning algorithms has focused on supervised learning, where the goal is to learn a mapping from datapoints to their respective labels.

In this chapter[1], we focus instead on unsupervised learning, where the goal is to learn a global latent model and encodings of each data point in the latent model space. A nonparametric unsupervised learning problem has a potentially infinite-dimensional latent model that grows with the dataset size. The algorithms we examine in this chapter iteratively search for the best encoding for each datapoint, growing the model as necessary to obtain better representations. Such algorithms pose a particular challenge for parallelization: each encoding is based on the current model, which could potentially be updated by other concurrent iterations.

We explore the use of optimistic concurrency control (OCC) for distributed nonparametric unsupervised learning. OCC exploits the infrequency of changes to the global latent model to allow most iterations to run in parallel without blocking or waiting; potential conflicts are detected through OCC validation and forced to serialize. Hence, our OCC algorithms are serializable, and analysis is only necessary to guarantee optimal scaling performance.

We apply OCC to distributed nonparametric unsupervised learning—including but not limited to clustering—and implement distributed versions of the DP-Means [84], BP-Means [29], and online facility location (OFL) algorithms. We demonstrate how to analyze OCC in the context of the DP-Means algorithm and evaluate the empirical scalability of the OCC approach on all three of the proposed algorithms. The primary contributions of this chapter are:

1. A concurrency control approach to distributing unsupervised learning algorithms.

[1] Work done as part of [109].

2. Reinterpretation of online nonparametric clustering in the form of facility location with approximation guarantees.
3. Analysis of optimistic concurrency control for unsupervised learning.
4. Application to feature modeling and clustering.

The rest of this chapter is organized as follows. In Section 4.2, we present a general pattern of nonparametric unsupervised learning algorithms in the iterative transformation framework. We then discuss the application of OCC to this general pattern in Section 4.3, with three concrete examples of OCC versions of nonparametric unsupervised learning algorithms. We theoretically analyze the correctness and overheads to scalability in Section 4.4, before empirically evaluating our distributed algorithms in Section 4.5. Related work is presented in Section 4.6, and we conclude in Section 4.7.

4.2 Nonparametric Unsupervised Learning

Many machine learning problems can be classified as supervised learning problems: given data $\{(x_i, y_i) \ : \ i = 1, \ldots, N\}$, the goal of supervised learning is to learn a function g that minimizes some regularized loss $r(g) + \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(g(x_i), y_i)$, where $\mathcal{L}(g(x), y)$ is a measure of difference between $g(x)$ and y, and r is a regularizer that prefers simpler functions. This problem is termed 'classification' when the labels $\{y_i\}$ are categorical, and 'regression' when $\{y_i\}$ are ordinal.

In this chapter, we focus instead on unsupervised learning problems, where one is given data $\{x_i\}$ with the goal of learning some latent structure of the data. The K-means algorithm provides a paradigm example; here the inferential goal is to find K cluster centers $\{\mu_j \ : \ j = 1, \ldots, K\}$ and an assignment $\{z_i \ : \ i = 1, \ldots, K\}$ where $z_i \in \{1, \ldots, K\}$ such that each center μ_j is representative of its cluster $\mathcal{C}_j = \{x_i \ : \ z_i = j\}$, i.e., $\mu_j = \frac{1}{|\mathcal{C}_j|}\sum_{x_i \in \mathcal{C}_j} x_i$ and $z_i = \arg\min_j ||x_i - \mu_j||_2^2$.

More generally, in unsupervised learning, one is given data $\{x_i\}$ and seeks to learn a global latent model $\mathfrak{M}$ and individual encodings $\{z_i\}$ to minimize $r(\mathfrak{M}) + \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\textsc{Decode}(\mathfrak{M}, z_i), x_i)$. Here, $\textsc{Decode}(\mathfrak{M}, z_i)$ generates a representation for x_i using the model $\mathfrak{M}$ and encoding z_i; for example, the decoder for K-means is $\textsc{Decode}(\{\mu_j\}, z_i) = \mu_{z_i}$. As before, $\mathcal{L}(\cdot, \cdot)$ is a measure of difference, and r is a regularizer.

The algorithms we examine in this chapter have the general pattern given in Algorithm 4.1, iterating over datapoints to generate encodings based on the current model (Line 3). The algorithm then checks (Line 4) that the encoding z_i is a sufficiently good representation of the datapoint x_i; if not, it extends the model $\mathfrak{M}$ (Line 5) so as to obtain a better encoding (Line 7).

The general nonparametric unsupervised learning pattern Algorithm 4.1 can be cast as

Algorithm 4.1: Nonparametric Unsupervised Learning

Input: data $\{x_i : i = 1, \ldots, N\}$
1 **while** *not converged* **do**
2 **for** i = *1 to* N **do**
3 $z_i \leftarrow \textsc{Encode}(\mathfrak{M}, x_i)$
4 **if not** $\textsc{isGoodRepresentation}(\textsc{Decode}(\mathfrak{M}, z_i), x_i)$ **then**
5 $\mathfrak{M}^{new} \leftarrow \textsc{ExtendModel}(\mathfrak{M}, z_i, x_i)$
6 $\mathfrak{M} \leftarrow \mathfrak{M} \cup \mathfrak{M}^{new}$
7 $z_i \leftarrow \textsc{Encode}(\mathfrak{M}, x_i)$
8 $\mathfrak{M} \leftarrow \textsc{UpdateModel}(\mathfrak{M}, \{z_i\}, \{x_i\})$
Output: $\mathfrak{M}$, $\{z_i : i = 1, \ldots, N\}$

an iterative algorithm, using the change function Λ_i defined as

$$z_i' = \textsc{Encode}(\mathfrak{M}, x_i)$$

$$[\Lambda_i((\mathfrak{M}, z_i), x_i)]_{\mathfrak{M}} = \begin{cases} \mathfrak{M} & \text{if } \textsc{isGoodRepresentation}(\textsc{Decode}(\mathfrak{M}, z_i'), x_i) \\ \mathfrak{M} \cup \textsc{ExtendModel}(\mathfrak{M}, z_i', x_i) & \text{otherwise} \end{cases}$$

$$[\Lambda_i((\mathfrak{M}, z_i), x_i)]_{z_i} = \textsc{Encode}([\Lambda_i((\mathfrak{M}, z_i), x_i)]_{\mathfrak{M}}, x_i)$$

and the transformation function T_i as the identity function.

4.3 Optimistic Concurrency Control for Nonparametric Unsupervised Learning

The challenge of parallelizing Algorithm 4.1 lies in the fact that each iteration (transaction) depends on the model $\mathfrak{M}$ but could also potentially extend $\mathfrak{M}$. We make two key observations that enable a scalable parallelization: first, as the algorithm progresses, $\mathfrak{M}$ is transformed into an increasingly better representation, and thus ExtendModel is infrequently triggered; secondly, if a transaction's isGoodRepresentation test passes, it may be safely committed without affecting serializability, since it does not alter the shared global model $\mathfrak{M}$. Our OCC-inspired parallelization exploits these facts to allow most transactions to proceed without blocking or waiting, only serializing them infrequently when model extensions are proposed. This general approach is encapsulated in Algorithm 4.2. Most transactions complete upon finding a good encoding (Line 4); some transactions propose extensions to the model and need to be validated (Line 6). The validation is done serially (Line 7), repeating the transaction's work using the latest model.

Algorithm 4.2 is serializable, and also repeatable if we assume datapoints are processed in order within the validation loop (Line 7). However, Algorithm 4.2 reorders transactions to minimize blocking and waiting, and hence is not deterministic. A deterministic execution can be obtained with additional overhead, as we show in Algorithm 4.3. In the first parallel loop (Line 4), all datapoints are processed in parallel to identify potential model extensions which are then serially validated (Line 8). The second parallel loop (Line 13) then completes the

Algorithm 4.2: OCC Serializable Nonparametric Unsupervised Learning

Input: data $\{x_i : i = 1, \ldots, N\}$
1 **while** *not converged* **do**
2 $V = \emptyset$
3 **for** i = *1 to* N **do in parallel**
4 $z_i \leftarrow \textsc{Encode}(\mathfrak{M}, x_i)$
5 **if not** $\textsc{isGoodRepresentation}(\textsc{Decode}(\mathfrak{M}, z_i), x_i)$ **then**
6 $V \leftarrow V \cup \{i\}$ // Add to V for validation
// Serially validate
7 **for** $i \in V$ **do**
8 $z_i \leftarrow \textsc{Encode}(\mathfrak{M}, x_i)$
9 **if not** $\textsc{isGoodRepresentation}(\textsc{Decode}(\mathfrak{M}, z_i), x_i)$ **then**
10 $\mathfrak{M}^{new} \leftarrow \textsc{ExtendModel}(\mathfrak{M}, z_i, x_i)$
11 $\mathfrak{M} \leftarrow \mathfrak{M} \cup \mathfrak{M}^{new}$
12 $z_i \leftarrow \textsc{Encode}(\mathfrak{M}, x_i)$
13 $\mathfrak{M} \leftarrow \textsc{UpdateModel}(\mathfrak{M}, \{z_i\}, \{x_i\})$
Output: $\mathfrak{M}$, $\{z_i : i = 1, \ldots, N\}$

encoding of each z_i using the updated model comprising of all model extensions up to time i. We assume that $\textsc{Encode}$ is linear in $\mathfrak{M}$, i.e., there exists a merge operator $\sqcup$ such that $\textsc{Encode}(\mathfrak{M} \cup \mathfrak{M}', x) = \textsc{Encode}(\mathfrak{M}, x) \sqcup \textsc{Encode}(\mathfrak{M}', x)$, which allows us to reuse work and write Line 15 instead of $z_i \leftarrow \textsc{Encode}(\mathfrak{M}_0 \cup \mathfrak{M}_i^+, x_i)$.

Algorithm 4.3: OCC Deterministic Nonparametric Unsupervised Learning

Input: data $\{x_i : i = 1, \ldots, N\}$
1 **while** *not converged* **do**
2 $V = \emptyset$
3 $\mathfrak{M}_0 \leftarrow \mathfrak{M}$
// Check for model extension proposals
4 **for** i = *1 to* N **do in parallel**
5 $z_i \leftarrow \textsc{Encode}(\mathfrak{M}, x_i)$
6 **if not** $\textsc{isGoodRepresentation}(\textsc{Decode}(\mathfrak{M}, z_i), x_i)$ **then**
7 $V \leftarrow V \cup \{i\}$ // Add to V for validation
// Serially validate model extensions
8 **for** $i \in V$ **do**
9 $z_i \leftarrow \textsc{Encode}(\mathfrak{M}, x_i)$
10 **if not** $\textsc{isGoodRepresentation}(\textsc{Decode}(\mathfrak{M}, z_i), x_i)$ **then**
11 $\mathfrak{M}_i^{new} \leftarrow \textsc{ExtendModel}(\mathfrak{M}, z_i, x_i)$
12 $\mathfrak{M} \leftarrow \mathfrak{M} \cup \mathfrak{M}_i^{new}$
// Commit transactions with correct models
13 **for** i = *1 to* N **do in parallel**
14 $\mathfrak{M}_i^+ \leftarrow \bigcup_{i' \le i} \mathfrak{M}_{i'}^{new}$
15 $z_i \leftarrow z_i \sqcup \textsc{Encode}(\mathfrak{M}_i^+, x_i)$
16 $\mathfrak{M} \leftarrow \textsc{UpdateModel}(\mathfrak{M}, \{z_i\}, \{x_i\})$
Output: $\mathfrak{M}$, $\{z_i : i = 1, \ldots, N\}$

In the sequel, we will present and analyze only the serializable versions, omitting the obvious extension to deterministic algorithms.

Next, we present concrete OCC parallelization for three nonparametric unsupervised learning algorithms from recent work, which have been obtained by taking Bayesian nonparametric (BNP) models based on combinatorial stochastic processes such as the Dirichlet process, the beta process, and hierarchical versions of these processes, and subjecting them to *small-variance asymptotics* where the posterior probability under the BNP model is transformed into a cost function that can be optimized [29]. The algorithms considered to date in this literature have been developed and analyzed in the serial setting; our goal is to explore distributed algorithms for optimizing these cost functions that preserve the structure and analysis of their serial counterparts.

OCC-DP-MEANS

Algorithm 4.4: Serial DP-means

Input: data $\{x_i\}_{i=1}^N$, threshold α
$\mathcal{C} \leftarrow \emptyset$
while *not converged* **do**
  for $i = 1$ *to* N **do**
    $\mu^* \leftarrow \text{argmin}_{\mu \in \mathcal{C}} \|x_i - \mu\|$
    if $\|x_i - \mu^*\| > \alpha$ **then**
      $z_i \leftarrow x_i$
      $\mathcal{C} \leftarrow \mathcal{C} \cup x_i$ // New cluster
    else
      $z_i \leftarrow \mu^*$ // Use nearest
  for $\mu \in \mathcal{C}$ **do** // Recompute Centers
    $\mu \leftarrow$ Mean($\{x_i \mid z_i = \mu\}$)
Output: Accepted cluster centers $\mathcal{C}$

Algorithm 4.5: DPValidate

Input: Set of proposed cluster centers $\hat{\mathcal{C}}$
$\mathcal{C} \leftarrow \emptyset$
for $x \in \hat{\mathcal{C}}$ **do**
  $\mu^* \leftarrow \text{argmin}_{\mu \in \mathcal{C}} \|x - \mu\|$
  if $\|x_i - \mu^*\| < \alpha$ **then** // Reject
    $x \leftarrow \mu^*$ // Rollback Assgs
  else
    $\mathcal{C} \leftarrow \mathcal{C} \cup x$ // Accept
Output: Accepted cluster centers $\mathcal{C}$

Algorithm 4.6: Parallel DP-means

Input: data $\{x_i\}_{i=1}^N$, threshold α
Input: Epoch size b and P processors
Input: Partitioning $\mathcal{B}(p,t)$ of data $\{x_i\}_{i \in \mathcal{B}(p,t)}$ to processor-epochs where $b = |\mathcal{B}(p,t)|$
$\mathcal{C} \leftarrow \emptyset$
while *not converged* **do**
  for *epoch* $t = 1$ *to* $N/(Pb)$ **do**
    $\hat{\mathcal{C}} \leftarrow \emptyset$ // New candidate centers
    for $p \in \{1, \ldots, P\}$ **do in parallel**
      // Process local data
      for $i \in \mathcal{B}(p,t)$ **do**
        $\mu^* \leftarrow \text{argmin}_{\mu \in \mathcal{C}} \|x_i - \mu\|$
        // Optimistic Transaction
        if $\|x_i - \mu^*\| > \alpha$ **then**
          $z_i \leftarrow x_i$
          $\hat{\mathcal{C}} \leftarrow \hat{\mathcal{C}} \cup x_i$
        else
          $z_i \leftarrow \mu^*$ // Always Safe
    // Serially validate clusters
    $\mathcal{C} \leftarrow \mathcal{C} \cup$ DPValidate($\hat{\mathcal{C}}$)
  for $\mu \in \mathcal{C}$ **do** // Recompute Centers
    $\mu \leftarrow$ Mean($\{x_i \mid z_i = \mu\}$)
Output: Accepted cluster centers $\mathcal{C}$

We first consider the *DP-means* algorithm (Algorithm 4.4) introduced by [84]. Like the K-means algorithm, DP-Means alternates between updating the cluster assignment z_i for each point x_i and recomputing the centroids $\mathcal{C} = \{\mu_k\}_{k=1}^K$ associated with each clusters. However, DP-Means differs in that the number of clusters is not fixed a priori. Instead, if the distance from a given data point to all existing cluster centroids is greater than a parameter α, then a new cluster is created. While the second phase is trivially parallel, the process

of introducing clusters in the first phase is inherently serial. However, clusters tend to be introduced infrequently, and thus DP-Means provides an opportunity for OCC.

In Algorithm 4.6 we present an OCC parallelization of the DP-Means algorithm in which each iteration of the serial DP-Means algorithm is divided into $N/(Pb)$ bulk-synchronous epochs. The data is evenly partitioned $\{x_i\}_{i\in\mathcal{B}(p,t)}$ across processor-epochs into blocks of size $b = |\mathcal{B}(p,t)|$. During each epoch t, each processor p evaluates the cluster membership of its assigned data $\{x_i\}_{i\in\mathcal{B}(p,t)}$ using the cluster centers $\mathcal{C}$ from the previous epoch and optimistically proposes a new set of cluster centers $\hat{\mathcal{C}}$. At the end of each epoch the proposed cluster centers, $\hat{\mathcal{C}}$, are *serially* validated using Algorithm 4.5. The validation process accepts cluster centers that are not covered by (i.e., not within α of) already accepted cluster centers. When a cluster center is rejected we update its reference to point to the already accepted center, thereby correcting the original point assignment.

OCC Facility Location

Algorithm 4.7: Parallel OFL (serializable)

Input: Same as DP-Means

for *epoch* $t = 1$ *to* $N/(Pb)$ **do** $\hat{\mathcal{C}} \leftarrow \emptyset$
  for $p \in \{1, \ldots, P\}$ **do in parallel**
    for $i \in \mathcal{B}(p,t)$ **do**
      $d \leftarrow \min_{\mu\in\mathcal{C}} \|x_i - \mu\|$
      with probability $\min\left\{d^2, \alpha^2\right\}/\alpha^2$
        $\hat{\mathcal{C}} \leftarrow \hat{\mathcal{C}} \cup (x_i, d)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup$ `OFLValidate`$(\hat{\mathcal{C}})$

Output: Accepted cluster centers $\mathcal{C}$

Algorithm 4.8: OFLValidate (serializable)

Input: Set of proposed cluster centers $\hat{\mathcal{C}}$

$\mathcal{C} \leftarrow \emptyset$
for $(x, d) \in \hat{\mathcal{C}}$ **do**
  $d^* \leftarrow \min_{\mu\in\mathcal{C}} \|x - \mu\|$
  with probability $\min\left\{d^{*2}, d^2\right\}/d^2$
    $\mathcal{C} \leftarrow \mathcal{C} \cup x$ `// Accept`

Output: Accepted cluster centers $\mathcal{C}$

The DP-Means objective turns out to be equivalent to the classic Facility Location (FL) objective:

$$J(\mathcal{C}) = \sum_{x\in X} \min_{\mu\in\mathcal{C}} \|x - \mu\|^2 + \alpha^2|\mathcal{C}|,$$

which selects the set of cluster centers (facilities) $\mu \in \mathcal{C}$ that minimizes the shortest distance $\|x - \mu\|$ to each point (customer) x as well as the penalized cost of the clusters $\alpha^2\,|\mathcal{C}|$. However, while DP-Means allows the clusters to be arbitrary points (e.g., $\mathcal{C} \in \mathbb{R}^D$), FL constrains the clusters to be points $\mathcal{C} \subseteq \mathcal{F}$ in a set of candidate locations $\mathcal{F}$. Hence, we obtain a link between combinatorial Bayesian models and FL allowing us to apply algorithms with known approximation bounds to Bayesian inspired nonparametric models. As we will see in Section 4.4, our OCC algorithm provides constant-factor approximations for both FL and DP-means.

Facility location has been studied intensely. We build on the *online* facility location (OFL) algorithm described by Meyerson [102]. The OFL algorithm processes each data

Algorithm 4.9: Parallel OFL (deterministic)

Input: Same as DP-Means
Input: $\{u_i : i = 1, \ldots, N\}$ where $u_i \sim Uniform(0, 1)$
for *epoch* t *= 1 to* $N/(Pb)$ **do** $\hat{\mathcal{C}} \leftarrow \emptyset$
  for $p \in \{1, \ldots, P\}$ **do in parallel**
    for $i \in \mathcal{B}(p, t)$ **do**
      $d \leftarrow \min_{\mu \in \mathcal{C}} \|x_i - \mu\|$
      if $u_i \leq \min\left\{d^2, \alpha^2\right\}/\alpha^2$ **then**
        $\hat{\mathcal{C}} \leftarrow \hat{\mathcal{C}} \cup (x_i, d)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup$ `OFLValidate`$(\hat{\mathcal{C}})$
Output: Accepted cluster centers $\mathcal{C}$

Algorithm 4.10: OFLValidate (deterministic)

Input: Set of proposed cluster centers $\hat{\mathcal{C}}$
$\mathcal{C} \leftarrow \emptyset$
for $(x, d) \in \hat{\mathcal{C}}$ **do**
  $d^* \leftarrow \min_{\mu \in \mathcal{C}} \|x - \mu\|$
  if $u_i \leq \min\left\{d^{*2}, \alpha^2\right\}/\alpha^2$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup x$ `// Accept`
Output: Accepted cluster centers $\mathcal{C}$

point x serially in a single pass by either adding x to the set of clusters with probability $\min(1, \min_{\mu \in \mathcal{C}} \|x - \mu\|^2 / \alpha^2)$ or assigning x to the nearest existing cluster. Using OCC we are able to construct a distributed OFL algorithm (Algorithm 4.7) which is nearly identical to the OCC-DP-means algorithm (Algorithm 4.6) but which provides strong approximation bounds. The OCC-OFL algorithm differs only in that clusters are introduced and validated stochastically—the validation process ensures that the new clusters are accepted with probability equal to the serial algorithm.

OCC-BP-means

BP-means is an algorithm for learning collections of latent binary *features*, providing a way to define groupings of data points that need not be mutually exclusive or exhaustive like clusters.

As with serial DP-means, there are two phases in serial BP-means (Algorithm 4.11). In the first phase, each data point x_i is labeled with binary assignments from a collection of features ($z_{ik} = 0$ if x_i doesn't belong to feature k; otherwise $z_{ik} = 1$) to construct a representation $x_i \approx \sum_k z_{ik} f_k$. In the second phase, parameter values (the feature means $f_k \in \hat{\mathcal{C}}$) are updated based on the assignments. The first step also includes the possibility of introducing an additional feature. While the second phase is trivially parallel, the inherently serial nature of the first phase combined with the infrequent introduction of new features points to the usefulness of OCC in this domain.

The OCC parallelization for BP-means follows the same basic structure as OCC-DP-means. Each transaction operates on a data point x_i in two phases. In the first, analysis phase, the optimal representation $\sum_k z_{ik} f_k$ is found. If x_i is not well represented (i.e., $\|x_i - \sum_k z_{ik} f_k\| > \alpha$), the difference is proposed as a new feature in the second validation phase. At the end of epoch t, the proposed features $\{f_i^{new}\}$ are serially validated to obtain a set of accepted features $\tilde{\mathcal{C}}$. For each proposed feature f_i^{new}, the validation process first

finds the optimal representation $f_i^{new} \approx \sum_{f_k \in \tilde{\mathcal{C}}} z_{ik} f_k$ using *newly accepted features*. If f_i^{new} is not well represented, the difference $f_i^{new} - \sum_{f_k \in \tilde{\mathcal{C}}} z_{ik} f_k$ is added to $\tilde{\mathcal{C}}$ and accepted as a new feature.

Finally, to update the feature means, let F be the K-row matrix of feature means. The feature means update $F \leftarrow (Z^T Z)^{-1} Z^T X$ can be evaluated as a single transaction by computing the sums $Z^T Z = \sum_i z_i z_i^T$ (where z_i is a $K \times 1$ column vector so $z_i z_i^T$ is a $K \times K$ matrix) and $Z^T X = \sum_i z_i x_i^T$ in parallel.

Here we show the Serial BP-Means algorithm (Algorithm 4.11) and a parallel implementation of BP-means using the OCC pattern (Algorithm 4.12 and Algorithm 4.13), similar to OCC-DP-MEANS. Instead of proposing new clusters centered at the data point x_i, in OCC-BP-MEANS we propose features f_i^{new} that allow us to obtain perfect representations of the data point. The validation process continues to improve on the representation $x_i \approx \sum_k z_{ik} f_k$ by using the most recently accepted features $f_{k'} \in \hat{\mathcal{C}}$, and only accepts a proposed feature if the data point is still not well-represented.

Algorithm 4.11: Serial BP-means

```
   Input: data {x_i}_{i=1}^N, threshold α
1  Initialize z_i1 = 1, f_1 = N^{-1} Σ_i x_i, K = 1
2  while not converged do
3      for i = 1 to N do
4          for k = 1 to K do
5              Set z_ik to minimize ‖x_i − Σ_{j=1}^K z_ij f_j‖_2^2
6          if ‖x_i − Σ_{j=1}^K z_ij f_{i,j}‖_2^2 > α^2 then
7              Set K ← K + 1
8              Create feature f_K ← x_i − Σ_{k=1}^K z_ik f_j
9              Assign z_iK ← 1 (and z_iK ← 0 for j ≠ i)
10     F ← (Z^T Z)^{-1} Z^T X
```

4.4 Analysis of Correctness and Scalability

We now establish the correctness and scalability of the proposed OCC algorithms. In contrast to the coordination-free pattern in which scalability is trivial and correctness often requires strong assumptions or holds only in expectation, the OCC pattern leads to simple proofs of correctness and challenging scalability analysis. In many cases it is preferable to have algorithms that are correct and probably fast rather than fast and possibly correct.

We first establish serializability:

Theorem 4.1 (Serializability)**.** *The distributed DP-means, OFL, and BP-means algorithms are serially equivalent to DP-means, OFL and BP-means, respectively.*

The proof (Appendix 4.A) of Theorem 4.1 is relatively straightforward and is obtained by constructing a permutation function that describes an equivalent serial execution for each

Algorithm 4.12: Parallel BP-means

Input: data $\{x_i\}_{i=1}^N$, threshold α
Input: Epoch size b and P processors
Input: Partitioning $\mathcal{B}(p,t)$ of data $\{x_i\}_{i\in\mathcal{B}(p,t)}$ to processor-epochs where $b = |\mathcal{B}(p,t)|$

$\mathcal{C} \leftarrow \emptyset$
while *not converged* **do**
  for *epoch* $t = 1$ *to* $N/(Pb)$ **do**
    $\hat{\mathcal{C}} \leftarrow \emptyset$ `// New candidate features`
    for $p \in \{1, \ldots, P\}$ **do in parallel**
      `// Process local data`
      for $i \in \mathcal{B}(p,t)$ **do**
        `// Optimistic Transaction`
        for $f_k \in \mathcal{C}$ **do**
          Set z_{ik} to minimize $\|x_i - \sum_j z_{ij} f_j\|_2^2$
        if $\|x_i - \sum_j z_{ij} f_j\|_2^2 > \alpha^2$ **then**
          $f_i^{new} \leftarrow x_i - \sum_j z_{ij} f_j$
          $z_i \leftarrow z_i \oplus f_i^{new}$
          $\hat{\mathcal{C}} \leftarrow \hat{\mathcal{C}} \cup f_i^{new}$
    `// Serially validate features`
    $\mathcal{C} \leftarrow \mathcal{C} \cup$ `BPValidate`$(\hat{\mathcal{C}})$
  Compute $Z^T Z = \sum_i z_i z_i^T$ and $Z^T X = \sum_i z_i x_i^T$ in parallel
  Re-estimate features $F \leftarrow (Z^T Z)^{-1} Z^T X$
Output: Accepted feature centers $\mathcal{C}$

Algorithm 4.13: `BPValidate`

Input: Set of proposed feature centers $\hat{\mathcal{C}}$

$\mathcal{C} \leftarrow \emptyset$
for $f^{new} \in \hat{\mathcal{C}}$ **do**
  for $f_{k'} \in \mathcal{C}$ **do**
    Set $z_{ik'}$ to minimize $\|f^{new} - \sum_{f_j \in \mathcal{C}} z_{ij} f_j\|_2^2$
  if $\|f^{new} - \sum_{f_j \in \mathcal{C}} z_{ij} f_j\|_2^2 > \alpha^2$ **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \left\{ f^{new} - \sum_{f_j \in \mathcal{C}} z_{ij} f_j \right\}$
  $f^{new} \leftarrow \{z_{ij}\}_{f_j \in \mathcal{C}}$
Output: Accepted feature centers $\mathcal{C}$

distributed execution. This proof technique can easily be extended to many other machine learning algorithms, as we will show in later chapters.

Serializability allows us to easily extend important theoretical properties of the serial algorithm to the distributed setting. For example, by invoking serializability, we can establish the following result for the OCC version of the online facility location (OFL) algorithm:

Lemma 4.2. *If the data is randomly ordered, then the* OCC-OFL *algorithm provides a constant-factor approximation for the DP-means objective. If the data is adversarially ordered, then* OCC-OFL *provides a log-factor approximation to the DP-means objective.*

The proof (Appendix 4.A) of Lemma 4.2 is first derived in the serial setting then extended to the distributed setting through serializability. In contrast to divide-and-conquer schemes, whose approximation bounds commonly depend *multiplicatively* on the number of levels [103], Lemma 4.2 is unaffected by distributed processing and has no communication or coarsening tradeoffs. Furthermore, to retain the same factors as a batch algorithm on the full data, divide-and-conquer schemes need a large number of preliminary centers at lower levels [103, 4]. In that case, the communication cost can be high, since all proposed clusters are sent at the same time, as opposed to the OCC approach. We address the communication overhead (the number of rejections) for our scheme next.

Scalability The scalability of the OCC algorithms depends on the number of transactions that are rejected during validation (i.e., the rejection rate). While a general scalability analysis can be challenging, it is often possible to gain some insight into the asymptotic dependencies by making simplifying assumptions. In contrast to the coordination-free approach, we can still *safely* apply OCC algorithms in the absence of a scalability analysis or when simplifying assumptions do not hold.

To illustrate the techniques employed in OCC scalability analysis we study the DP-Means algorithm. The scalability limiting factor of the DP-Means algorithm is determined by the number of points that must be serially validated. In the following theorem we show that the communication cost only depends on the number of clusters and processing resources and does not directly depend on the number of data points. The proof is in App. 4.B.

Theorem 4.3 (DP-Means Scalability)**.** *Assume N data points are generated iid to form a random number (K_N) of well-spaced clusters of diameter α: α is an upper bound on the distances within clusters and a lower bound on the distance between clusters. Then the expected number of* serially *validated points is bounded above by $Pb + \mathbf{E}[K_N]$ for P processors and b points per epoch.*

Under the separation assumptions of the theorem, the number of clusters present in N data points, K_N, is exactly equal to the number of clusters found by DP-Means in N data points; call this latter quantity k_N. The experimental results in Figure 4.1 suggest that the bound of $Pb + k_N$ may hold more generally beyond the assumptions above. Since the master must process at least k_N points, the overhead caused by rejections is Pb and independent of N.

To analyze the total running time, we note that after each of the $N/(Pb)$ epochs the master and workers must communicate. Each worker must process N/P data points, and the master sees at most $k_N + Pb$ points. Thus, the total expected running time is $O(N/(Pb) + N/P + Pb)$.

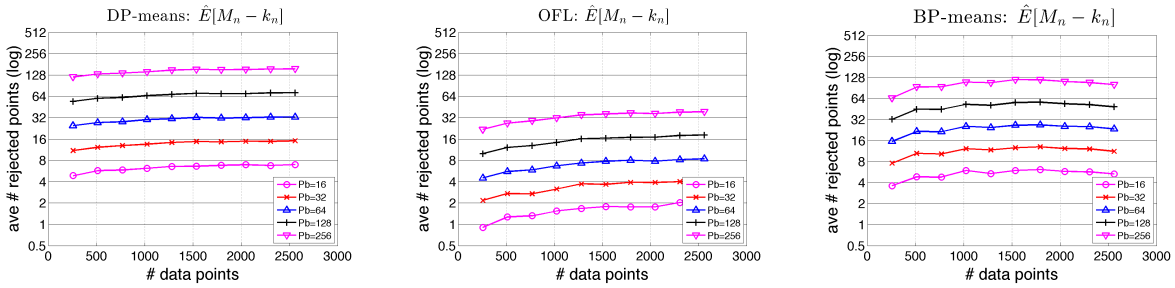
4.5 Evaluation

For our experiments, we generated synthetic data for clustering (DP-means and OFL) and feature modeling (BP-means). The cluster and feature proportions were generated nonparametrically as described below. All data points were generated in $\mathbb{R}^{16}$ space. The threshold parameter α was fixed at 1.

Clustering: The cluster proportions and indicators were generated simultaneously using the stick-breaking procedure for Dirichlet processes—'sticks' are 'broken' on-the-fly to generate new clusters as and when necessary.[2] For our experiments, we used a fixed concentration parameter $\theta = 1$. Cluster means were sampled $\mu_k \sim N(0, I_{16})$, and data points were generated at $x_i \sim N(\mu_{z_i}, \frac{1}{4}I_{16})$.

Feature modeling: We use the stick-breaking procedure of [107] to generate feature weights. Unlike with Dirichlet processes, we are unable to perform stick-breaking on-the-fly with Beta processes. Instead, we generate enough features so that with high probability (> 0.9999) the remaining non-generated features will have negligible weights (< 0.0001). The concentration parameter was also fixed at $\theta = 1$. We generated feature means $f_k \sim N(0, I_{16})$ and data points $x_i \sim N(\sum_k z_{ik} f_k, \frac{1}{4}I_{16})$.

Simulated experiments



(a) OCC-DP-means (b) OCC-OFL (c) OCC-BP-means

Figure 4.1: Simulated distributed DP-means, OFL and BP-means: expected number of data points proposed but not accepted as new clusters / features is independent of size of data set.

To test the efficiency of our algorithms, we simulated the first iteration (one complete pass over all the data, where most clusters / features are created and thus greatest coordination is needed) of each algorithm in MATLAB. The number of data points, N, was varied from 256 to 2560 in intervals of 256. We also varied Pb, the number of data points processed in one epoch, from 16 to 256 in powers of 2. For each value of N and Pb, we empirically measured k_N, the number of accepted clusters / features, and M_N, the number of proposed clusters

[2]We chose to use stick-breaking procedures because the Chinese restaurant and Indian buffet processes are inherently sequential. Stick-breaking procedures can be distributed by either truncation, or using OCC!

/ features. This was repeated 400 times to obtain the empirical average $\hat{\mathbb{E}}[M_N - k_N]$, the number of rejections.

For OCC-DP-MEANS, we observe $\hat{\mathbb{E}}[M_N - k_N]$ is bounded above by Pb (Fig. 4.1a), and that this bound is independent of the data set size, even when the assumptions of Thm 4.3 are violated. (We also verified that similar empirical results are obtained when the assumptions are not violated; see Appendix 4.B.) As shown in Fig. 4.1b and Fig. 4.1c the same behavior is observed for the OCC-OFL and OCC-BP-MEANS algorithms.

Distributed implementation and experiments

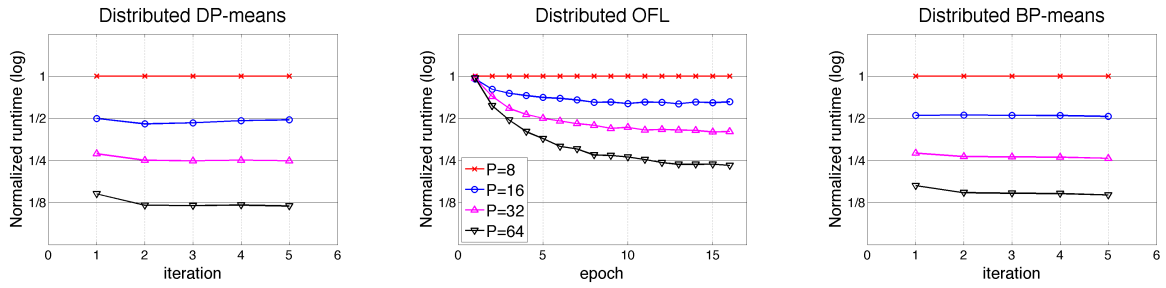
We also implemented the distributed algorithms in Spark [139], an open-source cluster computing system. The DP-means and BP-means algorithms were bootstrapped by pre-processing a small number of data points (1/16 of the first Pb points)—this reduces the number of data points sent to the master on the first epoch, while still preserving serializability of the algorithms. Our Spark implementations were tested on Amazon EC2 by processing a fixed data set on 1, 2, 4, 8 m2.4xlarge (memory-optimized, quadruple extra large, with 8 virtual cores and 64.8GiB memory) instances.

Ideally, to process the same amount of data, an algorithm and implementation with perfect scaling would take half the runtime on 8 machines as it would on 4, and so on. The plots in Figure 4.2 shows this comparison by dividing all runtimes by the runtime on one machine.

DP-means: We ran the distributed DP-means algorithm on $2^{27} \approx 134$M data points, using $\alpha = 2$. The block size b was chosen to keep $Pb = 2^{23} \approx 8$M constant. The algorithm was run for 5 iterations (complete pass over all data in 16 epochs). We were able to get perfect scaling (Figure 4.2a) in all but the first iteration, when the master has to perform the most synchronization of proposed centers.

OFL: The distributed OFL algorithm was run on $2^{20} \approx 1$M data points, using $\alpha = 2$. Unlike DP-means and BP-means, we did not perform bootstrapping, as it would not significantly improve speed-up. Also, OFL is a single pass (one iteration) algorithm. The block size b was chosen such that $Pb = 2^{16} \approx 66$K data points are processed each epoch, which gives us 16 epochs. Figure 4.2b shows that we get no scaling in the first epoch, where all the work is performed by the master processing all Pb data points. In later epochs, the master's workload decreases as fewer data points are proposed, but the workers' workload increases as the total number of centers increases. Thus, scaling improves in the later epochs.

BP-means: Distributed BP-means was run on $2^{23} \approx 8$M data points, with $\alpha = 1$; block size was chosen such that $Pb = 2^{19} \approx 0.5$M is constant. Five iterations were run, with 16 epochs per iteration. As with DP-means, we were able to achieve nearly perfect scaling; see Figure 4.2c.



(a) OCC-DP-MEANS (b) OCC-OFL (c) OCC-BP-MEANS

Figure 4.2: Normalized runtime for distributed algorithms. Runtime of each iteration / epoch is divided by that using 1 machine ($P = 8$). Ideally, the runtime with 2, 4, 8 machines ($P = 16, 32, 64$) should be respectively 1/2, 1/4, 1/8 of the runtime using 1 machine. OCC-DP-MEANS and BP-means obtain nearly perfect scaling for all iterations. OCC-OFL rejects a lot initially, but quickly gets better in later epochs.

4.6 Additional Related Work

Others have proposed alternatives to locking and coordination-free parallelism for machine learning algorithm design. Newman [106] proposed transforming the underlying model to expose additional parallelism while preserving the marginal posterior. However, such constructions can be challenging or infeasible and many hinder mixing or convergence. Likewise, Lovell [96] proposed a reparameterization of the underlying model to expose additional parallelism through conditional independence.

Additional work similar in spirit to ours using OCC-like techniques includes [49] who proposed an approximate parallel sampling algorithm for the IBP which is made exact by introducing an additional Metropolis-Hastings step, and [136] who proposed a look-ahead strategy in which future samples are computed optimistically based on the likely outcomes of current samples.

A great amount of work addresses scalable clustering algorithms [48, 40, 53]. Many algorithms with provable approximation factors are streaming algorithms [103, 124, 33] and inherently use hierarchies, or related divide-and-conquer approaches [4]. The approximation factors in such algorithms multiply across levels [103], and demand a careful tradeoff between communication and approximation quality that is obviated in our framework. Other approaches use core sets [12, 55]. A lot of methods [4, 13, 124] first collect a set of centers and then re-cluster them, and therefore need to communicate all intermediate centers. Our approach avoids that, since a center causes no rejections in the epochs after it is established: the rejection rate does not grow with K. Still, as our examples demonstrate, our OCC framework can easily integrate and exploit many of the ideas in the cited works.

4.7 Discussion

In this chapter we have shown how optimistic concurrency control can inspire the design of distributed machine learning algorithms, preserving correctness, in most cases at a small cost. We established the equivalence of our distributed OCC DP-means, OFL and BP-means algorithms to their serial counterparts, thus preserving their theoretical properties. In particular, the strong approximation guarantees of serial OFL translate immediately to the distributed algorithm. Our theoretical analysis ensures OCC-DP-MEANS achieves high parallelism without sacrificing correctness. We implemented and evaluated all three OCC algorithms on a distributed computing platform and demonstrate strong scalability in practice.

4.A Proof of Serializability of Distributed Algorithms

Proof of Theorem 4.1 for DP-means

We note that both distributed DP-means and BP-means iterate over z-updates and cluster / feature means re-estimation until convergence. In each iteration, distributed DP-means and BP-means perform the same set of updates as their serial counterparts. Thus, it suffices to show that each iteration of the distributed algorithm is serially equivalent to an iteration of the serial algorithm.

Consider the following ordering on transactions:

- Transactions on individual data points are ordered before transactions that re-estimate cluster / feature means are ordered.
- A transaction on data point x_i is ordered before a transaction on data point x_j if
  1. x_i is processed in epoch t, x_j is processed in epoch t', and $t < t'$
  2. x_i and x_j are processed in the same epoch, x_i and x_j are not sent to the master for validation, and $i < j$
  3. x_i and x_j are processed in the same epoch, x_i is not sent to the master for validation but x_j is
  4. x_i and x_j are processed in the same epoch, x_i and x_j are sent to the master for validation, and the master serially validates x_i before x_j

We show below that the distributed algorithms are equivalent to the serial algorithms under the above ordering, by inductively demonstrating that the outputs of each transaction is the same in both the distributed and serial algorithms.

Denote the set of clusters after the t epoch as $\mathcal{C}^t$.

The first transaction on x_j in the serial ordering has $\mathcal{C}^0$ as its input. By definition of our ordering, this transaction belongs the first epoch, and is either (1) not sent to the master for

validation, or (2) the first data point validated at the master. Thus in both the serial and distributed algorithms, the first transaction either (1) assigns x_j to the closest cluster in $\mathcal{C}^0$ if $\min_{\mu_k \in \mathcal{C}^0} \|x_j - \mu_k\| < \alpha$, or (2) creates a new cluster with center at x_j otherwise.

Now consider any other transaction on x_j in epoch t.

Case 1: x_j is not sent to the master for validation.

In the distributed algorithm, the input to the transaction is $\mathcal{C}^{t-1}$. Since the transaction is not sent to the master for validation, we can infer that there exists $\mu_k \in \mathcal{C}^{t-1}$ such that $\|x_j - \mu_k\| < \alpha$.

In the serial algorithm, x_j is ordered after any x_i if (1) x_i was processed in an earlier epoch, or (2) x_i was processed in the same epoch but not sent to the master (i.e. does not create any new cluster) and $i < j$. Thus, the input to this transaction is the set of clusters obtained at the end of the previous epoch, $\mathcal{C}^{t-1}$, and the serial algorithm assigns x_j to the closest cluster in $\mathcal{C}^{t-1}$ (which is less than α away).

Case 2: x_j is sent to the master for validation.

In the distributed algorithm, x_j is not within α of any cluster center in $\mathcal{C}^{t-1}$. Let $\hat{\mathcal{C}}^t$ be the new clusters created at the master in epoch t before validating x_j. The distributed algorithm either (1) assigns x_j to $\mu_{k^*} = \operatorname{argmin}_{\mu_k \in \hat{\mathcal{C}}^t} \|x_j - \mu_k\|$ if $\|x_j - \mu_k\| \leq \alpha$, or (2) creates a new cluster with center at x_j otherwise.

In the serial algorithm, x_j is ordered after any x_i if (1) x_i was processed in an earlier epoch, or (2) x_i was processed in the same epoch t, but x_i was not sent to the master (i.e. does not create any new cluster), or (3) x_i was processed in the same epoch t, x_i was sent to the master, and serially validated at the master before x_j. Thus, the input to the transaction is $\mathcal{C}^{t-1} \cup \hat{\mathcal{C}}^t$. We know that x_j is not within α of any cluster center in $\mathcal{C}^{t-1}$, so the outcome of the transaction is either (1) assign x_j to $\mu_{k^*} = \operatorname{argmin}_{\mu_k \in \hat{\mathcal{C}}^t} \|x_j - \mu_k\|$ if $\|x_j - \mu_k\| \leq \alpha$, or (2) create a new cluster with center at x_j otherwise. This is exactly the same as the distributed algorithm.

Proof of Theorem 4.1 for BP-means

The serial ordering for BP-means is exactly the same as that in DP-means. The proof for the serializability of BP-means follows the same argument as in the DP-means case, except that we perform feature assignments instead of cluster assignments.

Proof of Theorem 4.1 for OFL

Here we prove Theorem 4.1 that the distributed OFL algorithm is equivalent to a serial algorithm.

(Theorem 4.1, OFL). We show that with respect to the returned centers (facilities), the distributed OFL algorithm is equivalent to running the serial OFL algorithm on a particular

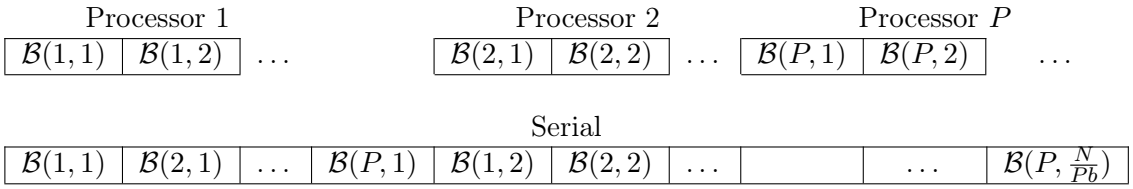


Figure 4.3: Illustration of distributed and serial order of blocks $\mathcal{B}(i,t)$ of length b for OFL. The order within each block is maintained. Block $\mathcal{B}(i,t)$ is processed in epoch t by processor p_i.

permutation of the input data. We assume that the input data is randomly permuted and the indices i of the points x_i refer to this permutation. We assign the data points to processors by assigning the first b points to processor p_1, the next b points to processor p_2, and so on, cycling through the processors and assigning them batches of b points, as illustrated in Figure 4.3. In this respect, our ordering is generic, and can be adapted to any assignments of points to processors. We assume that each processor visits its points in the order induced by the indices, and likewise the master processes the points of an epoch in that order.

For the serial algorithm, we will use the following ordering of the data: Point x_i precedes point x_j if

1. x_i is processed in epoch t and x_j is processed in epoch t', and $t < t'$, or
2. x_i and x_j are processed in the same epoch and $i < j$.

If the data is assigned to processors as outlined above, then the serial algorithm will process the points exactly in the order induced by the indices. That means the set of points processed in any given epoch t is the same for the serial and distributed algorithm. We denote by $\mathcal{C}^t$ the global set of validated centers collected by OCC-OFL up to (including) epoch t, and by $\tilde{\mathcal{C}}^i$ the set of centers collected by the serial algorithm up to (including) point x_i.

We will prove the equivalence inductively.

Epoch $t = 1$. In the first epoch, all points are sent to the master. These are the first Pb points. Since the master processes them in the same order as the serial algorithm, the distributed and serial algorithms are equivalent.

Epoch $t > 1$. Assume that the algorithms are equivalent up to point x_{i-1} in the serial order, and point x_i is processed in epoch t. By assumption, the set $\mathcal{C}^{t-1}$ of global facilities for the distributed algorithm is the same as the set $\tilde{\mathcal{C}}^{(t-1)Pb}$ collected by the serial algorithm up to point $x_{(t-1)Pb}$. For notational convenience, let $D(x_i, \mathcal{C}^t) = \min_{\mu \in \mathcal{C}^t} D(x_i, \mu)$ be the distance of x_i to the closest global facility.

The essential issue to prove is the following claim:

Claim 4.4. *If the algorithms are equivalent up to point x_{i-1}, then the probability of x_i becoming a new facility is the same for the distributed and serial algorithm.*

The serial algorithm accepts x_i as a new facility with probability $\min\{1, D(x_i, \tilde{\mathcal{C}}^{i-1})/\alpha^2\}$. The distributed algorithm sends x_i to the master with probability $\min\{1, D(x_i, \mathcal{C}^{t-1})\}$. The probability of ultimate acceptance (validation) of x_i as a global facility is the probability of being sent to the master *and* being accepted by the master. In epoch t, the master receives a set of candidate facilities with indices between $(t-1)Pb+1$ and tPb. It processes them in the order of their indices, i.e., all candidates x_j with $j < i$ are processed before i. Hence, the assumed equivalence of the algorithms up to point x_{i-1} implies that, when the master processes x_i, the set $\mathcal{C}^{t-1} \cup \hat{\mathcal{C}}$ equals the set of facilities $\tilde{\mathcal{C}}^{i-1}$ of the serial algorithm. The master consolidates x_i as a global facility with probability 1 if $D(x_i, \tilde{\mathcal{C}}^{i-1} \cup \hat{\mathcal{C}}) > \alpha^2$ and with probability $D(x_i, \tilde{\mathcal{C}}^{i-1} \cup \hat{\mathcal{C}})/D(x_i, \mathcal{C}^{t-1})$ otherwise.

We now distinguish two cases. If the serial algorithm accepts x_i because $D(x_i, \tilde{\mathcal{C}}^{i-1}) \geq \alpha^2$, then for the distributed algorithm, it holds that

$$D(x_i, \mathcal{C}^{t-1}) \geq D(x_i, \mathcal{C}^{t-1} \cup \hat{\mathcal{C}}) = D(x_i, \tilde{\mathcal{C}}^{i-1}) \geq \alpha^2 \tag{4.1}$$

and therefore the distributed algorithm also always accepts x_i.

Otherwise, if $D(x_i, \tilde{\mathcal{C}}^{i-1}) < \alpha^2$, then the serial algorithm accepts with probability $D(x_i, \tilde{\mathcal{C}}^{i-1})/\alpha^2$. The distributed algorithm accepts with probability

$$\mathbb{P}(x_i \text{ accepted}) = \mathbb{P}(x_i \text{ sent to master }) \cdot \mathbb{P}(x_i \text{ accepted at master}) \tag{4.2}$$

$$= \frac{D(x_i, \mathcal{C}^{t-1})}{\alpha^2} \cdot \frac{D(x_i, \tilde{\mathcal{C}}^{i-1} \cup \hat{\mathcal{C}})}{D(x_i, \mathcal{C}^{t-1})} \tag{4.3}$$

$$= \frac{D(x_i, \tilde{\mathcal{C}}^{i-1})}{\alpha^2}. \tag{4.4}$$

This proves the claim.

The claim implies that if the algorithms are equivalent up to point x_{i-1}, then they are also equivalent up to point x_i. This proves the theorem. □

Proof of Lemma 4.2 (Approximation Bound)

We begin by relating the results of facility location algorithms and DP-means. Recall that the objective of DP-means and FL is

$$J(\mathcal{C}) = \sum_{x \in X} \min_{\mu \in \mathcal{C}} \|x - \mu\|^2 + \alpha^2 |\mathcal{C}|. \tag{4.5}$$

In FL, the facilities may only be chosen from a pre-fixed set of centers (e.g., the set of all data points), whereas DP-means allows the centers to be arbitrary, and therefore be the empirical mean of the points in a given cluster. However, choosing centers from among the data points still gives a factor-2 approximation. Once we have established the corresponding clusters, shifting the means to the empirical cluster centers never hurts the objective. The following proposition has been a useful tool in analyzing clustering algorithms:

Proposition 4.5. *Let $\mathcal{C}^*$ be an optimal solution to the DP-means problem (4.5), and let $\mathcal{C}^{FL}$ be an optimal solution to the corresponding FL problem, where the centers are chosen from the data points. Then*

$$J(\mathcal{C}^{FL}) \leq 2J(\mathcal{C}^*).$$

Proof. (Proposition 4.5) It is folklore that Proposition 4.5) holds for the K-means objective, i.e.,

$$\min_{\mathcal{C}\subseteq X, |\mathcal{C}|=k} \sum_{i=1}^{n} \min_{\mu\in\mathcal{C}} \|x_i - \mu\|^2 \leq 2 \min_{\mathcal{C}\subseteq X} \sum_{i=1}^{n} \min_{\mu\in\mathcal{C}} \|x_i - \mu\|^2. \tag{4.6}$$

In particular, this holds for the optimal number $K^* = |\mathcal{C}^*|$. Hence, it holds that

$$J(\mathcal{C}^{\mathrm{FL}}) \leq \min_{\mathcal{C}\subseteq X, |\mathcal{C}|=K^*} \sum_{i=1}^{n} \min_{\mu\in\mathcal{C}} \|x_i - \mu\|^2 + \alpha^2 K^* \leq 2J(\mathcal{C}^*). \tag{4.7}$$

□

With this proposition at hand, all that remains is to prove an approximation factor for the FL problem.

Proof. (Lemma 4.2) First, we observe that the proof of Theorem 4.1 implies that, for any random order of the data, the OCC and serial algorithm process the data in exactly the same way, performing ultimately exactly the same operations. Therefore, any approximation factor that holds for the serial algorithm straightforwardly holds for the OCC algorithm too.

Hence, it remains to prove the approximation factor of the serial algorithm. Let $C_1^{\mathrm{FL}}, \ldots, C_k^{\mathrm{FL}}$ be the clusters in an optimal solution to the FL problem, with centers $\mu_1^{\mathrm{FL}}, \ldots \mu_k^{\mathrm{FL}}$. We analyze each optimal cluster individually. The proof follows along the lines of the proofs of Theorems 2.1 and 4.2 in [102], adapting it to non-metric squared distances. We show the proof for the constant factor, the logarithmic factor follows analogously by using the ring-splitting as in [102].

First, we see that the expected total cost of any point x is bounded by the distance to the closest open facility y that is present when x arrives. If we always count in the distance of $\|x-y\|^2$ into the cost of x, then the expected cost is $\gamma(x) = \alpha^2\|x-y\|^2/\alpha^2 + \|x-y\|^2 = 2\|x-y\|^2$.

We consider an arbitrary cluster C_i^* and divide it into $|C^*|/2$ *good* points and $|C^*|/2$ *bad points*. Let $D_i = \frac{1}{|\mathcal{C}^{\mathrm{FL}}|}\sum_{x\in C_i^*} \|x-\mu_i\|$ be the average service cost of the cluster, and let d_g and d_b be the service cost of the good and bad points, respectively (i.e., $D_i = (d_g + d_b)/|C_i^{\mathrm{FL}}|$). The good points satisfy $\|x - \mu_i^{\mathrm{FL}}\| \leq 2D_i$. Suppose the algorithm has chosen a center, say y, from the points C_i^{FL}. Then any other point $x \in C_i^{\mathrm{FL}}$ can be served at cost at most

$$\|x-y\|^2 \leq \left(\|x-\mu_i^{\mathrm{FL}}\| + \|y-\mu_i^{\mathrm{FL}}\|\right)^2 \leq 2\|x-\mu_i^{\mathrm{FL}}\|^2 + 4D_i. \tag{4.8}$$

That means once the algorithm has established a good center within C_i^{FL}, all other good points together may be serviced within a constant factor of the total optimal service cost

of C^{FL}, i.e., at $2d_g + 4(d_g + d_b)$. The assignment cost of all the good points in C_i^{FL} that are passed before opening a good facility is, by construction of the algorithm and expected waiting times, in expectation α^2. Hence, in expectation, the cost of the good points in C_i^{FL} will be bounded by $\sum_{x\mathrm{good}} \gamma(x) \leq 2(2d_g + 4d_g + 4d_b + \alpha^2)$.

Next, we bound the expected cost of the bad points. We may assume that the bad points are injected randomly in between the good points, and bound the servicing cost of a bad point $x_b \in C_i^{\mathrm{FL}}$ in terms of the closest good point $x_g \in C_i^{\mathrm{FL}}$ preceding it in our data sequence. Let y be the closest open facility to μ_i^{FL} when y arrives. Then

$$\|x_b - y\|^2 \leq 2\|y - \mu_i^{\mathrm{FL}}\|^2 + 2\|x_b - \mu^{\mathrm{FL}}\|^2. \tag{4.9}$$

Now assume that x_g was assigned to y'. Then

$$\|y - \mu_i^{\mathrm{FL}}\|^2 \leq \|y' - \mu_i^{\mathrm{FL}}\|^2 \leq 2\|y' - x_g\|^2 + 2\|x_g - \mu^{\mathrm{FL}}\|^2. \tag{4.10}$$

From (4.9) and (4.8), it then follows that

$$\|x_b - y\|^2 \leq 4\|y' - x_g\|^2 + 4\|x_g - \mu^{\mathrm{FL}}\|^2 + 2\|x_b - \mu^{\mathrm{FL}}\|^2 \tag{4.11}$$

$$= 2\gamma(x_g) + 4\|x_g - \mu^{\mathrm{FL}}\|^2 + 2\|x_b - \mu^{\mathrm{FL}}\|^2. \tag{4.12}$$

Since the data is randomly permuted, x_g could be, with equal probability, any good point, and in expectation we will average over all good points.

Finally, with probability $2/|C_i^{\mathrm{FL}}|$ there is no good point before x_g. In that case, we will count in x_b as the most costly case of opening a new facility, incurring cost α^2. In summary, we can bound the expected total cost of C^{FL} by

$$\sum_{x \text{ good}} \gamma(x) + \sum_{x \text{ bad}} \gamma(x)$$

$$\leq 12d_g + 8d_b + \alpha^2 + \frac{2C^{\mathrm{FL}}}{2C^{\mathrm{FL}}}\alpha^2 + 2(2\frac{2|C_i^{\mathrm{FL}}|}{2|C^{\mathrm{FL}}|}(12d_g + 8d_b + \alpha^2) + 4d_g + 2d_b) \tag{4.13}$$

$$\leq 68d_g + 42d_b + 4\alpha^2 \leq 68J(\mathcal{C}^{\mathrm{FL}}). \tag{4.14}$$

This result together with Proposition 4.5 proves the lemma. □

4.B Proof of Master Processing Bound for DP-means (Theorem 4.3)

Proof. As in the theorem statement, we assume P processors, b points assigned to each processor per epoch, and N total data points. We further assume a generative model for the cluster memberships: namely, that they are generated iid from an arbitrary distribution $(\pi_j)_{j=1}^{\infty}$. That is, we have $\sum_{j=1}^{\infty} \pi_j = 1$ and, for each j, $\pi_j \in [0, 1]$. We see that there are perhaps infinitely many latent clusters. Nonetheless, in any data set of finite size N, there

will of course be only finitely many clusters to which any data point in the set belongs. Call the number of such clusters K_N.

Consider any particular cluster indexed by j. At the end of the first epoch in which a worker sees j, that worker (and perhaps other workers) will send some data point from j to the master. By construction, some data point from j will belong to the collection of cluster centers at the master by the end of the processing done at the master and therefore by the beginning of the next epoch. It follows from our assumption (all data points within a single cluster are within a α diameter) that no other data point from cluster j will be sent to the master in future epochs. It follows from our assumption about the separation of clusters that no points in other clusters will be covered by any data point from cluster j.

Let S_j represent the (random) number of points from cluster j sent to the master. Since there are Pb points processed by workers in a single epoch, N_j is constrained to take values between 0 and Pb. Further, note that there are a total of $N/(Pb)$ epochs.

Let $A_{j,s,t}$ be the event that the master is sent s data points from cluster j in epoch t. All of the events $\{A_{j,s,t}\}$ with $s = 1, \ldots, Pb$ and $t = 1, \ldots, N/(Pb)$ are disjoint. Define $A'_{j,0}$ to be the event that, for all epochs $t = 1, \ldots, N/(Pb)$, zero data points are sent to the master; i.e., $A'_{j,0} := \bigcup_t A_{j,0,t}$. Then $A'_{j,0}$ is also disjoint from the events $\{A_{j,s,t}\}$ with $s = 1, \ldots, Pb$ and $t = 1, \ldots, N/(Pb)$. Finally,

$$A'_{j,0} \cup \bigcup_{s=1}^{Pb} \bigcup_{t=1}^{N/(Pb)} A_{j,s,t}$$

covers all possible data configurations. It follows that

$$\mathbb{E}[S_j] = 0 * \mathbb{P}[A'_{j,0}] + \sum_{s=1}^{Pb} \sum_{t=1}^{N/(Pb)} s\mathbb{P}[A_{j,s,t}] = \sum_{s=1}^{Pb} \sum_{t=1}^{N/(Pb)} s\mathbb{P}[A_{j,s,t}]$$

Note that, for s points from cluster j to be sent to the master at epoch t, it must be the case that no points from cluster j were seen by workers during epochs $1, \ldots, t-1$, and then s points were seen in epoch t. That is, $\mathbb{P}[A_{j,s,t}] = (1-\pi_j)^{Pb(t-1)} \cdot \binom{Pb}{s} \pi_j^s (1-\pi_j)^{Pb-s}$.

Then

$$\begin{aligned}
\mathbb{E}[S_j] &= \left(\sum_{s=1}^{Pb} s \binom{Pb}{s} \pi_j^s (1-\pi_j)^{Pb-s} \right) \cdot \left(\sum_{t=1}^{N/(Pb)} (1-\pi_j)^{Pb(t-1)} \right) \\
&= \pi_j Pb \cdot \frac{1-(1-\pi_j)^{Pb \cdot N/(Pb)}}{1-(1-\pi_j)^{Pb}},
\end{aligned}$$

where the last line uses the known, respective forms of the expectation of a binomial random variable and of the sum of a geometric series.

To proceed, we make use of a lemma.

Lemma 4.6. *Let m be a positive integer and $\pi \in (0, 1]$. Then*

$$\frac{1}{1-(1-\pi)^m} \leq \frac{1}{m\pi} + 1.$$

Proof. A particular subcase of Bernoulli's inequality tells us that, for integer $l \leq 0$ and real $x \geq -1$, we have $(1+x)^l \geq 1 + lx$. Choose $l = -m$ and $x = -\pi$. Then

$$\begin{aligned}
&(1-\pi)^m \leq \frac{1}{1+m\pi} \\
\Leftrightarrow\ &1-(1-\pi)^m \geq 1 - \frac{1}{1+m\pi} = \frac{m\pi}{1+m\pi} \\
\Leftrightarrow\ &\frac{1}{1-(1-\pi)^m} \leq \frac{m\pi+1}{m\pi} = \frac{1}{m\pi} + 1.
\end{aligned}$$

□

We can use the lemma to find the expected total number of data points sent to the master:

$$\begin{aligned}
\mathbb{E}\sum_{j=1}^{\infty} S_j = \sum_{j=1}^{\infty} \mathbb{E}S_j &= \sum_{j=1}^{\infty} \pi_j Pb \cdot \frac{1-(1-\pi_j)^N}{1-(1-\pi_j)^{Pb}} \\
&\leq \sum_{j=1}^{\infty} \pi_j Pb \cdot \left(1 + \frac{1}{\pi_j Pb}\right) \cdot \left(1-(1-\pi_j)^N\right) \\
&= Pb \sum_{j=1}^{\infty} \pi_j \left(1-(1-\pi_j)^N\right) + \sum_{j=1}^{\infty} \left(1-(1-\pi_j)^N\right) \\
&\leq Pb + \sum_{j=1}^{\infty} \mathbb{P}(\text{cluster } j \text{ occurs in the first } N \text{ points}) \\
&= Pb + \mathbb{E}[K_N].
\end{aligned}$$

Conversely,

$$\mathbb{E}\sum_{j=1}^{\infty} S_j \geq \sum_{j=1}^{\infty} \pi_j Pb = Pb.$$

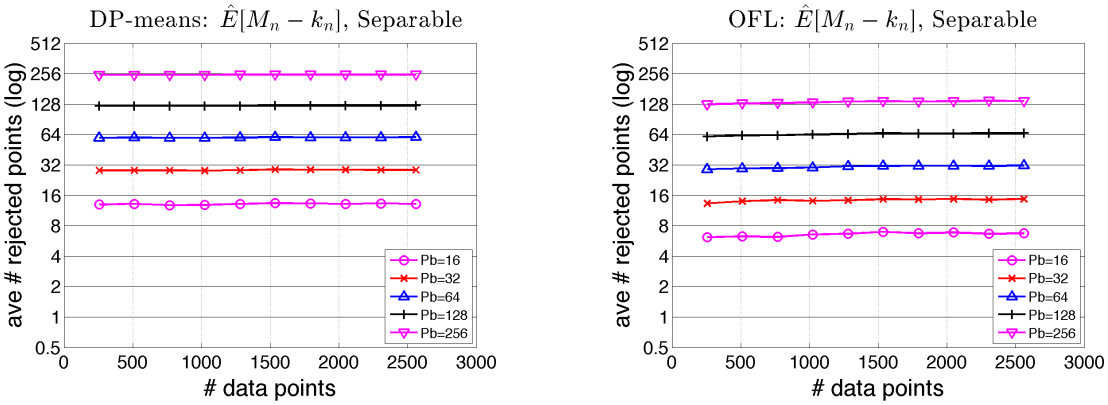
□

Experiment

To demonstrate the bound on the expected number of data points proposed but not accepted as new centers, we generated synthetic data with separable clusters. Cluster proportions are generated using the stick-breaking procedure for the Dirichlet process, with concentration

parameter $\theta = 1$. Cluster means are set at $\mu_k = (2k, 0, 0, \ldots, 0)$, and generated data uniformly in a ball of radius $1/2$ around each center. Thus, all data points from the same cluster are at most distance 1 from one another, and more than distance of 1 from any data point from a different cluster.

We follow the same experimental framework in Section 4.5.



(a) DP-means, separable

(b) OFL, separable

Figure 4.4: Simulated distributed DP-means and OFL: expected number of data points proposed but not accepted as new clusters is independent of size of data set.

In the case where we have separable clusters (Figure 4.4), $\hat{\mathbb{E}}[M_N - k_N]$ is bounded from above by Pb, which is in line with the above Theorem 4.3.

Chapter 5

Correlation Clustering

5.1 Introduction

Clustering items according to some notion of similarity is a major primitive in machine learning. Correlation clustering (CC) 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.

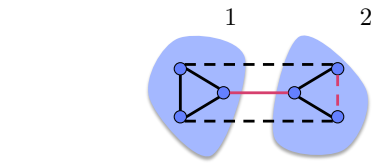


Figure 5.1: In the above graph, solid edges denote similarity and dashed dissimilarity. The number of disagreeing edges in the above clustering is 2; we color these edges with red.

Entity deduplication is the archetypal motivating example for correlation clustering, with applications in chat disentanglement, co-reference resolution, and spam detection [51, 8, 52, 68, 24, 37]. 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 [137, 31], gene clustering [15], and consensus clustering [52].

KwikCluster is the simplest CC algorithm that achieves a provable 3-approximation ratio [5], 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 [52].

KwikCluster is an inherently sequential iterative-transformation 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 [24, 37]. In [37] 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 [24], 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. In this chapter[1], 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. Specifically, we use a combination of BSP, locking-through-waiting, and monotone operations to achieve serializability. Locking is used to prevent adjacent cluster centers, whereas monotone operations ensure non-centers are assigned to the correct clusters. Our use of BSP for C4 serves only a theoretical purpose to aid analysis of C4's speedup; removal of the BSP barriers does not affect serializability, and is in fact faster in practice.

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

[1] Work done as part of [110].

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 $\{1, \ldots, n\}$.

5.2 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:*

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

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 [24], if we substitute the random choice of a vertex per peeling round, with a random order[2] π preassigned to vertices, (see Algorithm 5.1). 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] In an abuse of notation, we will use $\pi(i)$ to refer to the ith vertex in the ordering π, and $\pi(v)$ (instead of $\pi^{-1}(v)$) to refer to the position of v in this ordering.

Algorithm 5.1: KwikCluster with π

```
1 π = a random permutation of {1, ..., n}
2 while V ≠ ∅ do
3 |   Select the vertex v indexed by π(1)
4 |   C_v = {v, N(v)}
5 |   Remove clustered vertices from G and π
```

Algorithm 5.2: KwikCluster with π, as iterative transformation algorithm

```
1 π = a random permutation of {1, ..., n}
2 clusterID(1) = ... = clusterID(n) = ∞
3 for i = 1, ..., |V| do
4 |   v = π(i)
5 |   if clusterID(v) ≠ ∞ then continue
6 |   clusterID(v) = π(v)
7 |   for u ∈ Γ(v) do
8 |   |   clusterID(u) = min(clusterID(u), π(v))
```

We may also view KwikCluster through the iterative transformation framework of Chapter 2, which we present in Algorithm 5.2. Specifically, we use the change and transformation functions

$$\Lambda_v(\text{clusterID}) = \begin{cases} \emptyset & \text{if clusterID}(v) \neq \infty \\ \{v\} \cup (\Gamma(v) \cap \{u \ : \ \text{clusterID}(u) = \infty\}) & \text{if clusterID}(v) = \infty \end{cases}$$

$$[T_v(\text{clusterID}, \lambda)]_u = \begin{cases} \text{clusterID}(u) & \text{if } u \in \lambda \\ \pi(v) & \text{if } u \in \lambda \end{cases}$$

C4: 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 #1. Now, assume that v and u are not neighbors in G, and 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 Algorithm 5.1: w will go with the vertex that has the smallest permutation number, in this case v. This is concurency rule #2. Following the above simple rules, we develop C4, our serializable parallel CC algorithm. Since, C4 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 [20] to obtain a parallel algorithm for maximal independent set (MIS).

C4, shown as Algorithm 5.3, 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 #1: adjacent vertices cannot be cluster centers at the same time. C4 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, essentially implementing a lock on the variable clusterID(u). 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 [83].

Since C4 is serializable, it has to respect concurrency rule #2: if a vertex u is adjacent to two cluster centers, then it gets assigned to the one with smaller permutation order. We observe that the assignment uses a monotone min operator. It has recently been shown [64, 7] that monotone operations can be executed without coordination and still achieve a (eventually) consistent state. We exploit the monotonicity property in `createCluster` to perform the cluster assignments without further coordination.

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 5.3: C4 & ClusterWild!

Input: G, ϵ
clusterID$(1) = \ldots =$ clusterID$(n) = \infty$
$\pi =$ a random permutation of $\{1, \ldots, n\}$
while $V \neq \emptyset$ **do**
  $\Delta =$ maximum vertex degree in $G(V)$
  $\mathcal{A} =$ the first $\epsilon \cdot \frac{n}{\Delta}$ vertices in $V[\pi]$
  while $\mathcal{A} \neq \emptyset$ **do in parallel**
    $v =$ first element in $\mathcal{A}$
    $\mathcal{A} = \mathcal{A} - \{v\}$
    if C4 **then**
      `attemptCluster`(v)
    else if ClusterWild! **then**
      `createCluster`(v)
  Remove clustered vertices from V and π
Output: {clusterID$(1), \ldots,$ clusterID(n)}.

Algorithm 5.4: `createCluster`(v)

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

Algorithm 5.5: `attemptCluster`(v)

if *clusterID*$(u) = \infty$ *and* *isCenter*(v) **then**
  `createCluster`(v)

Algorithm 5.6: `isCenter`(v)

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

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 [37] 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.

5.3 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.

Number of rounds and running time

Our analysis follows those of [20] and [37]. The main idea is to track how fast the maximum degree decreases in the remaining graph at the end of each round.

Lemma 5.1. 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 5.1. *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 [83]), 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 5.2. *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 $\epsilon > 0$.

Approximation ratio

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

C4 is serializable and deterministic

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 5.3. C4 *achieves a 3-approximation ratio, in expectation.*

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 5.4. 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.

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 5.5. *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 5.6. *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*

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

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 5.6, 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):

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

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[3]

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

We are left to bound

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

To do that we use the following lemma.

Lemma 5.7. *If p_t satisfies*

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

then,

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

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,

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

□

Now, as with [37], we will simply have to bound the expectation of the bad triangles, adjacent to an edge (u, v):

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

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,

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

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

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

$$\begin{aligned}\Pr\{v \cap \mathcal{A} \neq \emptyset | \mathcal{S} \cap \mathcal{A} \neq \emptyset\} &= \frac{\Pr\{v \in \mathcal{A}\} \cdot \Pr\{\mathcal{S} \cap \mathcal{A} \neq \emptyset | v \in \mathcal{A}\}}{\Pr\{\mathcal{S} \cap \mathcal{A} \neq \emptyset\}} \\ &= \frac{\Pr\{v \in \mathcal{A}\}}{\Pr\{\mathcal{S} \cap \mathcal{A} \neq \emptyset\}} \\ &= \frac{\Pr\{v \in \mathcal{A}\}}{1 - \Pr\{\mathcal{S} \cap \mathcal{A} = \emptyset\}}.\end{aligned}$$

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

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

Hence, we obtain the following bound

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

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

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

which establishes our approximation ratio for ClusterWild!.

BSP Algorithms as a Proxy for Asynchronous Algorithms

Algorithm 5.7: C4 & ClusterWild! (asynchronous execution)

Input: G

clusterID$(1) = \ldots =$ clusterID$(n) = \infty$
$\pi =$ a random permutation of $\{1, \ldots, n\}$
while $V \neq \emptyset$ **do**
  $v =$ first element in V
  $V = V - \{v\}$
  if C4 **then** `attemptCluster`(v)
  else if ClusterWild! **then** `createCluster`(v)
  Remove clustered vertices from V and π

Output: $\{$clusterID$(1), \ldots,$ clusterID$(n)\}$.

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 Algorithm 5.7, where we remove the synchronization barriers.

The only difference between the asynchronous execution in Algorithm 5.7, compared to Algorithm 5.3, 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. Note that the asynchronous C4 is still serializable and deterministic and retains the 3-approximation ratio in expectation. In our experimental section we test the completely asynchronous algorithms against the BSP algorithms of the previous section, and observe that they perform quite similarly both in terms of accuracy of clustering, and running times.

5.4 Additional 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, 32]. 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 [47].

In the case of complete unweighted graphs, a long series of results establishes a 2.5 approximation via a rounded linear program (LP) [5]. A recent result establishes a 2.06 approximation using an elegant rounding to the same LP relaxation [35]. By avoiding the expensive LP, and by just using the rounding procedure of [5] 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, 128, 61], or maximizing the difference between the number of agreements and disagreements [34, 6], come with different hardness and approximation results. There are also several variants: chromatic CC [26], overlapping CC [25], or CC with small number of clusters and added constraints that are suitable for biology applications [114].

The way C4 finds the cluster centers can be seen as a variation of the MIS algorithm of [20]; 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 [24], 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.

5.5 Experiments

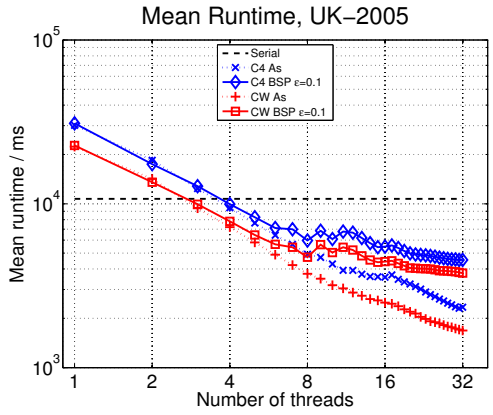
Our parallel algorithms were all implemented in Scala—we defer a full discussion of the implementation details to Appendix 5.C. We ran all our experiments on Amazon EC2's r3.8xlarge (32 vCPUs, 244Gb memory) instances, using 1-32 threads. The real graphs listed in

Graph	# vertices	# edges	Description
DBLP-2011	986,324	6,707,236	2011 DBLP co-authorship network [21, 22, 23].
ENWiki-2013	4,206,785	101,355,853	2013 link graph of English part of Wikipedia [21, 22, 23].
UK-2005	39,459,925	921,345,078	2005 crawl of the .uk domain [21, 22, 23].
IT-2004	41,291,594	1,135,718,909	2004 crawl of the .it domain [21, 22, 23].
WebBase-2001	118,142,155	1,019,903,190	2001 crawl by WebBase crawler [21, 22, 23].

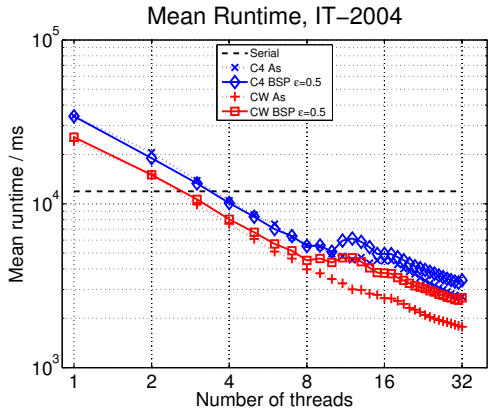
Table 5.1: Graphs used in the evaluation of our parallel algorithms.

Table 5.1 were each tested with 100 different random π 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 [37], which we denote as CDK for short[4]. Values of $\epsilon = 0.1, 0.5, 0.9$ were used for C4 BSP, ClusterWild! BSP and CDK. We present only representative plots of our results here; full results are given in our appendix.

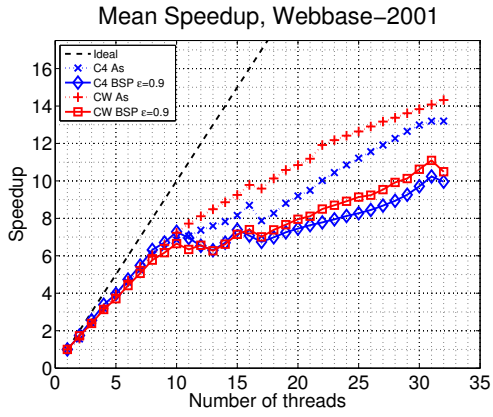
[4] CDK was only tested on the smaller graphs of DBLP-2011 and ENWiki-2013, because CDK was prohibitively slow, often 2-3 orders of magnitude slower than C4, ClusterWild!, and even serial Kwik-Cluster.



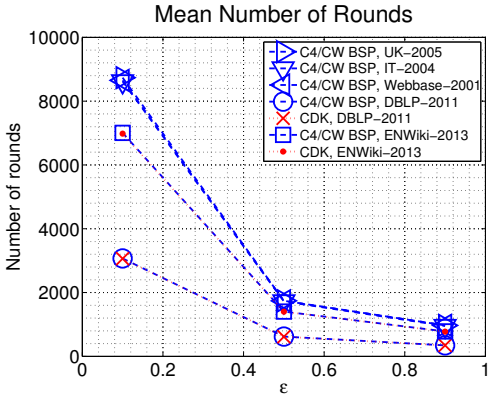
(a) Mean runtimes, UK-2005, $\epsilon = 0.1$



(b) Mean runtimes, IT-2004, $\epsilon = 0.5$



(c) Mean speedup, WebBase, $\epsilon = 0.9$

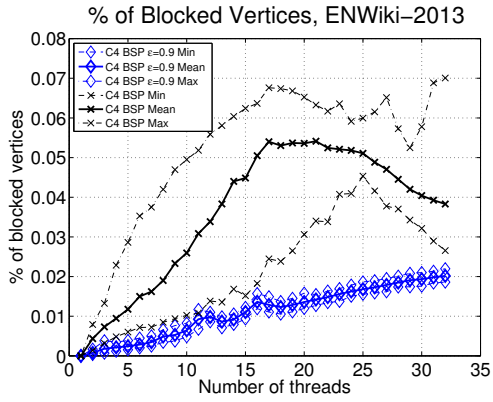


(d) Mean number of synchronization rounds for BSP algorithms

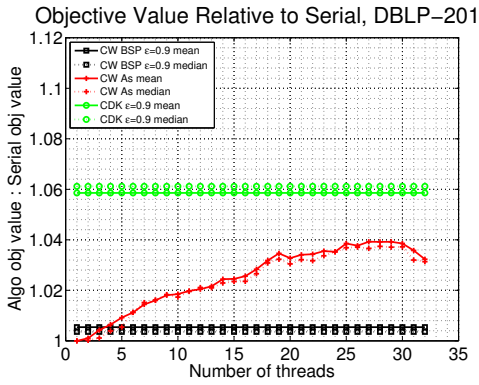
Figure 5.2: In the above figures, 'CW' is short for ClusterWild!, 'BSP' is short for the bulk-synchronous variants of the parallel algorithms, and 'As' is short for the asynchronous variants.

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.



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



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

Figure 5.3: 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.

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 $\epsilon = 0.9$.

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.

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.

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.

5.6 Discussions

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.

5.A Proofs of Theoretical Guarantees

Number of rounds for C4 and ClusterWild!

Lemma 5.1. 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

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

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[5]

$$\begin{aligned}
\frac{\binom{n-d_{i+j}}{P}}{\binom{n}{P}} &= \frac{(n-P)!}{(n-P-d_{i+j})!}\cdot\frac{(n-d_{i+j})!}{n!} \\
&= \frac{\prod_{t=1}^{d_{i+j}}(n-d_{i+1}+t-P)}{\prod_{t=1}^{d_{i+j}}(n-d_{i+1}+t)} = \prod_{t=1}^{d_{i+j}}\frac{n-d_{i+1}+t-P}{n-d_{i+1}+t} \\
&= \prod_{t=1}^{d_{i+j}}\left(1-\frac{P}{n-d_{i+1}+t}\right) \leq \left(1-\frac{P}{n}\right)^{d_{i+j}} \\
&\leq \left(1-\frac{\epsilon}{\Delta_i}\right)^{\Delta_i/2} = \left[\left(1-\frac{\epsilon}{\Delta_i}\right)^{\Delta_i/\epsilon}\right]^{\epsilon/2} \leq e^{-\epsilon/2}.
\end{aligned}$$

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 [20]. 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. □

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

Running times

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

Theorem 5.8 (Theorem 1 in [83]). *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 5.8, we first need to convert it into a result for sampling without replacement (instead of i.i.d. sampling).

Lemma 5.9. *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 $\rho_X = \Pr\left(f(X_1,\ldots,X_n) > C\right)$ 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

$$\begin{aligned}\rho_X &= \sum_{b=0}^{n} \Pr\left(f(X_1,\ldots,X_n) > C \,\middle|\, \sum_{i=1}^{n} X_i = b\right)\cdot \Pr\left(\sum_{i=1}^{n} X_i = b\right)\\ &= \sum_{b=0}^{n} q_b \cdot \Pr\left(\sum_{i=1}^{n} X_i = b\right) \qquad (5.1)\end{aligned}$$

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

$$\begin{aligned}\rho_Y &= \sum_{b=0}^{n} \Pr\left(f(Y_1,\ldots,Y_n) > C \,\middle|\, \sum_{i=1}^{n} Y_i = b\right)\cdot \Pr\left(\sum_{i=1}^{n} Y_i = b\right)\\ &\overset{(i)}{=} \sum_{b=0}^{n} q_b \cdot \Pr\left(\sum_{i=1}^{n} Y_i = b\right)\\ &\overset{(ii)}{=} q_B \cdot 1 \qquad (5.2)\end{aligned}$$

where (i) comes form the fact that $\Pr\left(f(Y_1,\ldots,Y_n) > C \,|\sum_{i=1}^{n} Y_i = b\right)$ 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

$$\rho_X = \sum_{b=0}^{n} q_b \cdot \Pr\left(\sum_{i=1}^{n} X_i = b\right) \geq q_B \cdot \Pr\left(\sum_{i=1}^{n} X_i = B\right) = \rho_Y \cdot \Pr\left(\sum_{i=1}^{n} X_i = B\right) \quad (5.3)$$

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 $\Pr\left(\sum_i X_i = B\right)$ 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

$$\Pr\left(\sum_{i=1}^{n} X_i = B\right) \geq \frac{1}{n}. \quad (5.4)$$

If we combine (5.3) and (5.4) we get $\rho_X \geq \rho_Y/n \Leftrightarrow \rho_Y \leq n \cdot \delta$. □

Corollary 5.10. *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 5.2. *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 5.10, 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 5.10, 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 5.1, we obtain the theorem.

□

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 5.5. *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 , hence both u,w belong to (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 , with one end point say $u' \in$ and one $w' \in V\backslash v$, 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 $\mathcal{C}_{\text{CW}}$. Then, consider the graph G' that is generated by deleting all edges between $\mathcal{C}_{\text{CW}}$. 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 5.11. *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. □

The above proposition is used to establish the τ_{new} bound for Lemma 5.6. 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 5.6 then follows.

Lemma 5.6. *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*

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

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.

5.B Implementation Details

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

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 monotonicity 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.

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 5.1, 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.

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.

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.

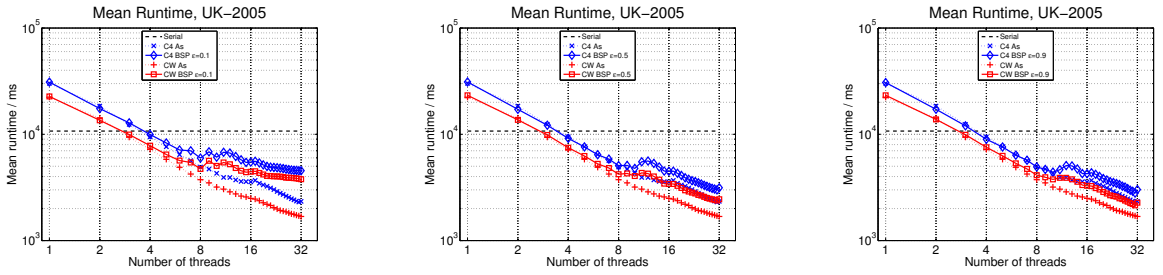
Comment on CDK Implementation

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.

5.C Complete Experiment Results

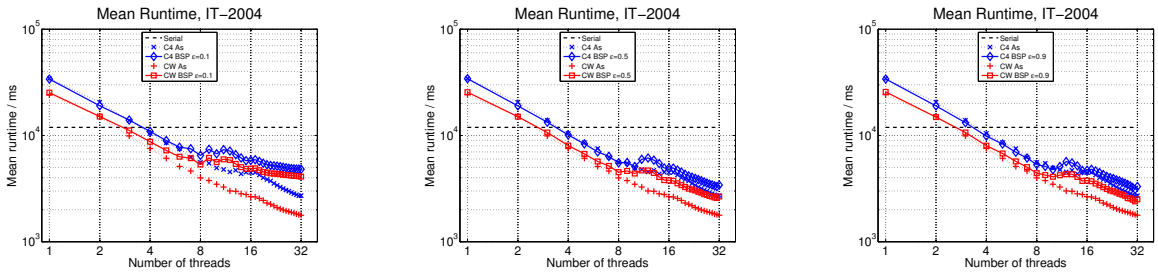
Empirical mean runtimes.

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.



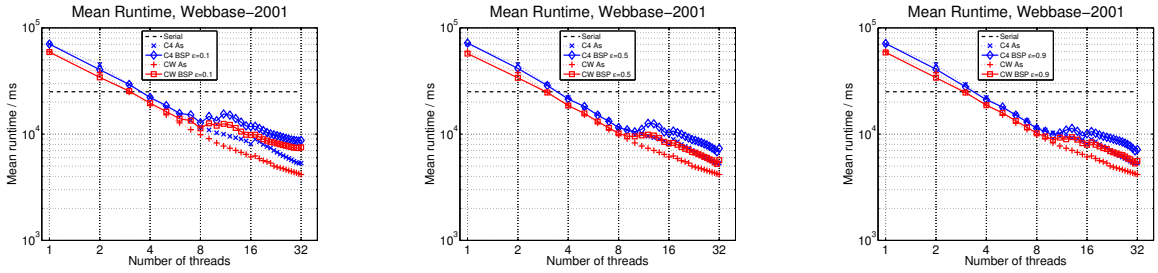
(a) UK-2005, $\epsilon = 0.1$ (b) UK-2005, $\epsilon = 0.5$ (c) UK-2005, $\epsilon = 0.9$

Figure 5.4: Empirical mean runtimes for UK-2005.



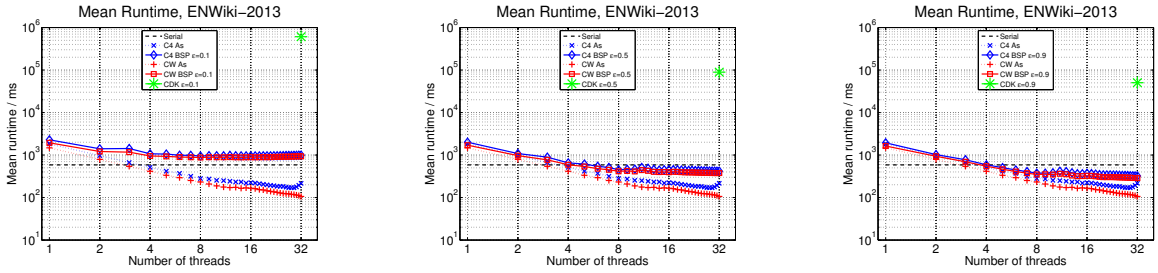
(a) IT-2004, $\epsilon = 0.1$ (b) IT-2004, $\epsilon = 0.5$ (c) IT-2004, $\epsilon = 0.9$

Figure 5.5: Empirical mean runtimes for IT-2004.



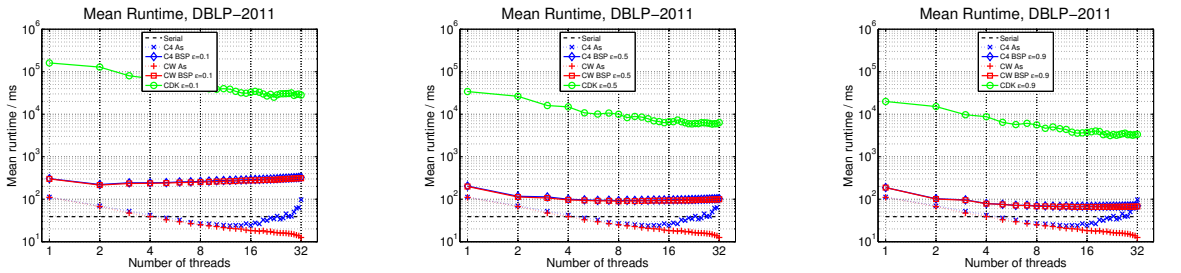
(a) Webbase-2001, $\epsilon = 0.1$ (b) Webbase-2001, $\epsilon = 0.5$ (c) Webbase-2001, $\epsilon = 0.9$

Figure 5.6: Empirical mean runtimes for Webbase-2001.



(a) ENWiki-2013, $\epsilon = 0.1$ (b) ENWiki-2013, $\epsilon = 0.5$ (c) ENWiki-2013, $\epsilon = 0.9$

Figure 5.7: Empirical mean runtimes for ENWiki-2013.

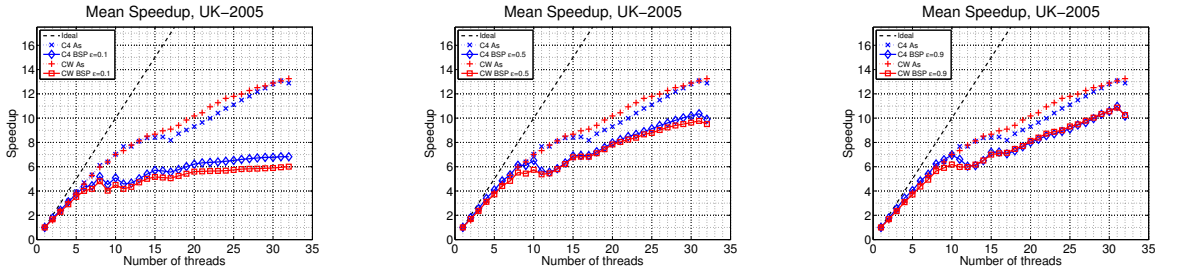


(a) DBLP-2011, $\epsilon = 0.1$ (b) DBLP-2011, $\epsilon = 0.5$ (c) DBLP-2011, $\epsilon = 0.9$

Figure 5.8: Empirical mean runtimes for DBLP-2011.

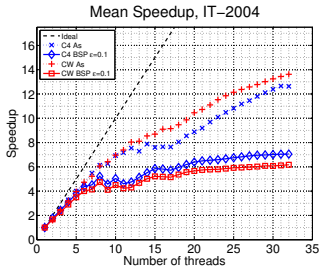
Empirical mean speedups.

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.

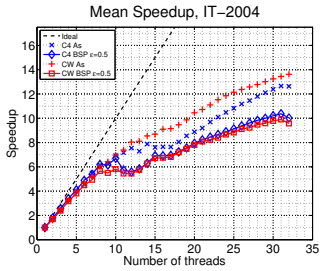


(a) UK-2005, $\epsilon = 0.1$ (b) UK-2005, $\epsilon = 0.5$ (c) UK-2005, $\epsilon = 0.9$

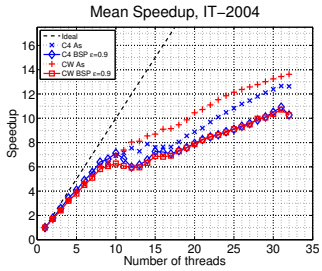
Figure 5.9: Empirical mean speedups for UK-2005.



(a) IT-2004, $\epsilon = 0.1$

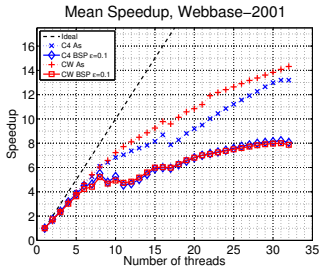


(b) IT-2004, $\epsilon = 0.5$

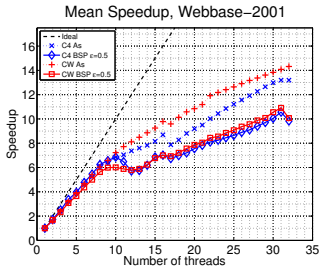


(c) IT-2004, $\epsilon = 0.9$

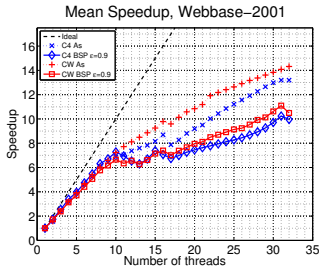
Figure 5.10: Empirical mean speedups for IT-2004.



(a) Webbase-2001, $\epsilon = 0.1$

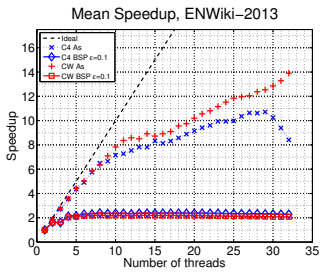


(b) Webbase-2001, $\epsilon = 0.5$

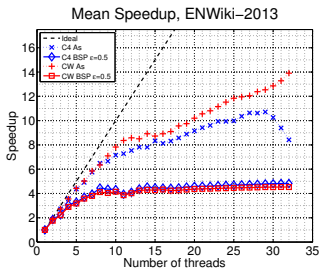


(c) Webbase-2001, $\epsilon = 0.9$

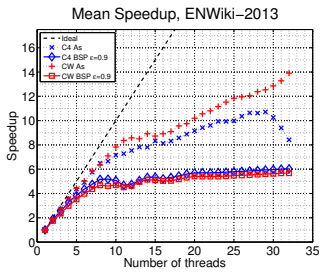
Figure 5.11: Empirical mean speedups for Webbase-2001.



(a) ENWiki-2013, $\epsilon = 0.1$

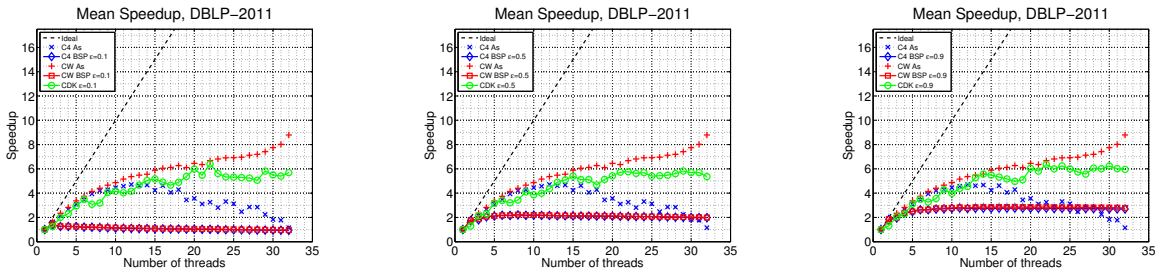


(b) ENWiki-2013, $\epsilon = 0.5$



(c) ENWiki-2013, $\epsilon = 0.9$

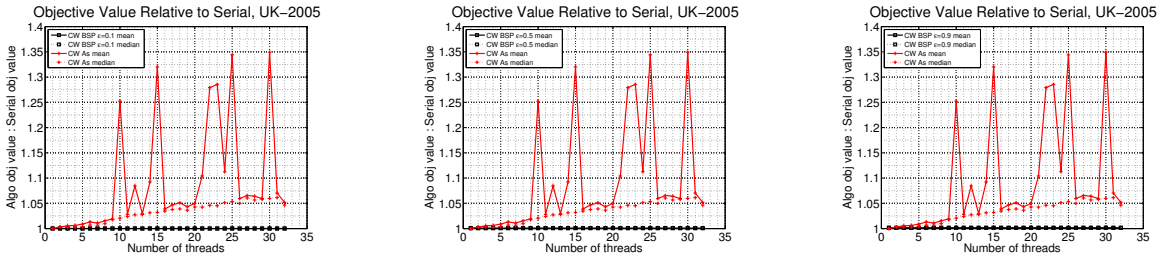
Figure 5.12: Empirical mean speedups for ENWiki-2013.



(a) DBLP-2011, $\epsilon = 0.1$ (b) DBLP-2011, $\epsilon = 0.5$ (c) DBLP-2011, $\epsilon = 0.9$

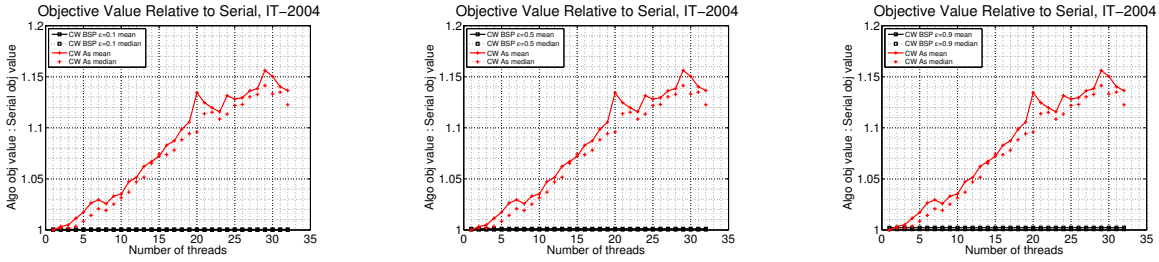
Figure 5.13: Empirical mean speedups for DBLP-2011.

Empirical objective values relative to mean objective value obtained by serial algorithm.



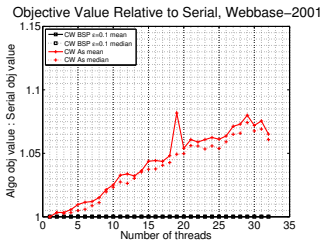
(a) UK-2005, $\epsilon = 0.1$ (b) UK-2005, $\epsilon = 0.5$ (c) UK-2005, $\epsilon = 0.9$

Figure 5.14: Empirical objective values relative to mean objective value obtained by serial algorithm, for UK-2005.

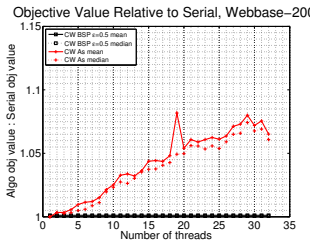


(a) IT-2004, $\epsilon = 0.1$ (b) IT-2004, $\epsilon = 0.5$ (c) IT-2004, $\epsilon = 0.9$

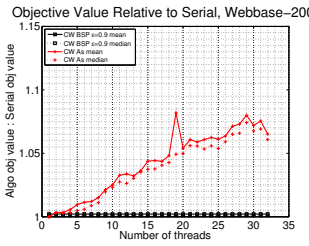
Figure 5.15: Empirical objective values relative to mean objective value obtained by serial algorithm, for IT-2004.



(a) Webbase-2001, $\epsilon = 0.1$

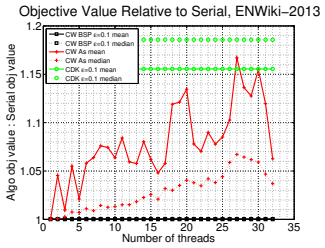


(b) Webbase-2001, $\epsilon = 0.5$

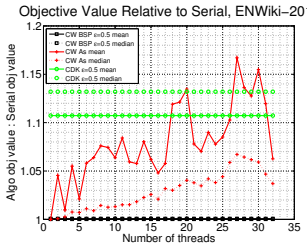


(c) Webbase-2001, $\epsilon = 0.9$

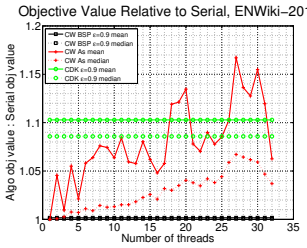
Figure 5.16: Empirical objective values relative to mean objective value obtained by serial algorithm, for Webbase-2001.



(a) ENWiki-2013, $\epsilon = 0.1$

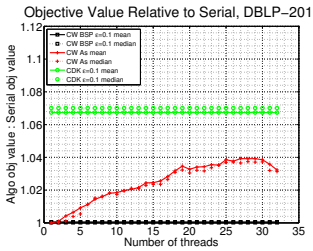


(b) ENWiki-2013, $\epsilon = 0.5$

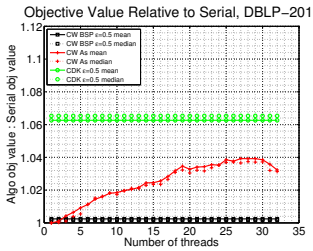


(c) ENWiki-2013, $\epsilon = 0.9$

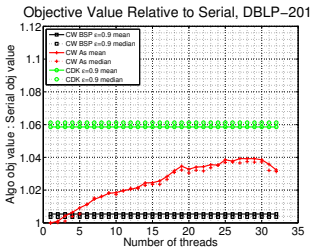
Figure 5.17: Empirical objective values relative to mean objective value obtained by serial algorithm, for ENWiki-2013.



(a) DBLP-2011, $\epsilon = 0.1$



(b) DBLP-2011, $\epsilon = 0.5$



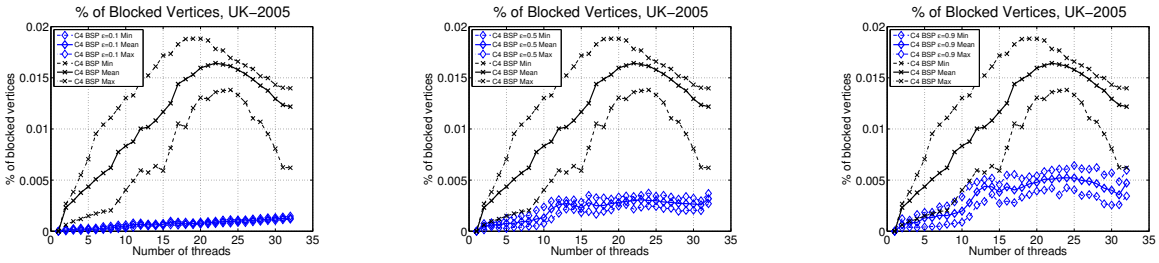
(c) DBLP-2011, $\epsilon = 0.9$

Figure 5.18: Empirical objective values relative to mean objective value obtained by serial algorithm, for DBLP-2011.

Empirical percentage of blocked vertices.

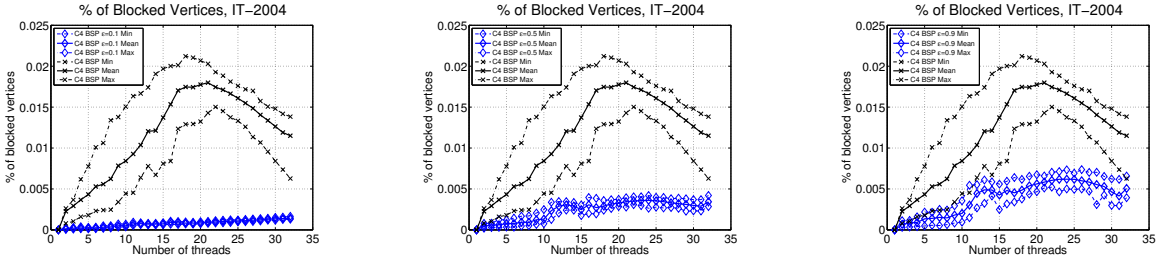
Generally the number of blocked vertices increases with the number of threads and larger ϵ values. C4 BSP has fewer blocked vertices than asynchronous C4, but at the cost of more synchronization barriers. We point out that across all 100 runs of every graphs, the maximum

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.



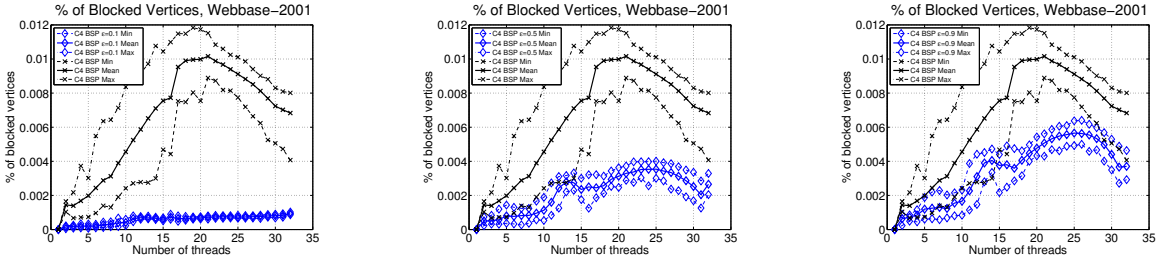
(a) UK-2005, $\epsilon = 0.1$ (b) UK-2005, $\epsilon = 0.5$ (c) UK-2005, $\epsilon = 0.9$

Figure 5.19: Empirical percentage of blocked vertices for UK-2005.



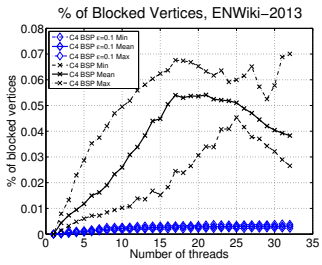
(a) IT-2004, $\epsilon = 0.1$ (b) IT-2004, $\epsilon = 0.5$ (c) IT-2004, $\epsilon = 0.9$

Figure 5.20: Empirical percentage of blocked vertices for IT-2004.

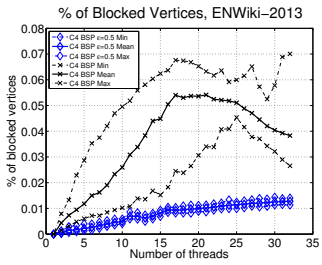


(a) Webbase-2001, $\epsilon = 0.1$ (b) Webbase-2001, $\epsilon = 0.5$ (c) Webbase-2001, $\epsilon = 0.9$

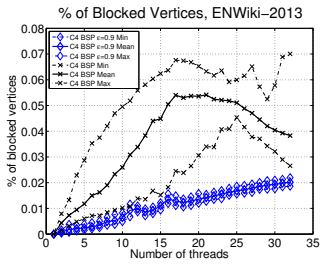
Figure 5.21: Empirical percentage of blocked vertices for Webbase-2001.



(a) ENWiki-2013, $\epsilon = 0.1$

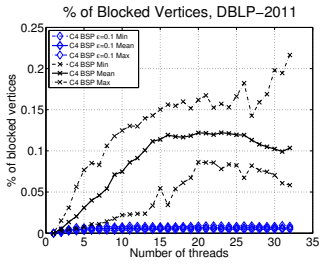


(b) ENWiki-2013, $\epsilon = 0.5$

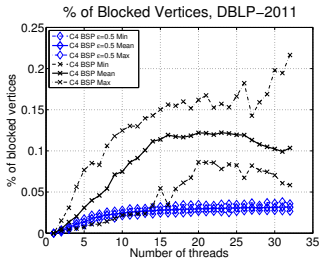


(c) ENWiki-2013, $\epsilon = 0.9$

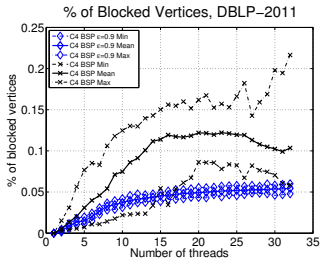
Figure 5.22: Empirical percentage of blocked vertices for ENWiki-2013.



(a) DBLP-2011, $\epsilon = 0.1$



(b) DBLP-2011, $\epsilon = 0.5$



(c) DBLP-2011, $\epsilon = 0.9$

Figure 5.23: Empirical percentage of blocked vertices for DBLP-2011.

Chapter 6

Non-monotone Submodular Maximization

6.1 Introduction

Many important problems including sensor placement [79], image co-segmentation [77], MAP inference for determinantal point processes [60], influence maximization in social networks [76], and document summarization [91] may be expressed as the maximization of a submodular function. The submodular formulation enables the use of targeted algorithms [30, 105] that offer theoretical worst-case guarantees on the quality of the solution. For several maximization problems of *monotone* submodular functions (satisfying $F(A) \leq F(B)$ for all $A \subseteq B$), a simple greedy algorithm [105] achieves the optimal approximation factor of $1 - \frac{1}{e}$. The optimal result for the wider, important class of *non-monotone* functions — an approximation guarantee of $1/2$ — is much more recent, and achieved by a *double greedy* algorithm by [30].

While theoretically optimal, in practice these algorithms do not scale to large real world problems, since the inherently serial nature of the algorithms poses a challenge to leveraging advances in parallel hardware. This limitation raises the question of parallel algorithms for submodular maximization that ideally preserve the theoretical bounds, or weaken them gracefully, in a quantifiable manner.

In this chapter[1], we focus on the double greedy algorithm from the perspective of parallel transaction processing systems. Specifically, we apply concurrency control to coordinate transactions, thereby providing upper and lower bounds on the transactions' read sets that are exploited to allow most transactions to proceed in parallel. Key to this exploitation is our insight that there is a low probability that concurrent transactions will affect one another's decisions. The resultant algorithm, CC-2G, is serializable and retains the optimality of the double greedy algorithm at the expense of increased coordination.

We also develop a coordination free CF-2G algorithm, and show that a natural weaker bound of the double greedy algorithm translates to a poorer approximation ratio.

[1] Work done as part of [111].

The primary contributions of this chapter are:

1. We propose two parallel algorithms for unconstrained non-monotone submodular maximization, which trade off parallelism and tight approximation guarantees.
2. We provide approximation guarantees for CF-2G and analytically bound the expected loss in objective value for set-cover with costs and max-cut as running examples.
3. We prove that CC-2G preserves the optimality of the serial double greedy algorithm and analytically bound the additional coordination overhead for covering with costs and max-cut.
4. We demonstrate empirically using two synthetic and four real datasets that our parallel algorithms perform well in terms of both speed and objective values.

The rest of the chapter is organized as follows. Section 6.2 discusses the problem of submodular maximization and introduces the double greedy algorithm. Section 6.3 provides background on concurrency control mechanisms. We describe and provide intuition for our CF-2G and CC-2G algorithms in Section 6.4 and Section 6.5, and then analyze the algorithms both theoretically (Section 6.6) and empirically (Section 6.7).

6.2 Submodular Maximization

A set function $F : 2^V \to \mathbb{R}$ defined over subsets of a ground set V is *submodular* if it satisfies *diminishing marginal returns*: for all $A \subseteq B \subseteq V$ and $e \notin B$, it holds that $F(A \cup \{e\}) - F(A) \geq F(B \cup \{e\}) - F(B)$. Throughout this chapter, we will assume that F is nonnegative and $F(\emptyset) = 0$. Submodular functions have emerged in areas such as game theory [123], graph theory [57], combinatorial optimization [122], and machine learning [81, 19]. Casting machine learning problems as submodular optimization enables the use of algorithms for submodular maximization [30, 105] that offer theoretical worst-case guarantees on the quality of the solution.

While those algorithms confer strong guarantees, their design is inherently serial, limiting their usability in large-scale problems. Recent work has addressed faster [11] and parallel [104, 85, 134] versions of the greedy algorithm by [105] for maximizing *monotone* submodular functions that satisfy $F(A) \leq F(B)$ for any $A \subseteq B \subseteq V$. However, many important applications in machine learning lead to *non-monotone* submodular functions. For example, graphical model inference [60, 119], or trading off any submodular gain maximization with costs (functions of the form $F(S) = G(S) - \lambda M(S)$, where $G(S)$ is monotone submodular and $M(S)$ a linear (modular) cost function), such as for utility-privacy tradeoffs [80], require maximizing non-monotone submodular functions. For non-monotone functions, the simple greedy algorithm in [105] can perform arbitrarily poorly (see Appendix 6.E for an example). Intuitively, the introduction of additional elements with monotone submodular functions never decreases the objective while introducing elements with non-monotone submodular

functions can *decrease* the objective to its minimum. For non-monotone functions, [30] recently proposed an optimal double greedy algorithm that works well in a serial setting. In this chapter, we study parallelizations of this algorithm.

The serial double greedy algorithm. The serial double greedy algorithm of [30] (SER-2G, in Algorithm 6.3) maintains two sets $A^i \subseteq B^i$. Initially, $A^0 = \emptyset$ and $B^0 = V$. In iteration i, the set A^{i-1} contains the items selected before item/iteration i, and B^{i-1} contains A^i and the items that are so far undecided. The algorithm serially passes through the items in V and determines online whether to keep item i (add to A^i) or discard it (remove from B^i), based on a threshold that trades off the gain $\Delta_+(i) = F(A^{i-1} \cup i) - F(A^{i-1})$ of adding i to the currently selected set A^{i-1}, and the gain $\Delta_-(i) = F(B^{i-1} \setminus i) - F(B^{i-1})$ of removing i from the candidate set, estimating its complementarity to other remaining elements. For any element ordering, this algorithm achieves a tight 1/2-approximation in expectation.

6.3 Concurrency Control with Coordinated Bounds

In this chapter we adopt a transactional view of the program state and explore parallelization strategies through the lens of parallel transaction processing systems. We recast the program state (the sets A and B) as data, and the operations (adding elements to A and removing elements from B) as transactions. More precisely we reformulate the double greedy algorithm (Algorithm 6.3) as a series of *exchangeable*, *Read-Write* transactions of the form:

$$\Lambda_e(A,B) := \begin{cases} (\{e\},\emptyset) & \text{if } u_e \leq \frac{[\Delta_+(A,e)]_+}{[\Delta_+(A,e)]_+ + [\Delta_-(B,e)]_+} \\ (\emptyset,\{e\}) & \text{otherwise.} \end{cases} \tag{6.1}$$

$$T((A,B),\lambda) := (A \cup \lambda_A, B - \lambda_B) \tag{6.2}$$

$$\mathcal{T}_e(A,B) := T((A,B),\lambda_e(A,B)) \tag{6.3}$$

The transaction $\mathcal{T}_e(A,B) := T(A,B,\Lambda_e(A,B))$ is a function from the sets A and B to new sets A and B based on the element $e \in V$ and the predetermined random bits u_e for that element.

By composing the transactions $\mathcal{T}_n(\mathcal{T}_{n-1}(\ldots\mathcal{T}_1(\emptyset,V)))$ we recover the serial double-greedy algorithm defined in Algorithm 6.3. In fact, any ordering of the *serial* composition of the transactions recovers a permuted execution of Algorithm 6.3 and therefore the optimal approximation algorithm. However, this raises the question: *is it possible to apply transactions in parallel?* If we execute transactions T_i and T_j, with $i \neq j$, in parallel we need a method to merge the resulting program states. In the context of the double greedy algorithm, we could define the parallel, coordination-free, execution of two transactions as:

$$\begin{aligned} \mathcal{T}_i(A,B) + \mathcal{T}_j(A,B) &:= T(T((A,B),\Lambda_i(A,B)),\Lambda_j(A,B)) \\ &= T((A,B),(\Lambda_i(A,B)_A \cup \Lambda_j(A,B)_A, \Lambda_i(A,B)_B \cup \Lambda_j(A,B)_B)), \end{aligned} \tag{6.4}$$

the union of the resulting A and the intersection of the resulting B. While we can easily generalize (6.4) to many parallel transactions, we cannot always guarantee that the result will be serializable, i.e., that it corresponds to a serial composition of transactions. As a consequence, we cannot directly apply the analysis of Buchbinder et al. [30] to derive strong approximation guarantees for the parallel execution.

In this chapter we adopt a coordinated bounds approach to parallel transaction processing in which parallel transactions are constructed under bounds on the possible program state. If the transaction could violate the bound then it is processed serially on the server. This approach achieves a high degree of parallelism when the cost of constructing the transaction dominates the cost of applying the transaction. By adjusting the definition of the bound we can span a space of coordination-free to serializable executions.

Algorithm 6.1: Generalized transactions with coordinated bounds

```
1 for p ∈ {1, ..., P} do in parallel
2     while ∃ element to process do
3         e = next element to process
4         (𝔤_e, i) = requestGuarantee(e)
5         ∂_i = propose(e, 𝔤_e)
6         commit(e, i, ∂_i) // Non-blocking
```

Algorithm 6.2: Commit transaction i

```
1 wait until ∀j < i, processed(j) = true
2 Atomically
3     if ∂_i = FAIL then
          // Deferred proposal
4         ∂_i = propose(e, 𝔖)
      // Advance the program state
5     𝔖 ← ∂_i(𝔖)
```

Figure 6.1: Algorithm for generalized transactions. Each transaction requests its position i in the commit ordering, as well as the bounds $\mathfrak{g}_e$ that are guaranteed to hold when it commits. Transactions are also guaranteed to be committed according to the given ordering.

In Figure 6.1 we describe the coordinated bounds transaction pattern. The clients (Algorithm 6.1), in parallel, construct and commit transactions under bounded assumptions about the program state $\mathfrak{S}$ (*i.e.,* the sets A and B). Transactions are constructed by requesting the latest bound $\mathfrak{g}_e$ on $\mathfrak{S}$ at logical time i and computing a change ∂_i to $\mathfrak{S}$ (*e.g.,* Add e to A). If the bound is insufficient to construct the transaction then $\partial_i = \text{FAIL}$ is returned. The client then sends the proposed change ∂_i to the server to be committed atomically and proceeds to the next element without waiting for a response.

The server (Algorithm 6.2) *serially* applies the transactions advancing the program state (*i.e.,* adding elements to A or removing elements from B). If the bounds were insufficient and the transaction failed at the client (*i.e.,* $\partial_i = \text{FAIL}$) then the server *serially* reconstructs and applies the transaction under the true program state. Moreover, the server is responsible for deriving bounds, processing transactions in the logical order i, and producing the serializable output $\partial_n(\partial_{n-1}(\ldots\partial_1(\mathfrak{S})))$.

In the case of submodular maximization, the cost of constructing the transaction depends on evaluating the marginal gains with respect to changes in A and B while the cost of applying the transaction reduces to setting a bit. Thus, distributing the work of proposals over multiple threads allows database systems to achieve parallelism, even with the serial

commit process on the server. It is also essential that only a few transactions fail at the client. Indeed, the analysis of these systems focuses on ensuring that the majority of the transactions succeed.

6.4 CF-2G: Coordination-Free Double Greedy Algorithm

The coordination-free approach attempts to reduce the need to coordinate guarantees and the logical ordering. This is achieved by operating on potentially stale states: the transaction guarantee reduces to requiring $\mathfrak{g}_e$ be a stale version of $\mathfrak{S}$, and the logical ordering is implicitly defined by the time of commit. In using these weak guarantees, CF-2G is overly optimistically assuming that concurrent transactions are independent, which could potentially lead to erroneous decisions.

Algorithm 6.3: SER-2G: serial double greedy

1 $A^0 = \emptyset$, $B^0 = V$
2 **for** $i = 1$ *to* n **do**
3 $\quad \Delta_+(i) = F(A^{i-1} \cup i) - F(A^{i-1})$
4 $\quad \Delta_-(i) = F(B^{i-1} \backslash i) - F(B^{i-1})$
5 $\quad$ Draw $u_i \sim Unif(0,1)$
6 $\quad$ **if** $u_i < \frac{[\Delta_+(i)]_+}{[\Delta_+(i)]_+ + [\Delta_-(i)]_+}$ **then**
7 $\quad\quad A^i := A^{i-1} \cup i$
8 $\quad\quad B^i := B^{i-1}$
9 $\quad$ **else**
10 $\quad\quad A^i := A^{i-1}$
11 $\quad\quad B^i := B^{i-1} \backslash i$

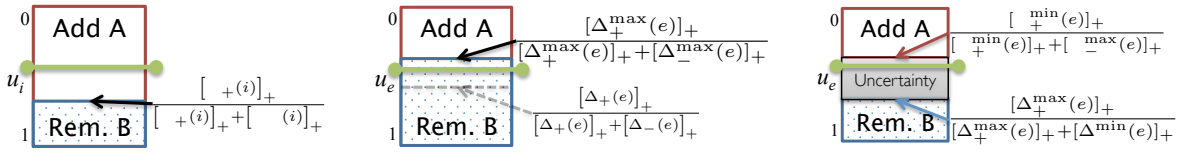
Algorithm 6.4: CF-2G: coord-free double greedy

1 $\widehat{A} = \emptyset$, $\widehat{B} = V$
2 **for** $p \in \{1, \ldots, P\}$ **do in parallel**
3 $\quad$ **while** $\exists$ *element to process* **do**
4 $\quad\quad e =$ next element to process
5 $\quad\quad \widehat{A}_e = \widehat{A}$; $\widehat{B}_e = \widehat{B}$
6 $\quad\quad \Delta_+^{\max}(e) = F(\widehat{A}_e \cup e) - F(\widehat{A}_e)$
7 $\quad\quad \Delta_-^{\max}(e) = F(\widehat{B}_e \backslash e) - F(\widehat{B}_e)$
8 $\quad\quad$ Draw $u_e \sim Unif(0,1)$
9 $\quad\quad$ **if** $u_e < \frac{[\Delta_+^{\max}(e)]_+}{[\Delta_+^{\max}(e)]_+ + [\Delta_-^{\max}(e)]_+}$ **then**
10 $\quad\quad\quad \widehat{A}(e) \leftarrow 1$
11 $\quad\quad$ **else** $\widehat{B}(e) \leftarrow 0$

Algorithm 6.4 is the coordination-free parallel double greedy algorithm.[2] CF-2G closely resembles the serial SER-2G, but the elements $e \in V$ are no longer processed in a fixed order. Thus, the sets A, B are replaced by potentially stale local estimates (bounds) $\widehat{A}, \widehat{B}$, where $\widehat{A}$ is a subset of the true A and $\widehat{B}$ is a superset of the actual B on each iteration (see Lemma 6.1). These bounding sets allow us to compute bounds $\Delta_+^{\max}, \Delta_-^{\max}$ which approximate Δ_+, Δ_- from the serial algorithm. We now formalize this idea.

To analyze the CF-2G algorithm we order the elements $e \in V$ according to the commit time (*i.e.,* when Algorithm 6.4 line 8 is executed). Let $\iota(e)$ be the position of e in this total ordering on elements. This ordering allows us to define monotonically non-decreasing sets $A^i =$

[2] We present only the parallelized probabilistic versions of [30]. Both parallel algorithms can be easily extended to the deterministic version of [30]; CF-2G can also be extended to the multilinear version of [30].



(a) SER-2G (b) CF-2G (c) CC-2G

Figure 6.2: Illustration of algorithms. **(a)** SER-2G computes a threshold based on the true values Δ_+, Δ_-, and chooses an action based by comparing a uniform random u_i against the threshold. **(b)** CF-2G approximates the threshold based on stale $\widehat{A}$, $\widehat{B}$, possibly choosing the wrong action. **(c)** CC-2G computes two thresholds based on the bounds on A, B, which defines an uncertainty region where it is not possible to choose the correct action locally. If the random value u_e falls inside the uncertainty interval than the transaction FAILS and must be recomputed serially by the server; otherwise the transaction holds under all possible global states.

$\{e' : e' \in A, \iota(e') < i\}$ where A is the final returned set, and monotonically non-increasing sets $B^i = A^i \cup \{e' : \iota(e') \geq i\}$. The sets A^i, B^i provide a serialization against which we can compare CF-2G; in this serialization, Algorithm 6.3 computes $\Delta_+(e) = F(A^{\iota(e)-1} \cup e) - F(A^{\iota(e)-1})$ and $\Delta_-(e) = F(B^{\iota(e)-1} \backslash e) - F(B^{\iota(e)-1})$. On the other hand, CF-2G uses stale versions[3] $\widehat{A}_e$, $\widehat{B}_e$: Algorithm 6.4 computes $\Delta_+^{\max}(e) = F(\widehat{A}_e \cup e) - F(\widehat{A}_e)$ and $\Delta_-^{\max}(e) = F(\widehat{B}_e \backslash e) - F(\widehat{B}_e)$.

The next lemma shows that $\widehat{A}_e, \widehat{B}_e$ are bounding sets for the serialization's sets $A^{\iota(e)-1}, B^{\iota(e)-1}$. Intuitively, the bounds hold because $\widehat{A}_e$, $\widehat{B}_e$ are stale versions of $A^{\iota(e)-1}$, $B^{\iota(e)-1}$, which are monotonically non-decreasing and non-increasing sets. Appendix 6.A gives a detailed proof.

Lemma 6.1. *In* CF-2G*, for any* $e \in V$*,* $\widehat{A}_e \subseteq A^{\iota(e)-1}$*, and* $\widehat{B}_e \supseteq B^{\iota(e)-1}$*.*

Corollary 6.2. *Submodularity of* F *implies for* CF-2G $\Delta_+(e) \leq \Delta_+^{\max}(e)$*, and* $\Delta_-(e) \leq \Delta_-^{\max}(e)$*.*

The error in CF-2G depends on the tightness of the bounds in Corollary 6.2. We analyze this in Section 6.6.

For some functions F, we can maintain sketches or statistics to aid the computation of $\Delta_+^{\max}$, $\Delta_-^{\max}$, and still obtain the bounds given in Corollary 6.2. In Appendix 17, we consider functions of *separable sums*, which are useful for applications such as document summarization [91]. Specifically we consider functions of the form $F(X) = \sum_{l=1}^{L} g\left(\sum_{i \in X \cup S_l} w_l(i)\right) - \lambda \sum_{i \in X} v(i)$, where $S_l \subseteq V$ are (possibly overlapping) groups of elements in the ground set, g is a non-decreasing concave scalar function, and $w_l(i)$ and $v(i)$ are non-negative scalar weights.

[3] For clarity, we present the algorithm as creating a copy of $\widehat{A}$, $\widehat{B}$, $\widetilde{A}$, and $\widetilde{B}$ for each element. In practice, it is more efficient to update and access them in shared memory. Nevertheless, our theorems hold for both settings.

6.5 CC-2G: Concurrency Control for the Double Greedy Algorithm

Algorithm 6.5: CC-2G: concurrency control double greedy

$\widehat{A} = \widetilde{A} = \emptyset$, $\widehat{B} = \widetilde{B} = V$
for $i = 1, \ldots, |V|$ **do** processed$(i) = false$
$\iota = 0$
for $p \in \{1, \ldots, P\}$ **do in parallel**
  while $\exists$ *element to process* **do**
    $(e, \widehat{A}_e, \widetilde{A}_e, \widehat{B}_e, \widetilde{B}_e, i)$ = getGuarantee()
    (result, u_e) = propose(e, $\widehat{A}_e$, $\widetilde{A}_e$, $\widehat{B}_e$, $\widetilde{B}_e$)
    commit(e, i, u_e, result)

Algorithm 6.6: CC-2G getGuarantee()

e = next element to process
$\widetilde{A}(e) \leftarrow 1$; $\widetilde{B}(e) \leftarrow 0$
do atomically
  $i = \iota$
  $\iota \leftarrow \iota + 1$
$\widehat{A}_e = \widehat{A}$; $\widehat{B}_e = \widehat{B}$
$\widetilde{A}_e = \widetilde{A}$; $\widetilde{B}_e = \widetilde{B}$
return $(e, \widehat{A}_e, \widetilde{A}_e, \widehat{B}_e, \widetilde{B}_e, i)$

Algorithm 6.7: CC-2G propose

$\Delta_+^{\min}(e) = F(\widetilde{A}_e) - F(\widetilde{A}_e \backslash e)$
$\Delta_+^{\max}(e) = F(\widehat{A}_e \cup e) - F(\widehat{A}_e)$
$\Delta_-^{\min}(e) = F(\widetilde{B}_e) - F(\widetilde{B}_e \cup e)$
$\Delta_-^{\max}(e) = F(\widehat{B}_e \backslash e) - F(\widehat{B}_e)$
Draw $u_e \sim Unif(0, 1)$
if $u_e < \frac{[\Delta_+^{\min}(e)]_+}{[\Delta_+^{\min}(e)]_+ + [\Delta_-^{\max}(e)]_+}$ **then**
  result $\leftarrow 1$
else if $u_e > \frac{[\Delta_+^{\max}(e)]_+}{[\Delta_+^{\max}(e)]_+ + [\Delta_-^{\min}(e)]_+}$ **then**
  result $\leftarrow -1$
else result $\leftarrow$ FAIL
return (result, u_e)

Algorithm 6.8: CC-2G getGuarantee(), deterministic

do atomically
  $i = \iota$
  $e = \pi(i)$
  $\widetilde{A}(e) \leftarrow 1$; $\widetilde{B}(e) \leftarrow 0$
  $\iota \leftarrow \iota + 1$
$\widehat{A}_e = \widehat{A}$; $\widehat{B}_e = \widehat{B}$
$\widetilde{A}_e = \widetilde{A}$; $\widetilde{B}_e = \widetilde{B}$
return $(e, \widehat{A}_e, \widetilde{A}_e, \widehat{B}_e, \widetilde{B}_e, i)$

Algorithm 6.9: CC-2G: commit(e, i, u_e, result)

wait until $\forall j < i$, processed$(j) = true$
if *result* $= FAIL$ **then**
  $\Delta_+^{\text{exact}}(e) = F(\widehat{A} \cup e) - F(\widehat{A})$
  $\Delta_-^{\text{exact}}(e) = F(\widehat{B} \backslash e) - F(\widehat{B})$
  if $u_e < \frac{[\Delta_+^{exact}(e)]_+}{[\Delta_+^{exact}(e)]_+ + [\Delta_-^{exact}(e)]_+}$ **then**
    result $\leftarrow 1$
  else result $\leftarrow -1$
if *result* $= 1$ **then** $\widehat{A}(e) \leftarrow 1$; $\widetilde{B}(e) \leftarrow 1$
else $\widetilde{A}(e) \leftarrow 0$; $\widehat{B}(e) \leftarrow 0$
processed$(i) = true$

The concurrency control-based double greedy algorithm[2], CC-2G, is presented in Algorithm 6.5, and closely follows the meta-algorithm of Algorithm 6.1 and Algorithm 6.2. Unlike

in CF-2G, the concurrency control mechanisms of CC-2G ensure that concurrent transactions are serialized when they are not independent.

Serializability is achieved by maintaining sets $\widehat{A}$, $\widetilde{A}$, $\widehat{B}$, $\widetilde{B}$, which serve as upper *and* lower bounds on the true state of A and B at commit time. Each thread can determine locally if a decision to include or exclude an element can be taken safely. Otherwise, the proposal is deferred to the commit process (Algorithm 6.9) which waits until it is certain about A and B before proceeding.

The commit order is given by $\iota(e)$, which is the value of ι in line 5 of Algorithm 6.5. We define $A^{\iota(e)-1}$, $B^{\iota(e)-1}$ as before with CF-2G. Additionally, let $\widehat{A}_e$, $\widehat{B}_e$, $\widetilde{A}_e$, and $\widetilde{B}_e$ be the sets that are returned by Algorithm 6.6.[3] Indeed, these sets are guaranteed to be bounds on $A^{\iota(e)-1}$, $B^{\iota(e)-1}$:

Lemma 6.3. *In* CC-2G, $\forall e \in V$, $\widehat{A}_e \subseteq A^{\iota(e)-1} \subseteq \widetilde{A}_e \backslash e$, *and* $\widehat{B}_e \supseteq B^{\iota(e)-1} \supseteq \widetilde{B}_e \cup e$.

Intuitively, these bounds are maintained by recording potential effects of concurrent transactions in $\widetilde{A}$, $\widetilde{B}$, and only recording the actual effects in $\widehat{A}$, $\widehat{B}$; we leave the full proof to Appendix 6.A. Furthermore, by committing transactions in order ι, we have $\widehat{A} = A^{\iota(e)-1}$ and $\widehat{B} = B^{\iota(e)-1}$ during commit.

Lemma 6.4. *In* CC-2G, *when committing element* e, *we have* $\widehat{A} = A^{\iota(e)-1}$ *and* $\widehat{B} = B^{\iota(e)-1}$.

Corollary 6.5. *Submodularity of* F *implies that the* Δ*'s computed by* CC-2G *satisfy* $\Delta_+^{\min}(e) \leq \Delta_+^{exact}(e) = \Delta_+(e) \leq \Delta_+^{\max}(e)$ *and* $\Delta_-^{\min}(e) \leq \Delta_-^{exact}(e) = \Delta_-(e) \leq \Delta_-^{\max}(e)$.

By using these bounds, CC-2G can determine when it is safe to construct the transaction locally. For failed transactions, the server is able to construct the correct transaction using the true program state. As a consequence we can guarantee that the parallel execution of CC-2G is serializable.

Using Algorithm 6.6 only guarantees serializability but not deterministic execution, since the commit order $\iota(e)$ may not correspond to the order in which elements are processed on Line 1 of Algorithm 6.6. To make CC-2G deterministic, we can enforce that these two orderings concur with each other — Algorithm 6.8 guarantees this by extracting e from π and incrementing the commit order within the same atomic block. This requires additional coordination overhead, and for the rest of this chapter, we consider only the serializable version of CC-2G.

6.6 Analysis of Algorithms

Our two algorithms trade off performance and strong approximation guarantees. The CF-2G algorithm emphasizes speed at the expense of the approximation objective. On the other hand, CC-2G emphasizes the tight 1/2-approximation at the expense of increased coordination. In this section we characterize the reduction in the approximation objective as well as the increased coordination. Our analysis connects the degradation in CC-2g scalability with the

degradation in the CF-2g approximation factor via the maximum inter-processor message delay τ.

Approximation of CF-2G double greedy

Theorem 6.6. *Let F be a non-negative submodular function.* CF-2G *solves the unconstrained problem* $\max_{A \subset V} F(A)$ *with worst-case approximation factor* $E[F(A_{\text{CF}})] \geq \frac{1}{2}F^* - \frac{1}{4}\sum_{i=1}^{N} E[\rho_i]$, *where* A_{CF} *is the output of the algorithm,* F^* *is the optimal value, and* $\rho_i = \max\{\Delta_+^{\max}(e) - \Delta_+(e), \Delta_-^{\max}(e) - \Delta_-(e)\}$ *is the maximum discrepancy in the marginal gain due to the bounds.*

The proof (Appendix 6.A) of Theorem 6.6 follows the structure in [30]. Theorem 6.6 captures the deviation from optimality as a function of width of the bounds which we characterize for two common applications.

Example: max graph cut. For the max cut objective we bound the expected discrepancy in the marginal gain ρ_i in terms of the sparsity of the graph and the maximum inter-processor message delay τ. By applying Theorem 6.6 we obtain the approximation factor $E[F(A^N)] \geq \frac{1}{2}F^* - \tau\frac{\#\text{edges}}{2N}$ which decreases linearly in both the message delays and graph density. In a complete graph, $F^* = \frac{1}{2}\#\text{edges}$, so $E[F(A^N)] \geq F^*\left(\frac{1}{2} - \frac{\tau}{N}\right)$, which makes it possible to scale τ linearly with N while retaining the same approximation factor.

Example: set cover. Consider the simple set cover function, $F(A) = \sum_{l=1}^{L} \min(1, |A \cap S_l|) - \lambda|A| = |\{l : A \cap S_l \neq \emptyset\}| - \lambda|A|$, with $0 < \lambda \leq 1$. We assume that there is some bounded delay τ. Suppose also the S_l's form a partition, so each element e belongs to exactly one set. Then, $\sum_e E[\rho_e] \geq \tau + L(1 - \lambda^\tau)$, which is linear in τ but independent of N.

Correctness of CC-2G

Theorem 6.7. CC-2G *is serializable and therefore solves the unconstrained submodular maximization problem* $\max_{A \subset V} F(A)$ *with approximation* $E[F(A_{\text{CC}})] \geq \frac{1}{2}F^*$, *where* A_{CC} *is the output of the algorithm, and* F^* *is the optimal value.*

The key challenge in the proof (Appendix 6.A) of Theorem 6.7 is to demonstrate that CC-2G guarantees a serializable execution. It suffices to show that CC-2G takes the same decision as SER-2G for each element – locally if it is safe to do so, and otherwise deferring the computation to the server. As an immediate consequence of serializability, we recover the optimal approximation guarantees of the serial SER-2G algorithm.

Scalability of CC-2G

Whenever a transaction is reconstructed on the server, the server needs to wait for all earlier elements to be committed, and is also blocked from committing all later elements. Each failed transaction effectively constitutes a barrier to the parallel processing. Hence, the scalability of CC-2G is dependent on the number of failed transactions.

We can directly bound the number of failed transactions (details in Appendix 6.B) for both the max-cut and set cover example problems. For the max-cut problem with a maximum inter-processor message delay τ we obtain the upper bound $2\tau\frac{\#edges}{N}$. Similarly for set cover the expected *number* of failed transactions is upper-bounded by 2τ. As a consequence, the coordination costs of CC-2G grows at the same rate as the reduction in accuracy of CF-2G. Moreover, the CC-2G algorithm will slow down in settings where the CF-2G algorithm produces sub-optimal solutions.

6.7 Evaluation

We implemented the parallel and serial double greedy algorithms in Java / Scala. Experiments were conducted on Amazon EC2 using one cc2.8xlarge machine, up to 16 threads, for 10 repetitions. We measured the runtime and speedup (ratio of runtime on 1 thread to runtime on p threads). For CF-2G, we measured $F(A_{\text{CF}}) - F(A_{\text{SER}})$, the difference between the objective value on the sets returned by CF-2G and SER-2G. We verified the correctness of CC-2G by comparing the output of CC-2G with SER-2G. We also measured the fraction of transactions that fail in CC-2G. Our parallel algorithms were tested on the max graph cut and set cover problems with two synthetic graphs and three real datasets (Table 6.1). We found that vertices were typically indexed such that nearby vertices in the graph were also close in their indices. To reduce this dependency, we randomly permuted the ordering of vertices.

Graph	# vertices	# edges	Description
Erdos-Renyi	20,000,000	$\approx 2 \times 10^9$	Each edge is included with probability 5×10^{-6}.
ZigZag	25,000,000	2,025,000,000	Expander graph. The 81-regular zig-zag product between the Cayley graph on $\mathbb{Z}_{2500000}$ with generating set $\{\pm 1, \ldots, \pm 5\}$, and the complete graph K_{10}.
Friendster	10,000,000	625,279,786	Subgraph of social network. [87]
Arabic-2005	22,744,080	631,153,669	2005 crawl of Arabic web sites [21, 22, 23].
UK-2005	39,459,925	921,345,078	2005 crawl of the .uk domain [21, 22, 23].
IT-2004	41,291,594	1,135,718,909	2004 crawl of the .it domain [21, 22, 23].

Table 6.1: Synthetic and real graphs used in the evaluation of our parallel algorithms.

We summarize of the key results here with more detailed experiments and discussion in Appendix 6.D. **Runtime, Speedup:** Both parallel algorithms are faster than the serial algorithm with three or more threads, and show good speedup properties as more threads are added ($\sim$ 10x or more for all graphs and both functions). **Objective value:** The objective value of CF-2G decreases with the number of threads, but differs from the serial objective value by less than 0.01%. **Failed transactions:** CC-2G fails more transactions as threads are added, but even with 16 threads, less than 0.015% transactions fail, which has negligible effect on the runtime / speedup.

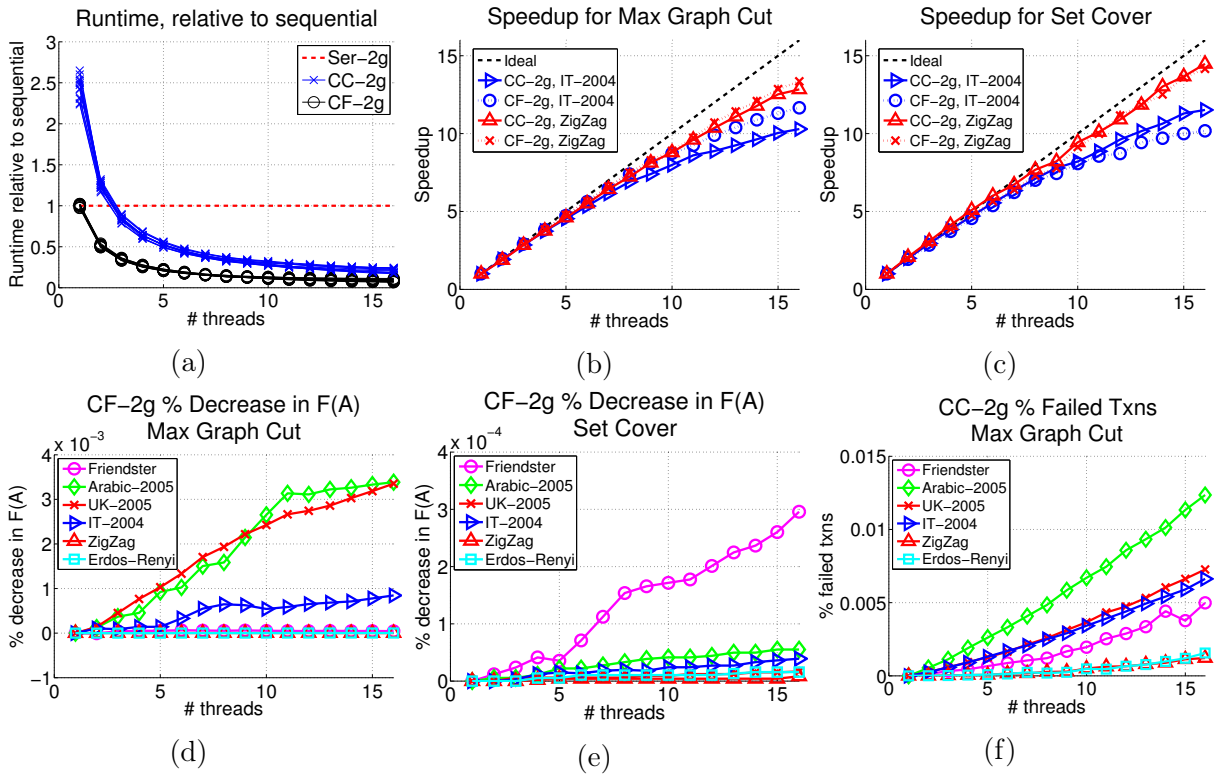


Figure 6.3: Experimental results. Figure 6.3a – runtime of the parallel algorithms as a ratio to that of the serial algorithm. Each curve shows the runtime of a parallel algorithm on a particular graph for a particular function F. Figure 6.3b, 6.3c – speedup (ratio of runtime on one thread to that on p threads). Figure 6.3d, 6.3e – % difference between objective values of SER-2G and CF-2G, i.e. $[F(A_{\text{CF}})/F(A_{\text{SER}}) - 1] \times 100\%$. Figure 6.3f – percentage of transactions that fail in CC-2G on the max graph cut problem.

Adversarial ordering

To highlight the differences in approaches between the two parallel algorithms, we conducted experiments on a ring Cayley expander graph on $\mathbb{Z}_{10^6}$ with generating set $\{\pm 1, \ldots, \pm 1000\}$. The algorithms are presented with an adversarial ordering, without permutation, so vertices close in the ordering are adjacent to one another, and tend to be processed concurrently. This causes CF-2G to make more mistakes, and CC-2G to fail more transactions. While more sophisticated partitioning schemes could improve scalability and eliminate the effect of adversarial ordering, we use the default data partitioning in our experiments to highlight the differences between the two algorithms. As Figure 6.4 shows, CC-2G sacrifices speed to ensure a serializable execution, eventually failing on $> 90\%$ of transactions. On the other hand, CF-2G focuses on speed, resulting in faster runtime, but achieves an objective value that is 20% of $F(A_{\text{SER}})$. We emphasize that we contrived this example to highlight differences between CC-2G and CF-2G, and we do not expect to see such orderings in practice.

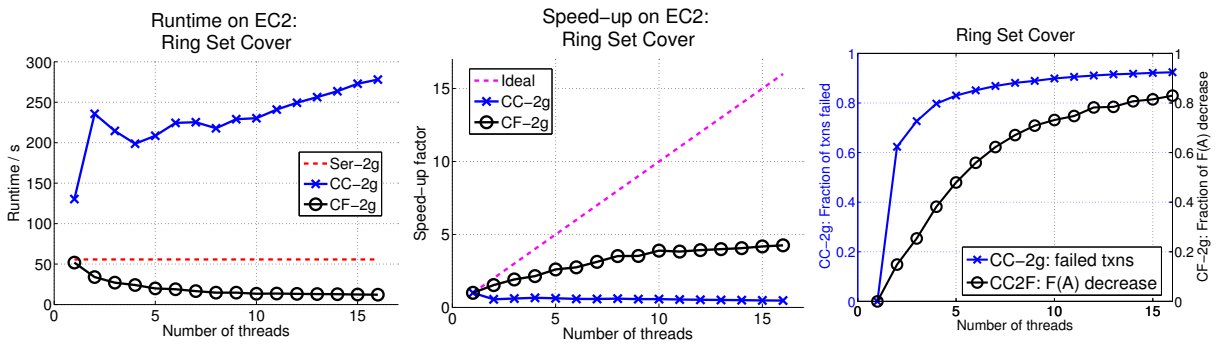


Figure 6.4: Experimental results for set cover problem on a ring expander graph demonstrating that for adversarially constructed inputs we can reduce the optimality of CF-2G and increase coordination costs for CC-2G.

6.8 Related Work

Similar approach: Coordination-free solutions have been proposed for stochastic gradient descent [116] and collapsed Gibbs sampling [3]. More generally, parameter servers [89, 65] apply the coordination-free approach to larger classes of problems. [109] applied concurrency control to parallelize some unsupervised learning algorithms. **Similar problem:** Distributed and parallel greedy submodular maximization is addressed in [104, 85, 134], but only for monotone functions.

6.9 Discussions

By adopting the transaction processing model from parallel database systems, we presented two approaches to parallelizing the double greedy algorithm for unconstrained submodular maximization. We quantified the weaker approximation guarantee of CF-2G and the additional coordination of CC-2G, allowing one to trade off between performance and objective optimality. Our evaluation on large scale data demonstrates the scalability and tradeoffs of the two approaches. Moreover, as the approximation quality of the CF-2G algorithm decreases so does the scalability of the CC-2G algorithm. The choice between the algorithm then reduces to a choice of guaranteed performance and guaranteed optimality.

We believe there are a number of areas for future work. One can imagine a system that allows a smooth interpolation between CF-2G and CC-2G. While both CF-2G and CC-2G can be immediately implemented as distributed algorithms, higher communication costs and delays may pose additional challenges. Finally, other problems such as constrained maximization of monotone / non-monotone functions could potentially be parallelized with the coordination-free and concurrency control frameworks.

6.A Proofs of Theoretical Guarantees

Proofs of $\widetilde{A}_e$, $\widehat{A}_e$, $\widetilde{B}_e$, $\widehat{B}_e$ as bounds on $A^{\iota(e)-1}$ and $B^{\iota(e)-1}$

Lemma 6.1. *In* CF-2G*, for any* $e \in V$*,* $\widehat{A}_e \subseteq A^{\iota(e)-1}$*, and* $\widehat{B}_e \supseteq B^{\iota(e)-1}$*.*

Proof. For any element e, we write T_e to denote the time at which Algorithm 6.4 line 8 is executed. Consider any element $e' \in V$. If $e' \in \widehat{A}_e$, it must be the case that the algorithm set $\widehat{A}(e')$ to 1 (line 10) before T_e, which implies $\iota(e') < \iota(e)$, and hence $e' \in A^{\iota(e)-1}$. So $\widehat{A}_e \subseteq A^{\iota(e)-1}$.

Similarly, if $e' \notin \widehat{B}_e$, then the algorithm set $\widehat{B}(e')$ to 0 (line 11) before T_e, so $\iota(e') < \iota(e)$. Also, $e' \notin A$ because the execution of line 11 excludes the execution of line 10. Therefore, $e' \notin A^{\iota(e)-1}$, and $e' \notin B^{\iota(e)-1}$. So $\widehat{B}_e \supseteq B^{\iota(e)-1}$. □

Lemma 6.3. *In* CC-2G*,* $\forall e \in V$*,* $\widehat{A}_e \subseteq A^{\iota(e)-1} \subseteq \widetilde{A}_e \backslash e$*, and* $\widehat{B}_e \supseteq B^{\iota(e)-1} \supseteq \widetilde{B}_e \cup e$*.*

Proof. Clearly, $e \in \widetilde{B}_e \cup e$ but $e \notin \widetilde{A}_e \backslash e$. By definition, $e \in B^{\iota(e)-1}$ but $e \notin A^{\iota(e)-1}$. CC-2G only modifies $\widehat{A}(e)$ and $\widehat{B}(e)$ when committing the transaction on e, which occurs after obtaining the bounds in getGuarantee(e), so $e \in \widehat{B}_e$ but $e \notin \widehat{A}_e$.

Consider any $e' \neq e$. Suppose $e' \in \widehat{A}_e$. This is only possible if we have committed the transaction on e' before the call getGuarantee(e), so it must be the case that $\iota(e') < \iota(e)$. Thus, $e' \in A^{\iota(e)-1}$.

Now suppose $e' \in A^{\iota(e)-1}$. By definition, this implies $\iota(e') < \iota(e)$ and $e' \in A$. Hence, it must be the case that we have already set $\widetilde{A}(e') \leftarrow 1$ (by the ordering imposed by ι on Line 5), but never execute $\widetilde{A}(e') \leftarrow 0$ (since $e' \in A$), so $e' \in \widetilde{A}_e$.

An analogous argument shows $e' \notin \widehat{B}_e \implies e' \notin B^{\iota(e)-1} \implies e' \notin \widetilde{B}_e \cup e$. □

Lemma 6.4. *In* CC-2G*, when committing element* e*, we have* $\widehat{A} = A^{\iota(e)-1}$ *and* $\widehat{B} = B^{\iota(e)-1}$*.*

Proof. Algorithm 6.9 Line 1 ensures that all elements ordered before e are committed, and that no element ordered after e are committed. This suffices to guarantee that $e' \in \widehat{A} \iff e' \in A^{\iota(e)-1}$ and $e' \in \widehat{B} \iff e' \in B^{\iota(e)-1}$. □

Proof of serial equivalence of CC-2G

Theorem 6.7. CC-2G *is serializable and therefore solves the unconstrained submodular maximization problem* $\max_{A \subset V} F(A)$ *with approximation* $E[F(A_{\mathrm{CC}})] \geq \frac{1}{2}F^*$*, where* A_{CC} *is the output of the algorithm, and* F^* *is the optimal value.*

Proof. We will denote by A^i_{seq}, B^i_{seq} the sets generated by SER-2G, reserving A^i, B^i for sets generated by the CC-2G algorithm. It suffices to show by induction that $A^i_{seq} = A^i$ and $B^i_{seq} = B^i$. For the base case, $A^0 = \emptyset = A^0_{seq}$, and $B^0 = V = B^0_{seq}$. Consider any element e. The CC-2G algorithm includes $e \in A$ iff $u_e < [\Delta^{\min}_+(e)]_+([\Delta^{\min}_+(e)]_+ + [\Delta^{\max}_-(e)]_+)^{-1}$ on Algorithm 6.5 Line 6 or $u_e < [\Delta^{\text{exact}}_+(e)]_+([\Delta^{\text{exact}}_+(e)]_+ + [\Delta^{\text{exact}}_-(e)]_+)^{-1}$ on Algorithm 6.9 Line 5. In both cases, Corollary 6.5 implies $u_e < [\Delta_+(e)]_+([\Delta_+(e)]_+ + [\Delta_-(e)]_+)^{-1}$. By induction, $A^{\iota(e)-1} = A^{\iota(e)-1}_{seq}$ and $B^{\iota(e)-1} = B^{\iota(e)-1}_{seq}$, so the threshold is exactly that computed by SER-2G. Hence, the CC-2G algorithm includes $e \in A$ iff SER-2G includes $e \in A$. (An analogous argument works for the case where e is excluded from B.) □

Proof of bound for CF-2G

We follow the proof outline of [30].

Consider an ordering ι inducted by running CF-2G. For convenience, we will use i to flexibly denote the element e and its ordering $\iota(e)$.

Let OPT be an optimal solution to the problem. Define $O^i := (OPT \cup A^i) \cap B^i$. Note that O^i coincides with A^i and B^i on elements $1, \ldots, i$, and O^i coincides with OPT on elements $i+1, \ldots, n$. Hence,

$$O^i \backslash (i+1) \supseteq A^i$$
$$O^i \cup (i+1) \subseteq B^i.$$

Lemma 6.8. *For every* $1 \le i \le n$, $\Delta_+(i) + \Delta_-(i) \ge 0$.

Proof. This is just Lemma II.1 of [30]. □

Lemma 6.9. *Let* $\rho_i = \max\{\Delta^{\max}_+(e) - \Delta_+(e), \Delta^{\max}_-(e) - \Delta_-(e)\}$. *For every* $1 \le i \le n$,

$$E[F(O^{i-1}) - F(O^i)] \le \frac{1}{2} E[F(A^i) - F(A^{i-1}) + F(B^i) - F(B^{i-1}) + \rho_i].$$

Proof. We follow the proof outline of [30]. First, note that it suffices to prove the inequality conditioned on knowing A^{i-1}, $\widehat{A}_i$ and $\widehat{B}_i$, then applying the law of total expectation. Under this conditioning, we also know B^{i-1}, O^{i-1}, $\Delta_+(i)$, $\Delta^{\max}_+(i)$, $\Delta_-(i)$, and $\Delta^{\max}_-(i)$.

We consider the following 6 cases.

Case 1: $0 < \Delta_+(i) \le \Delta^{\max}_+(i)$, $0 \le \Delta^{\max}_-(i)$. Since both $\Delta^{\max}_+(i) > 0$ and $\Delta^{\max}_-(i) > 0$, the probability of including i is just $\Delta^{\max}_+(i)/(\Delta^{\max}_+(i) + \Delta^{\max}_-(i))$, and the probability of

excluding i is $\Delta_-^{\max}(i)/(\Delta_+^{\max}(i)+\Delta_-^{\max}(i))$.

$$\begin{aligned}
E[F(A^i)-F(A^{i-1})|A^{i-1},\widehat{A}_i,\widehat{B}_i] &= \frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(A^{i-1}\cup i)-F(A^{i-1})) \\
&= \frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}\Delta_+(i) \\
&\geq \frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(\Delta_+^{\max}(i)-\rho_i) \\
E[F(B^i)-F(B^{i-1})|A^{i-1},\widehat{A}_i,\widehat{B}_i] &= \frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(B^{i-1}\backslash i)-F(B^{i-1})) \\
&= \frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}\Delta_-(i) \\
&\geq \frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(\Delta_-^{\max}(i)-\rho_i)
\end{aligned}$$

$$\begin{aligned}
&E[F(O^{i-1})-F(O^i)|A^{i-1},\widehat{A}_i,\widehat{B}_i] \\
=&\frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(O^{i-1})-F(O^{i-1}\cup i)) \\
&+\frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(O^{i-1})-F(O^{i-1}\backslash i)) \\
=&\begin{cases}\frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(O^{i-1})-F(O^{i-1}\cup i)) & \text{if } i\notin OPT \\ \frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(O^{i-1})-F(O^{i-1}\backslash i)) & \text{if } i\in OPT\end{cases} \\
\leq&\begin{cases}\frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(B^{i-1}\backslash i)-F(B^{i-1})) & \text{if } i\notin OPT \\ \frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}(F(A^{i-1}\cup i)-F(A^{i-1})) & \text{if } i\in OPT\end{cases} \\
=&\begin{cases}\frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}\Delta_-(i) & \text{if } i\notin OPT \\ \frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}\Delta_+(i) & \text{if } i\in OPT\end{cases} \\
\leq&\begin{cases}\frac{\Delta_+^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}\Delta_-^{\max}(i) & \text{if } i\notin OPT \\ \frac{\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}\Delta_+^{\max}(i) & \text{if } i\in OPT\end{cases} \\
=&\frac{\Delta_+^{\max}(i)\Delta_-^{\max}(i)}{\Delta_+^{\max}(i)+\Delta_-^{\max}(i)}
\end{aligned}$$

where the first inequality is due to submodularity: $O^{i-1}\backslash i \supseteq A^{i-1}$ and $O^{i-1}\cup i \subseteq B^{i-1}$.

Putting the above inequalities together:

$$
\begin{aligned}
&E\left[F(O^{i-1}) - F(O^i) - \frac{1}{2}\Big(F(A^i) - F(A^{i-1}) + F(B^i) - F(B^{i-1}) + \rho_i\Big)\middle| A^{i-1}, \widehat{A}_i, \widehat{B}_i\right] \\
&\leq \frac{1/2}{\Delta_+^{\max}(i) + \Delta_-^{\max}(i)}\Big[2\Delta_+^{\max}(i)\Delta_-^{\max}(i) - \Delta_-^{\max}(i)(\Delta_-^{\max}(i) - \rho_i) \\
&\qquad\qquad - \Delta_+^{\max}(i)(\Delta_+^{\max}(i) - \rho_i)\Big] - \frac{1}{2}\rho_i \\
&= \frac{1/2}{\Delta_+^{\max}(i) + \Delta_-^{\max}(i)}\Big[-(\Delta_+^{\max}(i) - \Delta_-^{\max}(i))^2 + \rho_i(\Delta_+^{\max}(i) + \Delta_-^{\max}(i))\Big] - \frac{1}{2}\rho_i \\
&\leq \frac{\frac{1}{2}\rho_i(\Delta_+^{\max}(i) + \Delta_-^{\max}(i))}{\Delta_+^{\max}(i) + \Delta_-^{\max}(i)} - \frac{1}{2}\rho_i \\
&= 0.
\end{aligned}
$$

Case 2: $0 < \Delta_+(i) \leq \Delta_+^{\max}(i)$, $\Delta_-^{\max}(i) < 0$. In this case, the algorithm always choses to include i, so $A^i = A^{i-1} \cup i$, $B^i = B^{i-1}$ and $O^i = O^{i-1} \cup i$:

$$
\begin{aligned}
E[F(A^i) - F(A^{i-1})|A^{i-1}, \widehat{A}_i, \widehat{B}_i] &= F(A^{i-1} \cup i) - F(A^{i-1}) = \Delta_+(i) > 0 \\
E[F(B^i) - F(B^{i-1})|A^{i-1}, \widehat{A}_i, \widehat{B}_i] &= F(B^{i-1}) - F(B^{i-1}) = 0 \\
E[F(O^{i-1}) - F(O^i)|A^{i-1}, \widehat{A}_i, \widehat{B}_i] &= F(O^{i-1}) - F(O^{i-1} \cup i) \\
&\leq \begin{cases} 0 & \text{if } i \in OPT \\ F(B^{i-1} \backslash i) - F(B^{i-1}) & \text{if } i \notin OPT \end{cases} \\
&= \begin{cases} 0 & \text{if } i \in OPT \\ \Delta_-(i) & \text{if } i \notin OPT \end{cases} \\
&\leq 0 \\
&< \frac{1}{2}E[F(A^i) - F(A^{i-1}) + F(B^i) - F(B^{i-1}) + \rho_i|A^{i-1}, \widehat{A}_i, \widehat{B}_i]
\end{aligned}
$$

where the first inequality is due to submodularity: $O^{i-1} \cup i \subseteq B^{i-1}$.

Case 3: $\Delta_+(i) \leq 0 < \Delta_+^{\max}(i)$, $0 < \Delta_-(i) < \Delta_-^{\max}(i)$. Analogous to Case 1.

Case 4: $\Delta_+(i) \leq 0 < \Delta_+^{\max}(i)$, $\Delta_-(i) \leq 0$. This is not possible, by Lemma 6.8.

Case 5: $\Delta_+(i) \leq \Delta_+^{\max}(i) \leq 0$, $0 < \Delta_-(i) \leq \Delta_-^{\max}(i)$. Analogous to Case 2.

Case 6: $\Delta_+(i) \leq \Delta_+^{\max}(i) \leq 0$, $\Delta_-(i) \leq 0$. This is not possible, by Lemma 6.8.

□

We will now prove the main theorem.

Theorem 6.6. *Let F be a non-negative submodular function.* CF-2G *solves the unconstrained problem* $\max_{A\subset V} F(A)$ *with worst-case approximation factor* $E[F(A_{\mathrm{CF}})] \geq \frac{1}{2}F^* - \frac{1}{4}\sum_{i=1}^{N} E[\rho_i]$, *where* A_{CF} *is the output of the algorithm,* F^* *is the optimal value, and* $\rho_i = \max\{\Delta_+^{\max}(e) - \Delta_+(e), \Delta_-^{\max}(e) - \Delta_-(e)\}$ *is the maximum discrepancy in the marginal gain due to the bounds.*

Proof. Summing up the statement of Lemma 6.9 for all i gives us a telescoping sum, which reduces to:

$$\begin{aligned} E[F(O^0) - F(O^n)] &\leq \frac{1}{2}E[F(A^n) - F(A^0) + F(B^n) - F(B^0)] + \frac{1}{2}\sum_{i=1}^{n} E[\rho_i] \\ &\leq \frac{1}{2}E[F(A^n) + F(B^n)] + \frac{1}{2}\sum_{i=1}^{n} E[\rho_i]. \end{aligned}$$

Note that $O^0 = OPT$ and $O^n = A^n = B^n$, so $E[F(A^n)] \geq \frac{1}{2}F^* - \frac{1}{4}\sum_i E[\rho_i]$. □

Example: max graph cut

Let $C_i = (A^{i-1}\backslash\widehat{A}_i) \cup (\widehat{B}_i\backslash B^{i-1})$ be the set of elements concurrently processed with i but ordered after i, and $D_i = B^i\backslash A^i$ be the set of elements ordered after i. Denote $\bar{A}_i = V\backslash(\widehat{A}_i \cup C_i \cup D_i) = \{1,\dots,i\}\backslash\widehat{A}_i$ be the elements up to i that are not included in $\widehat{A}_i$. Let $w_i(S) = \sum_{j\in S,(i,j)\in E} w(i,j)$. For the max graph cut function, it is easy to see that

$$\begin{aligned} \Delta_+ &\geq -w_i(\widehat{A}_i) - w_i(C_i) + w_i(D_i) + w_i(\bar{A}_i) \\ \Delta_+^{\max} &= -w_i(\widehat{A}_i) + w_i(C_i) + w_i(D_i) + w_i(\bar{A}_i) \\ \Delta_- &\geq +w_i(\widehat{A}_i) - w_i(C_i) + w_i(D_i) - w_i(\bar{A}_i) \\ \Delta_-^{\max} &= +w_i(\widehat{A}_i) + w_i(C_i) + w_i(D_i) - w_i(\bar{A}_i) \end{aligned}$$

Thus, we can see that $\rho_i \leq 2w_i(C_i)$.

Suppose we have bounded delay τ, so $|C_i| \leq \tau$. Then $w_i(C_i)$ has a hypergeometric distribution with mean $\frac{\deg(i)}{N}\tau$, and $E[\rho_i] \leq 2\tau\frac{\deg(i)}{N}$. The approximation of the hogwild algorithm is then $E[F(A^n)] \geq \frac{1}{2}F^* - \tau\frac{\#\mathrm{edges}}{2N}$. In sparse graphs, the hogwild algorithm is off by a small additional term, which albeit grows linearly in τ. In a complete graph, $F^* = \frac{1}{2}\#\mathrm{edges}$, so $E[F(A^n)] \geq F^*\left(\frac{1}{2} - \frac{\tau}{N}\right)$, which makes it possible to scale τ linearly with N while retaining the same approximation factor.

Example: set cover

Consider the simple set cover function, for $\lambda < L/N$:

$$F(A) = \sum_{l=1}^{L} \min(1, |A\cap S_l|) - \lambda|A| = |\{l : A\cap S_l \neq \emptyset\}| - \lambda|A|.$$

We assume that there is some bounded delay τ.

Suppose also that the sets S_l form a partition, so each element e belongs to exactly one set. Let $n_l = |S_l|$ denote the size of S_l. Given any ordering π, let e_l^t be the tth element of S_l in the ordering, i.e. $|\{e' : \pi(e') \leq \pi(e_l^t) \wedge e' \in S_l\}| = t$.

For any $e \in S_l$, we get

$$\begin{aligned}
\Delta_+(e) &= -\lambda + 1\{A^{\iota(e)-1} \cap S_l = \emptyset\} \\
\Delta_+^{\max}(e) &= -\lambda + 1\{\widehat{A}_e \cap S_l = \emptyset\} \\
\Delta_-(e) &= +\lambda - 1\{B^{\iota(e)-1} \backslash e \cap S_l = \emptyset\} \\
\Delta_-^{\max}(e) &= +\lambda - 1\{\widehat{B}_e \backslash e \cap S_l = \emptyset\}
\end{aligned}$$

Let η be the position of the first element of S_l to be accepted, i.e. $\eta = \min\{t : e_l^t \in A \cap S_l\}$. (For convenience, we set $\eta = n_l$ if $A \cap S_l = \emptyset$.) We first show that η is independent of π: for $\eta < n_l$,

$$\begin{aligned}
P(\eta|\pi) &= \frac{\Delta_+^{\max}(e_l^\eta)}{\Delta_+^{\max}(e_l^\eta) + \Delta_-^{\max}(e_l^\eta)} \prod_{t=1}^{\eta-1} \frac{\Delta_-^{\max}(e_l^t)}{\Delta_+^{\max}(e_l^t) + \Delta_-^{\max}(e_l^t)} \\
&= \frac{1-\lambda}{1-\lambda+\lambda} \prod_{t=1}^{\eta-1} \frac{\lambda}{1-\lambda+\lambda} \\
&= (1-\lambda)\lambda^{\eta-1},
\end{aligned}$$

and $P(\eta = n_l|\pi) = \lambda^{\eta-1}$.

Note that, $\Delta_-^{\max}(e) - \Delta_-(e) = 1$ iff $e = e_l^{n_l}$ is the last element of S_l in the ordering, there are no elements accepted up to $\widehat{B}_{e_l^{n_l}} \backslash e_l^{n_l}$, and there is some element e' in $\widehat{B}_{e_l^{n_l}} \backslash e_l^{n_l}$ that is rejected and not in $B^{\iota(e_l^{n_l})-1}$. Denote by $m_l \leq \min(\tau, n_l - 1)$ the number of elements before $e_l^{n_l}$ that are inconsistent between $\widehat{B}_{e_l^{n_l}}$ and $B^{\iota(e_l^{n_l})-1}$. Then $\mathbb{E}[\Delta_-^{\max}(e_l^{n_l}) - \Delta_-(e_l^{n_l})] = P(\Delta_-^{\max}(e_l^{n_l}) \neq \Delta_-(e_l^{n_l}))$ is

$$\lambda^{n_l-1-m_l}(1-\lambda^{m_l}) \quad = \quad \lambda^{n_l-1}(\lambda^{-m_l}-1) \quad \leq \quad \lambda^{n_l-1}(\lambda^{-\min(\tau,n_l-1)}-1) \quad \leq \quad 1-\lambda^\tau.$$

If $\lambda = 1$, $\Delta_+^{\max}(e) \leq 0$, so no elements before $e_l^{n_l}$ will be accepted, and $\Delta_-^{\max}(e_l^{n_l}) = \Delta_-(e_l^{n_l})$.

On the other hand, $\Delta_+^{\max}(e) - \Delta_+(e) = 1$ iff $(A^{\iota(e)-1} \backslash \widehat{A}_e) \cap S_l \neq \emptyset$, that is, if an element has been accepted in A but not yet observed in $\widehat{A}_e$. Since we assume a bounded delay, only

the first τ elements after the first acceptance of an $e \in S_l$ may be affected.

$$\begin{aligned}
&\mathbb{E}\left[\sum_{e\in S_l} \Delta_+^{\max}(e) - \Delta_+(e)\right] \\
&= \mathbb{E}[\#\{e : e \in S_l \wedge e_l^\eta \in A^{\iota(e)-1} \wedge e_l^\eta \notin \widehat{A}_e\}] \\
&= \mathbb{E}[\mathbb{E}[\#\{e : e \in S_l \wedge e_l^\eta \in A^{\iota(e)-1} \wedge e_l^\eta \notin \widehat{A}_e\} \mid \eta = t, \pi(e_l^t) = k]] \\
&= \sum_{t=1}^{n_l} \sum_{k=t}^{N-n+t} P(\eta = t, \pi(e_l^t) = k)\mathbb{E}[\#\{e : e \in S_l \wedge e_l^\eta \in A^{\iota(e)-1} \wedge e_l^\eta \notin \widehat{A}_e\} \mid \eta = t, \pi(e_l^t) = k] \\
&= \sum_{t=1}^{n_l} P(\eta = t) \sum_{k=t}^{N-n+t} P(\pi(e_l^t) = k)\mathbb{E}[\#\{e : e \in S_l \wedge e_l^\eta \in A^{\iota(e)-1} \wedge e_l^\eta \notin \widehat{A}_e\} \mid \eta = t, \pi(e_l^t) = k].
\end{aligned}$$

Under the assumption that every ordering π is equally likely, and a bounded delay τ, conditioned on $\eta = t, \pi(e_l^t) = k$, the random variable $\#\{e : e \in S_l \wedge e_l^\eta \in A^{\iota(e)-1} \wedge e_l^\eta \notin \widehat{A}_e\}$ has hypergeometric distribution with mean $\frac{n_l - t}{N-k}\tau$. Also, $P(\pi(e_l^t) = k) = \frac{n_l}{N}\binom{n-1}{t-1}\binom{N-n}{k-t}/\binom{N-1}{k-1}$, so the above expression becomes

$$\begin{aligned}
&\mathbb{E}\left[\sum_{e\in S_l} \Delta_+^{\max}(e) - \Delta_+(e)\right] \\
&= \sum_{t=1}^{n_l} P(\eta = t) \sum_{k=t}^{N-n+t} \frac{n_l}{N} \frac{\binom{n-1}{t-1}\binom{N-n}{k-t}}{\binom{N-1}{k-1}} \frac{n-t}{N-k}\tau \\
&= \frac{n_l}{N}\tau \sum_{t=1}^{n_l} P(\eta = t) \sum_{k=t}^{N-n+t} \frac{\binom{k-1}{t-1}\binom{N-k}{n-t}}{\binom{N-1}{n-1}} \frac{n-t}{N-k} && \text{(symmetry of hypergeometric)} \\
&= \frac{n_l}{N}\tau \sum_{t=1}^{n_l} \frac{P(\eta = t)}{\binom{N-1}{n-1}} \sum_{k=t}^{N-n+t} \binom{k-1}{t-1}\binom{N-k-1}{n-t-1} \\
&= \frac{n_l}{N}\tau \sum_{t=1}^{n_l} \frac{P(\eta = t)}{\binom{N-1}{n-1}} \binom{N-1}{n-1} && \text{(Lemma 6.10, } a = N-2, b = n_l - 2, j = 1) \\
&= \frac{n_l}{N}\tau \sum_{t=1}^{n_l} P(\eta = t) \\
&= \frac{n_l}{N}\tau.
\end{aligned}$$

Since $\Delta_+^{\max}(e) \geq \Delta_+(e)$ and $\Delta_-^{\max}(e) \geq \Delta_-^{\max}(e)$, we have that $\rho_e \leq \Delta_+^{\max}(e) - \Delta_+(e) +$

$\Delta_-^{\max}(e) - \Delta_-(e)$, so

$$\begin{aligned}
\mathbb{E}\left[\sum_e \rho_e\right] &= \mathbb{E}\left[\sum_e \Delta_+^{\max}(e) - \Delta_+(e) + \Delta_-^{\max}(e) - \Delta_-(e)\right] \\
&= \sum_l \mathbb{E}\left[\sum_{e\in S_l} \Delta_+^{\max}(e) - \Delta_+(e)\right] + \mathbb{E}\left[\sum_{e\in S_l} \Delta_-^{\max}(e) - \Delta_-(e)\right] \\
&\leq \tau\frac{\sum_l n_l}{N} + L(1-\lambda^\tau) \\
&= \tau + L(1-\lambda^\tau).
\end{aligned}$$

Note that $\mathbb{E}\left[\sum_e \rho_e\right]$ does not depend on N and is linear in τ. Also, if $\tau = 0$ in the sequential case, we get $\mathbb{E}\left[\sum_e \rho_e\right] \leq 0$.

6.B Upper Bound on Expected Number of Failed Transactions

Let N be the number of elements, i.e. the cardinality of the ground set. Let $C_i = (A^{i-1}\backslash\widehat{A}_i) \cup (\widehat{B}_i\backslash B^{i-1})$. We assume a bounded delay τ, so that $|C_i| \leq \tau$ for all i.

We call element i *dependent* on i' if $\exists A, F(A\cup i) - F(A) \neq F(A\cup i'\cup i) - F(A\cup i')$ or $\exists B, F(B\backslash i) - F(B) \neq F(B\cup i'\backslash i) - F(B\cup i')$, i.e. the result of the processing i' will affect the computation of Δ's for i. For example, for the graph cut problem, every vertex is dependent on its neighbors; for the separable sums problem, i is dependent on $\{i' : \exists S_l, i \in S_l, i' \in S_l\}$.

Let n_i be the number of elements that i is dependent on. Now, we note that if C_i does not contain any elements on which i is dependent, then $\Delta_+^{\max}(i) = \Delta_+(i) = \Delta_+^{\min}(i)$ and $\Delta_-^{\max}(i) = \Delta_-(i) = \Delta_-^{\min}(i)$, so i will not fail. Conversely, if i fails, there must be some element $i' \in C_i$ such that i is dependent on i'.

$$\begin{aligned}
E(\text{number of failed transactions}) &= \sum_i P(i \text{ fails}) \\
&\leq \sum_i P(\exists i' \in C_i, i \text{ depends on } i') \\
&\leq \sum_i E\left[\sum_{i'\in C_i} 1\{i \text{ depends on } i'\}\right] \\
&\leq \sum_i \frac{\tau n_i}{N}
\end{aligned}$$

The last inequality follows from the fact that $\sum_{i'\in C_i} 1\{i \text{ depends on } i'\}$ is a hypergeometric random variable and $|C_i| \leq \tau$.

Note that the bound established above is generic to functions F, and additional knowledge of F can lead to better analyses on the algorithm's concurrency.

Upper bound for max graph cut

By applying the above generic bound, we see that the number of failed transactions for max graph cut is upper bounded by $\frac{\tau}{N}\sum_i n_i = \tau\frac{2\#\text{edges}}{N}$.

Upper bound for set cover

For the set cover problem, we can provide a tighter bound on the number of failed items. We make the same assumptions as before in the CF-2G analysis, i.e. the sets S_l form a partition of V, there is a bounded delay τ.

Observe that for any $e \in S_l$, $\Delta_-^{\min}(e) \neq \Delta_-^{\max}(e)$ if $\widehat{B}_e \backslash e \cap S_l \neq \emptyset$ and $\widetilde{B}_e \backslash e \cap S_l = \emptyset$. This is only possible if $e_l^{n_l} \notin \widetilde{B}_e$ and $\widetilde{B}_e \supset \widehat{A}_e \cap S_l = \emptyset$, that is $\pi(e) \geq \pi(e_l^{n_l}) - \tau$ and $\forall e' \in S_l, (\pi(e') < \pi(e_l^{n_l}) - \tau) \implies (e' \notin A)$. The latter condition is achieved with probability $\lambda^{n_l - m_l}$, where $m_l = \#\{e' : \pi(e') \geq \pi(e_l^{n_l}) - \tau\}$. Thus,

$$
\begin{aligned}
&\mathbb{E}\left[\#\{e : \Delta_-^{\min}(e) \neq \Delta_-^{\max}(e)\}\right] \\
&= \mathbb{E}[m_l \; 1(\forall e' \in S_l, (\pi(e') < \pi(e_l^{n_l}) - \tau) \implies (e' \notin A))] \\
&= \mathbb{E}[\mathbb{E}[m_l \; 1(\forall e' \in S_l, (\pi(e') < \pi(e_l^{n_l}) - \tau) \implies (e' \notin A))|u_{1:N}]] \\
&= \mathbb{E}[m_l \; \mathbb{E}[1(\forall e' \in S_l, (\pi(e') < \pi(e_l^{n_l}) - \tau) \implies (e' \notin A))|u_{1:N}]] \\
&= \mathbb{E}[m_l \lambda^{n_l - m_l}] \\
&\leq \lambda^{(n_l - \tau)_+}\mathbb{E}[m_l] \\
&= \lambda^{(n_l - \tau)_+}\mathbb{E}[\mathbb{E}[m_l|\pi(e_l^{n_l}) = k]] \\
&= \lambda^{(n_l - \tau)_+}\sum_{k=n_l}^{N} P(\pi(e_l^{n_l}) = k)\mathbb{E}[m_l|\pi(e_l^{n_l}) = k].
\end{aligned}
$$

Conditioned on $\pi(e_l^{n_l}) = k$, m_l is a hypergeometric random variable with mean $\frac{n_l - 1}{k-1}\tau$. Also

$P(\pi(e_l^{n_l}) = k) = \frac{n_l}{N}\binom{n_l-1}{0}\binom{N-n_l}{N-k}/\binom{N-1}{N-k}$. The above expression is therefore

$$
\begin{aligned}
&\mathbb{E}\left[\#\{e : \Delta_-^{\min}(e) \neq \Delta_-^{\max}(e)\}\right] \\
&= \lambda^{(n_l-\tau)_+} \sum_{k=n_l}^{N} \frac{n_l}{N} \frac{\binom{n_l-1}{0}\binom{N-n_l}{N-k}}{\binom{N-1}{N-k}} \frac{n_l-1}{k-1}\tau \\
&= \lambda^{(n_l-\tau)_+} \frac{n_l}{N}\tau \sum_{k=n_l}^{N} \frac{\binom{N-k}{0}\binom{k-1}{n_l-1}}{\binom{N-1}{n_l-1}} \frac{n_l-1}{k-1} && \text{(symmetry of hypergeometric)} \\
&= \lambda^{(n_l-\tau)_+} \frac{n_l}{N} \frac{\tau}{\binom{N-1}{n_l-1}} \sum_{k=n_l}^{N} \binom{N-k}{0}\binom{k-2}{n_l-2} \\
&= \lambda^{(n_l-\tau)_+} \frac{n_l}{N} \frac{\tau}{\binom{N-1}{n_l-1}} \binom{N-1}{n_l-1} && \text{(Lemma 6.10, } a = N-2,\ b = n_l-2,\ j = 2,\ t = n_l\text{)} \\
&= \lambda^{(n_l-\tau)_+} \frac{n_l}{N}\tau.
\end{aligned}
$$

Now we consider any element $e \in S_l$ with $\pi(e) < \pi(e_l^{n_l}) - \tau$ that fails. (Note that $e_l^{n_l} \in \widehat{B}_e$ and $\widetilde{B}_e$, so $\Delta_-^{\min}(e) = \Delta_-^{\max}(e) = \lambda$.) It must be the case that $\widehat{A}_e \cap S_l = \emptyset$, for otherwise $\Delta_+^{\min}(e) = \Delta_+^{\max}(e) = -\lambda$ and it does not fail. This implies that $\Delta_+^{\max}(e) = 1 - \lambda \geq u_i$. At commit, if $A^{\iota(e)-1} \cap S_l = \emptyset$, we accept e into A. Otherwise, $A^{\iota(e)-1} \cap S_l \neq \emptyset$, which implies that some other element $e' \in S_l$ has been accepted. Thus, we conclude that every element $e \in S_l$ that fails must be within τ of the first accepted element e_l^{η} $in S_l$. The expected number of such elements is exactly as we computed in the CF-2Ganalysis: $\frac{n_l}{N}\tau$.

Hence, the expected number of elements that fails is upper bounded as

$$
\begin{aligned}
\mathbb{E}[\#\text{failed transactions}] &\leq \sum_l (1 + \lambda^{(n_l-\tau)_+}) \frac{n_l}{N}\tau \\
&\leq \sum_l 2\frac{n_l}{N}\tau \\
&= 2\tau.
\end{aligned}
$$

Technical Lemma

Lemma 6.10. $\sum_{k=t}^{a-b+t} \binom{k-j}{t-j}\binom{a-k+j}{b-t+j} = \binom{a+1}{b+1}$.

Proof.

$$
\begin{aligned}
&\sum_{k=t}^{a-b+t} \binom{k-j}{t-j}\binom{a-k+j}{b-t+j} \\
&= \sum_{k'=0}^{a-b} \binom{k'+t-j}{t-j}\binom{a-k'-t+j}{b-t+j} \\
&= \sum_{k'=0}^{a-b} \binom{k'+t-j}{k'}\binom{a-k'-t+j}{a-b-k'} && \text{(symmetry of binomial coeff.)} \\
&= (-1)^{a-b}\sum_{k'=0}^{a-b} \binom{-t+j-1}{k'}\binom{-b+t-j-1}{a-b-k'} && \text{(upper negation)} \\
&= (-1)^{a-b}\binom{-b-2}{a-b} && \text{(Chu-Vandermonde's identity)} \\
&= \binom{a+1}{a-b} && \text{(upper negation)} \\
&= \binom{a+1}{b+1} && \text{(symmetry of binomial coeff.)}
\end{aligned}
$$

□

6.C Parallel Algorithms for Separable Sums

For some functions F, we can maintain sketches / statistics to aid the computation of $\Delta_+^{\max}$, $\Delta_-^{\max}$, $\Delta_+^{\min}$, $\Delta_-^{\min}$. In particular, we consider functions of the form $F(X) = \sum_{l=1}^{L} g\left(\sum_{i\in X\cup S_l} w_l(i)\right) - \lambda\sum_{i\in X} v(i)$, where $S_l \subseteq V$ are (possibly overlapping) groups of elements in the ground set, g is a non-decreasing concave scalar function, and $w_l(i)$ and $v(i)$ are non-negative scalar weights. An example of such functions is set cover $F(A) = \sum_{l=1}^{L}\min(1, |A\cup S_l|) - \lambda|A|$. It is easy to see that $F(X\cup e) - F(X) = \sum_{l:e\in S_l}\left[g\left(w_l(e) + \sum_{i\in X\cup S_l} w_l(i)\right) - g\left(\sum_{i\in X\cup S_l} w_l(i)\right)\right] - \lambda v(e)$. Define

$$
\widehat{\alpha}_l = \sum_{j\in\widehat{A}\cup S_l} w_l(j), \qquad \widehat{\alpha}_{l,e} = \sum_{j\in\widehat{A}_e\cup S_l} w_l(j), \qquad \alpha_l^{\iota(e)-1} = \sum_{j\in A^{\iota(e)-1}\cup S_l} w_l(j).
$$

$$
\widehat{\beta}_l = \sum_{j\in\widehat{B}\cup S_l} w_l(j), \qquad \widehat{\beta}_{l,e} = \sum_{j\in\widehat{B}_e\cup S_l} w_l(j), \qquad \beta_l^{\iota(e)-1} = \sum_{j\in B^{\iota(e)-1}\cup S_l} w_l(j).
$$

CF-2G for separable sums F

Algorithm 6.10 updates $\widehat{\alpha}_l$ and $\widehat{\beta}_l$, and computes $\Delta_+^{\max}(e)$ and $\Delta_-^{\max}(e)$ using $\widehat{\alpha}_{l,e}$ and $\widehat{\beta}_{l,e}$. Following arguments analogous to that of Lemma 6.1, we can show:

Lemma 6.11. *For each l and $e \in V$, $\widehat{\alpha}_{l,e} \leq \alpha_l^{\iota(e)-1}$ and $\widehat{\beta}_{l,e} \geq \beta_l^{\iota(e)-1}$.*

Corollary 6.12. *Concavity of g implies that Δ's computed by Algorithm 6.10 satisfy*

$$\begin{aligned}
\Delta_+^{\max}(e) &\geq \sum_{S_l \ni e} \left[g(\alpha_l^{\iota(e)-1} + w_l(e)) - g(\alpha_l^{\iota(e)-1}) \right] - \lambda v(e) = \Delta_+(e), \\
\Delta_-^{\max}(e) &\geq \sum_{S_l \ni e} \left[g(\beta_l^{\iota(e)-1} - w_l(e)) - g(\beta_l^{\iota(e)-1}) \right] + \lambda v(e) = \Delta_-(e),
\end{aligned}$$

The analysis of Section 6.6 follows immediately from the above.

Algorithm 6.10: CF-2G for separable sums

```
1  for e ∈ V do Â(e) = 0
2
3  for l = 1, ..., L do α̂_l = 0, β̂_l = Σ_{e∈S_l} w_l(e)
4
5  for p ∈ {1, ..., P} do in parallel
6      while ∃ element to process do
7          e = next element to process
8          Δ+max(e) = −λv(e) + Σ_{S_l ∋ e} g(α̂_l + w_l(e)) − g(α̂_l)
9          Δ−max(e) = +λv(e) + Σ_{S_l ∋ e} g(β̂_l − w_l(e)) − g(β̂_l)
10         Draw u_e ∼ Unif(0, 1)
11         if u_e < [Δ+max(e)]+ / ([Δ+min(e)]+ + [Δ−max(e)]+) then
12             Â(e) ← 1
13             for l : e ∈ S_l do
14                 α̂_l ← α̂_l + w_l(e)
15         else
16             for l : e ∈ S_l do
17                 β̂_l ← β̂_l − w_l(e)
```

CC-2G for separable sums F

Analogous to the CF-2G algorithm, we maintain $\widehat{\alpha}_l$, $\widehat{\beta}_l$ and additionally $\widetilde{\alpha}_l = \sum_{j \in \widetilde{A} \cup S_l} w_l(j)$ and $\widetilde{\beta}_l = \sum_{j \in \widetilde{B} \cup S_l} w_l(j)$. Following the arguments of Lemma 6.3 and Corollary 6.5, we can show the following.

Lemma 6.13. $\widehat{\alpha}_{l,e} \leq \alpha^{\iota(e)-1} \leq \widetilde{\alpha}_{l,e} - w_l(e)$ *and* $\widehat{\beta}_{l,e} \geq \beta^{\iota(e)-1} \geq \widetilde{\beta}_{l,e} + w_l(e)$

Corollary 6.14. *Concavity of g implies that the Δ's computed by Algorithm 6.11 satisfy:*

$$\begin{aligned}
\Delta_+^{\max}(e) &= -\lambda v(e) + \sum_{S_l \ni e} [g(\widehat{\alpha}_{l,e} + w_l(e)) - g(\widehat{\alpha}_{l,e})] \\
&\geq -\lambda v(e) + \sum_{S_l \ni e} \left[g(\widehat{\alpha}_l^{\iota(e)-1} + w_l(e)) - g(\widehat{\alpha}_l^{\iota(e)-1}) \right] && = \Delta_+(e) \\
&\geq -\lambda v(e) + \sum_{S_l \ni e} [g(\widetilde{\alpha}_{l,e}) - g(\widetilde{\alpha}_{l,e} - w_l(e))] && = \Delta_+^{\min}(e), \\
\Delta_-^{\max}(e) &= \lambda v(e) + \sum_{S_l \ni e} \left[g(\widehat{\beta}_{l,e} - w_l(e)) - g(\widehat{\beta}_{l,e}) \right] \\
&\geq \lambda v(e) + \sum_{S_l \ni e} \left[g(\widehat{\beta}_l^{\iota(e)-1} - w_l(e)) - g(\widehat{\beta}_l^{\iota(e)-1}) \right] && = \Delta_-(e) \\
&\geq \lambda v(e) + \sum_{S_l \ni e} \left[g(\widetilde{\beta}_l^{\iota(e)-1}) - g(\widetilde{\beta}_l^{\iota(e)-1} + w_l(e)) \right] && = \Delta_-^{\min}(e).
\end{aligned}$$

The analysis of Section 6.6 and 6.6 follows immediately from the above.

Algorithm 6.11: CC-2G for separable sums

for $e \in V$ **do** $\widehat{A}(e) = \widetilde{A}(e) = 0$, $\widehat{B}(e) = \widetilde{B}(e) = 1$

for $l = 1, \ldots, L$ **do**
    $\widehat{\alpha}_l = \widetilde{\alpha}_l = 0$
    $\widehat{\beta}_l = \widetilde{\beta}_l = \sum_{e \in S_l} w_l(e)$

for $i = 1, \ldots, |V|$ **do** processed$(i) = false$

$\iota = 0$

for $p \in \{1, \ldots, P\}$ **do in parallel**
    while $\exists$ *element to process* **do**
        $e =$ next element to process
        $(\widehat{\alpha}_{\cdot,e}, \widetilde{\alpha}_{\cdot,e}, \widehat{\beta}_{\cdot,e}, \widetilde{\beta}_{\cdot,e}) =$ getGuarantee(e)
        $(\text{result}, u_e) =$ propose$(e, \widehat{\alpha}_{\cdot,e}, \widetilde{\alpha}_{\cdot,e}, \widehat{\beta}_{\cdot,e}, \widetilde{\beta}_{\cdot,e})$
        commit$(e, i, u_e, \text{result})$

Algorithm 6.12: CC-2G getGuarantee(e) for separable sums

$\widetilde{A}(e) \leftarrow 1$; $\widetilde{B}(e) \leftarrow 0$
for $l : e \in S_l$ **do**
  $\widetilde{\alpha}_l \leftarrow \widetilde{\alpha}_l + w_l(e)$
  $\widetilde{\beta}_l \leftarrow \widetilde{\beta}_l - w_l(e)$
$i = \iota$; $\iota \leftarrow \iota + 1$
$\widehat{\alpha}_{\cdot,e} = \widehat{\alpha}_{\cdot}$; $\widehat{\beta}_{\cdot,e} = \widehat{\beta}_{\cdot}$
$\widetilde{\alpha}_{\cdot,e} = \widetilde{\alpha}_{\cdot}$; $\widetilde{\beta}_{\cdot,e} = \widetilde{\beta}_{\cdot}$
return $(\widehat{\alpha}_{\cdot,e}, \widetilde{\alpha}_{\cdot,e}, \widehat{\beta}_{\cdot,e}, \widetilde{\beta}_{\cdot,e})$

Algorithm 6.13: CC-2G propose($e, \widehat{\alpha}_{\cdot,e}, \widetilde{\alpha}_{\cdot,e}, \widehat{\beta}_{\cdot,e}, \widetilde{\beta}_{\cdot,e}$) for separable sums

$\Delta_+^{\min}(e) = -\lambda v(e) + \sum_{S_l \ni e} g(\widetilde{\alpha}_l) - g(\widetilde{\alpha}_l - w_l(e))$
$\Delta_+^{\max}(e) = -\lambda v(e) + \sum_{S_l \ni e} g(\widehat{\alpha}_l + w_l(e)) - g(\widehat{\alpha}_l)$
$\Delta_-^{\min}(e) = +\lambda v(e) + \sum_{S_l \ni e} g(\widetilde{\beta}_l) - g(\widetilde{\beta}_l + w_l(e))$
$\Delta_-^{\max}(e) = +\lambda v(e) + \sum_{S_l \ni e} g(\widehat{\beta}_l - w_l(e)) - g(\widehat{\beta}_l)$
Draw $u_e \sim Unif(0,1)$
if $u_e < \frac{[\Delta_+^{\min}(e)]_+}{[\Delta_+^{\min}(e)]_+ + [\Delta_-^{\max}(e)]_+}$ **then** result $\leftarrow 1$
else if $u_e > \frac{[\Delta_+^{\max}(e)]_+}{[\Delta_+^{\max}(e)]_+ + [\Delta_-^{\min}(e)]_+}$ **then** result $\leftarrow -1$
else result $\leftarrow$ FAIL
return (result, u_e)

Algorithm 6.14: CC-2G commit(e, i, u_e,result) for separable sums

wait until $\forall j < i$, processed(j) = *true*
if *result* = *FAIL* **then**
  $\Delta_+^{\text{exact}}(e) = -\lambda v(e) + \sum_{S_l \ni e} g(\widehat{\alpha}_l + w_l(e)) - g(\widehat{\alpha}_l)$
  $\Delta_-^{\text{exact}}(e) = +\lambda v(e) + \sum_{S_l \ni e} g(\widehat{\beta}_l - w_l(e)) - g(\widehat{\beta}_l)$
  if $u_e < \frac{[\Delta_+^{exact}(e)]_+}{[\Delta_+^{exact}(e)]_+ + [\Delta_-^{exact}(e)]_+}$ **then** result$\leftarrow 1$
  else result$\leftarrow -1$
if *result*$= 1$ **then**
  $\widehat{A}(e) \leftarrow 1$; $\widetilde{B}(e) \leftarrow 1$
  for $l : e \in S_l$ **do**
    $\widehat{\alpha}_l \leftarrow \widehat{\alpha}_l + w_l(e)$
    $\widetilde{\beta}_l \leftarrow \widetilde{\beta}_l + w_l(e)$
else
  $\widetilde{A}(e) \leftarrow 0$; $\widehat{B}(e) \leftarrow 0$
  for $l : e \in S_l$ **do**
    $\widetilde{\alpha}_l \leftarrow \widetilde{\alpha}_l - w_l(e)$
    $\widehat{\beta}_l \leftarrow \widehat{\beta}_l - w_l(e)$
processed(i) = *true*

6.D Complete Experiment Results

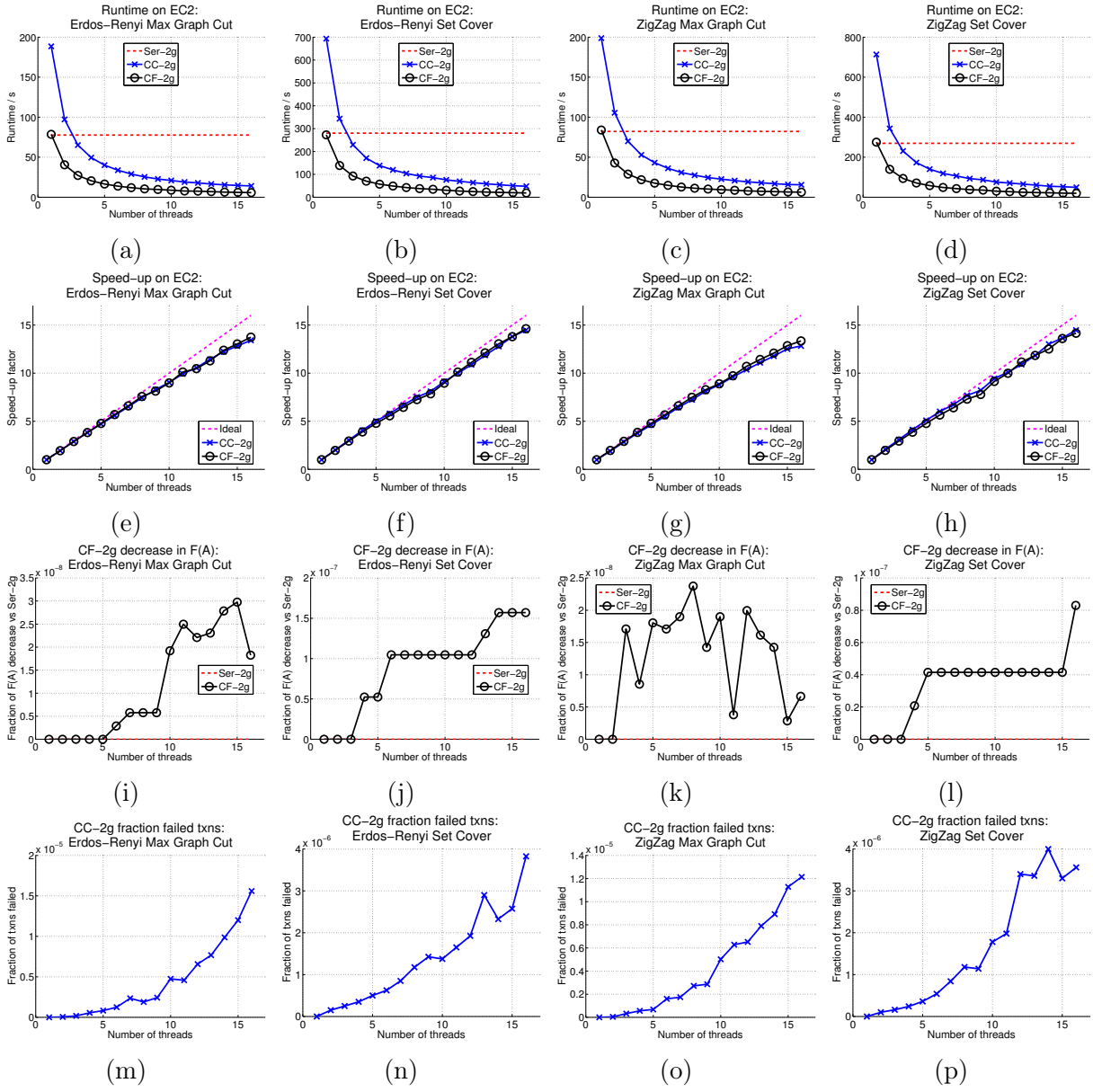


Figure 6.5: Experimental results on Erdos-Renyi and ZigZag synthetic graphs.

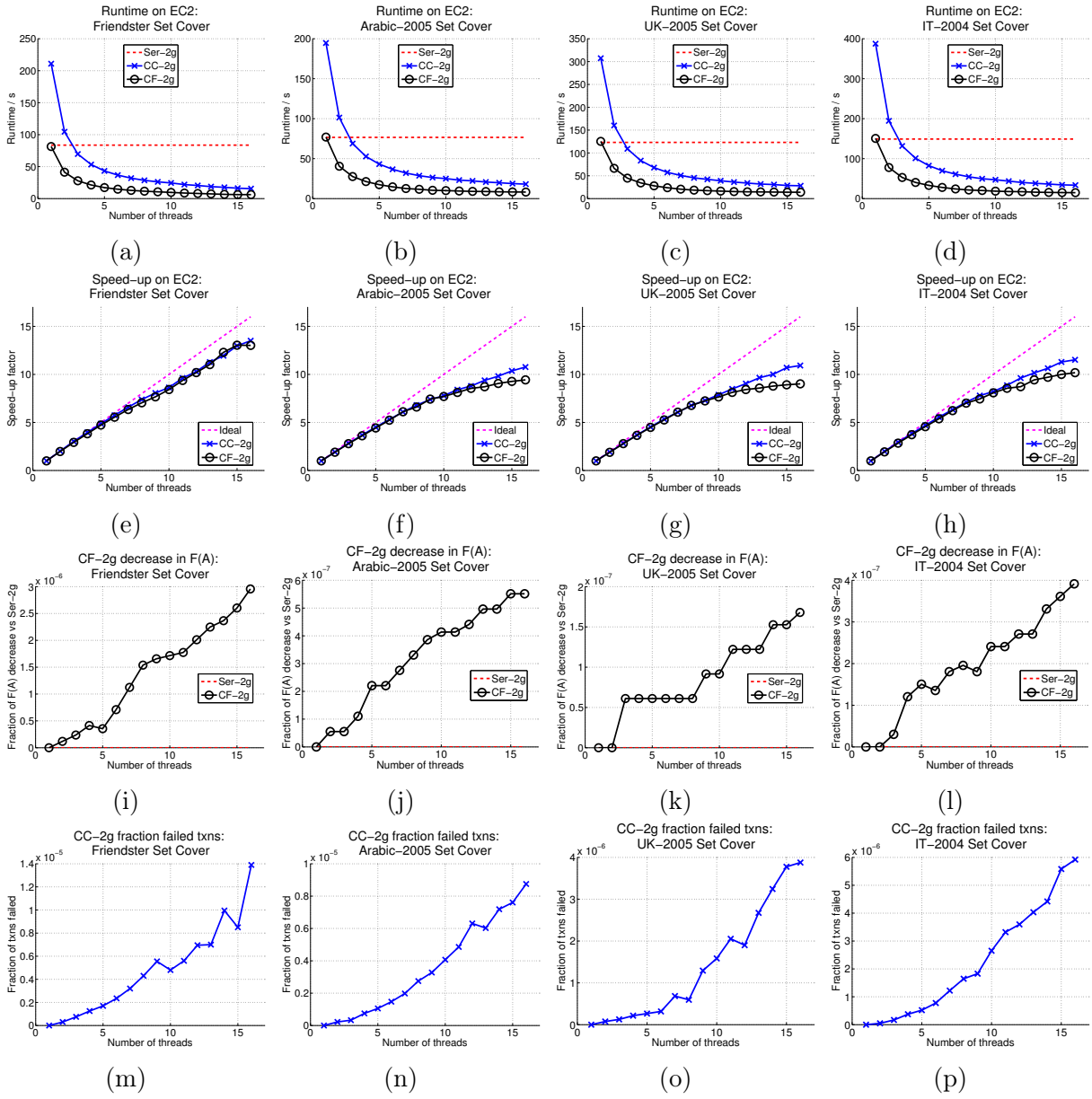


Figure 6.6: Set cover on 4 real graphs.

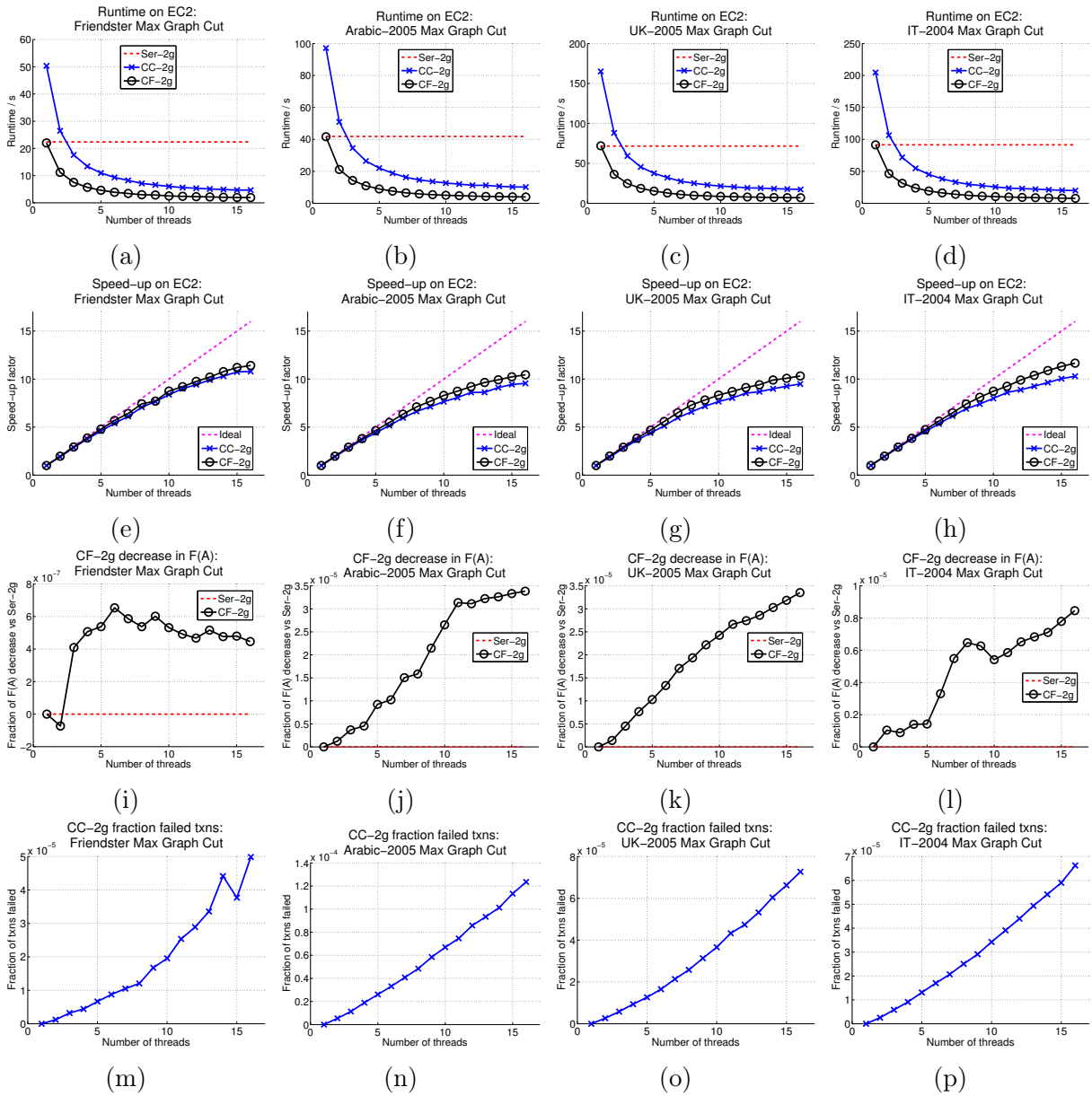


Figure 6.7: Max graph cut on 4 real graphs.

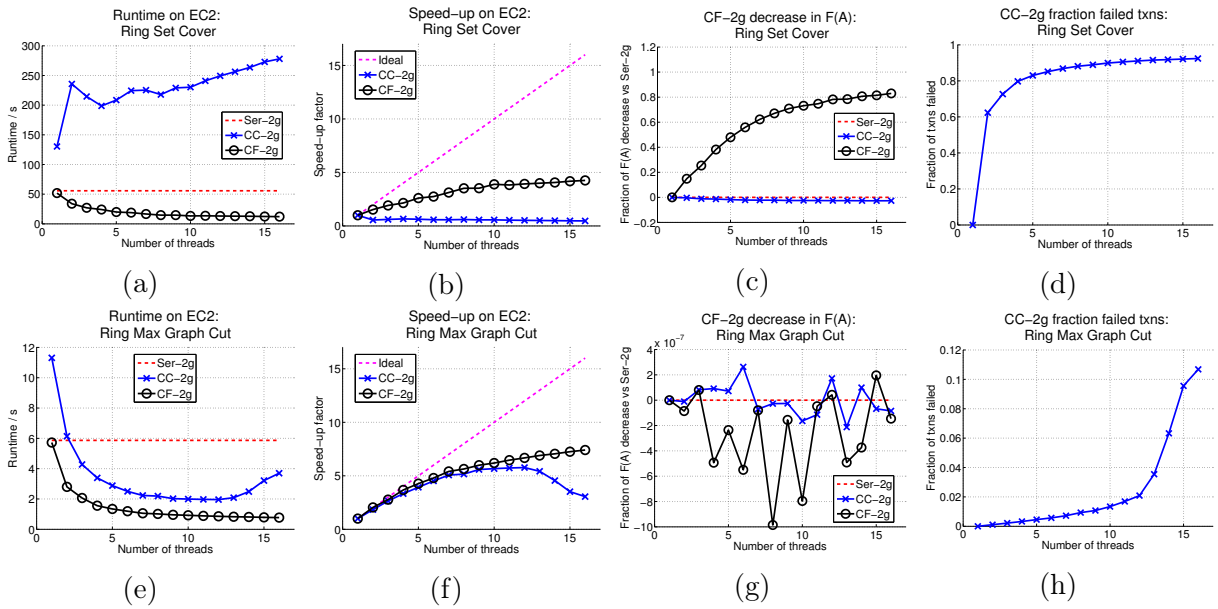


Figure 6.8: Experimental results for ring graph on set cover problem.

6.E Illustrative Examples

The following examples illustrate how (i) the simple (uni-directional) greedy algorithm may fail for non-monotone submodular functions, and (ii) where the coordination-free double greedy algorithm can run into trouble.

Greedy and non-monotone functions

For illustration, consider the following toy example of a non-monotone submodular function. We are given a ground set $V = \{v_0, v_1, v_2, \ldots, v_k\}$ of $k+1$ elements, and a universe $U = \{u_1, \ldots, u_k\}$. Each element v_i in V covers elements $\mathrm{Cov}(v_i) \subseteq U$ of the universe. In addition, each element in V has a cost $c(v_i)$. We are aiming to maximize the submodular function

$$F(S) = \Big| \bigcup_{v \in S} \mathrm{Cov}(v) \Big| - \sum_{v \in S} c(v). \tag{6.5}$$

Let the costs and coverings be as follows:

$$\mathrm{Cov}(v_0) = U \qquad c(v_0) = k - 1 \tag{6.6}$$

$$\mathrm{Cov}(v_i) = u_i \qquad c(v_i) = \epsilon < 1/k^2 \quad \text{for all } i > 0. \tag{6.7}$$

Then the optimal solution is $S^* = V \setminus v_0$ with $F(S^*) = k - k\epsilon$.

The greedy algorithm of [105] always adds the element with the largest marginal gain. Since $F(v_0) = 1$ and $F(v_i) = 1 - \epsilon$ for all $i > 0$, the algorithm would pick v_0 first. After

that, any additional element only has a negative marginal gain, $F(\{v_0, v_i\}) - F(v_0) = -\epsilon$. Hence, the algorithm would end up with a solution $F(v_0) = 1$ or worse, which means an approximation factor of only approximately $1/k$.

For the double greedy algorithm, the scenario would be the following. If v_0 happens to be the first element, then it is picked with probability

$$P(v_0) = \frac{[F(v_0) - F(\emptyset)]_+}{[F(v_0) - F(\emptyset)]_+ + [F(V \setminus v_0) - F(V)]_-} = \frac{1}{1 + (k-1)} = \frac{1}{k}. \tag{6.8}$$

If v_0 is selected, nothing else will be added afterwards, since $[F(v_0, v_i) - F(v_0)]_+ = 0$. If it does not pick v_0, then any other element is added with a probability of

$$P(v_i \mid \neg v_0) = \frac{[F(v_i) - F(\emptyset)]_+}{[F(v_i) - F(\emptyset)]_+ + F(V \setminus \{v_0, v_i\}) - F(V \setminus v_0)]_-} = \frac{1-\epsilon}{1-\epsilon} = 1. \tag{6.9}$$

If v_0 is not the first element, then any element before v_0 is added with probability $p(v_i) = 1 - \epsilon$, and as soon as an element v_i has been picked, v_0 will not be added any more. Hence, with high probability, this algorithm returns the optimal solution. The deterministic version surely does.

Coordination vs no coordination

The following example illustrates the differences between coordination and no coordination. In this example, let V be split into m disjoint groups G_j of equal size $k = |V|/m$, and let

$$F(S) = \sum_{j=1}^{m} \min\{1, |S \cap G_j|\} - \frac{|S \cap G_j|}{k}. \tag{6.10}$$

A maximizing set S^* contains one element from each group, and $F(S^*) = m - m/k$.

If the sequential double greedy algorithm has not picked an element from a group, it will retain the next element from that group with probability

$$\frac{1 - 1/k}{1 - 1/k + 1/k} = 1 - 1/k. \tag{6.11}$$

Once it has sampled an element from a group G_j, it does not pick any more elements from G_j, and therefore $|S \cap G_j| \leq 1$ for all j and the set S returned by the algorithm. The probability that S does not contain any element from G_j is k^{-k} —fairly low. Hence, with probability $1 - m/k^k$ the algorithm returns the optimal solution.

Without coordination, the outcome heavily depends on the order of the elements. For simplicity, assume that k is a multiple of the number q of processors (or q is a multiple of k). In the worst case, the elements are sorted by their groups and the members of each group are processed in parallel. With q processors working in parallel, the first q elements from a group G (up to shifts) will be processed with a bound $\widehat{A}$ that does not contain any

element from G, and will each be selected with probability $1 - 1/k$. Hence, in expectation, $|S \cap G_j| = \min\{q, k\}(1 - 1/k)$ for all j.

If $q > k$, then in expectation $k-1$ elements from each group are selected, which corresponds to an approximation factor of

$$\frac{m(1 - \frac{k-1}{k})}{m(1 - 1/k)} = \frac{1}{k-1}. \tag{6.12}$$

If $k > q$, then in expectation we obtain an approximation factor of

$$\frac{m(1 - \frac{q(1-1/k)}{k})}{m(1 - 1/k)} = 1 - \frac{q}{k} + \frac{1}{k-1} \tag{6.13}$$

which decreases linearly in q. If $q = k$, then the factor is $1/(q-1)$ instead of $1/2$.

Chapter 7

Sparse Stochastic Updates

7.1 Introduction

In this chapter[1], we focus on the parallelization of sparse stochastic update algorithms (Section 7.A), which includes a large class of popular optimization algorithms such as SGD, SVRG, and SAGA.

Following the seminal work of HOGWILD! [116], many studies have demonstrated that near-linear speedups are achievable on a variety of machine learning tasks via asynchronous, lock-free, coordination-free implementations [117, 143, 138, 94, 50, 133, 67, 101]. In all of these studies, classic algorithms are parallelized by simply running parallel and asynchronous model updates without locks nor coordination. These lock-free, asynchronous algorithms exhibit speedups even when applied to large, non-convex problems, as demonstrated by deep learning systems such as Google's Downpour SGD [44] and TensorFlow [1], and Microsoft's Project Adam [38].

While these techniques have been remarkably successful, many of the papers cited above require delicate and tailored analyses to understand the benefits of asynchrony for the particular learning task at hand. Moreover, in non-convex settings, we currently have little quantitative insight into how much speedup is gained from asynchrony and how much accuracy may be lost.

We present CYCLADES, a general framework for lock-free, asynchronous machine learning algorithms that obviates the need for specialized analyses. CYCLADES runs updates asynchronously and *maintains serializability*, i.e., it produces an outcome equivalent to a serial execution of the updates. Furthermore, CYCLADES is *deterministic* — given the serial algorithm's ordering of updates, CYCLADES is able to produce the same output as the serial algorithm. Hence, any algorithm parallelized by our framework inherits the correctness proof of the serial counterpart without modifications. Additionally, if a particular heuristic serial algorithm is popular, but does not have a rigorous analysis, such as backpropagation on

[1]Work done as part of [108].

neural networks, CYCLADES still guarantees that its execution will return a serially equivalent output.

CYCLADES achieves serializability and determinism by using BSP and scheduling to temporally and spatially separate conflicting updates. Within each BSP batch, CYCLADES partitions updates among cores in a way that ensures that there are no conflicts between cores. Such a partition can always be found efficiently by leveraging a powerful result on graph phase transitions [82]. When applied to our setting, this result guarantees that a sufficiently small BSP batch of updates will have only a *logarithmic* number of conflicts. This allows us to evenly partition model updates within the batch across cores, with the guarantee that all conflicts are spatially localized within each core. The global BSP synchronization barriers prevents conflicts across batches, and enables CYCLADES to obtain good intra-batch load-balancing.

Given enough problem sparsity, CYCLADES guarantees a nearly linear speedup, while inheriting all the qualitative properties of the serial counterpart of the algorithm, e.g., proofs for rates of convergence. Enforcing a deterministic execution in CYCLADES comes with additional practical benefits. Determinism is helpful for hyperparameter tuning, or locating the best model produced by the asynchronous execution, since experiments are reproducible, and solutions are easily verifiable. Moreover, a CYCLADES program is easy to debug, because bugs are repeatable and we can examine the step-wise execution to localize them.

A significant benefit of the update partitioning in CYCLADES is that it induces considerable access locality compared to the more unstructured nature of the memory accesses during HOGWILD!. Cores will access the same data points and read/write the same subset of model variables. This has the additional benefit of reducing false sharing across cores. Because of these gains, CYCLADES can actually *outperform* HOGWILD! in practice on sufficiently sparse problems, despite appearing to require more computational overhead. Remarkably, because of the added locality, even a single threaded implementation of CYCLADES can actually be faster than serial SGD. In our SGD experiments for matrix completion and word embedding problems, CYCLADES can offer a speedup gain of up to 40% compared to that of HOGWILD!. Furthermore, for variance reduction techniques such as SAGA [45] and SVRG [74], CYCLADES yields better accuracy and more significant speedups, with up to 5× performance gains over HOGWILD!-type implementations.

The remainder of our paper is organized as follows. Section 7.2 establishes some preliminaries. Details and theory of CYCLADES are presented in Section 7.3. We present our experiments in Section 7.4, we discuss related work in Section 7.5, and then conclude with Section 7.6.

7.2 The Algorithmic Family of Stochastic-Updates

We study parallel asynchronous iterative algorithms on the computational model used by [116], and similar to the partially asynchronous model of [18]: a number of cores have access to the

same shared memory, and each of them can read and update components of the model x in parallel from the shared memory.

In this chapter, we consider a large family of randomized algorithms that we will refer to as Stochastic Updates (SU). The main algorithmic component of SU focuses on updating small subsets of a model variable x, that lives in shared memory, according to prefixed access patterns, as sketched by Algorithm 7.1.

Algorithm 7.1: Stochastic Updates pseudo-algorithm

Input: $x; f_1, \ldots, f_n; u_1, \ldots, u_n; \mathcal{D}; T$.

1 **for** $t = 1 : T$ **do**

2 Sample $i \sim \mathcal{D}$

3 //Update global model on $\mathcal{S}_i$

4 $x_{\mathcal{S}_i} = u_i(x_{\mathcal{S}_i}, f_i)$

Output: x

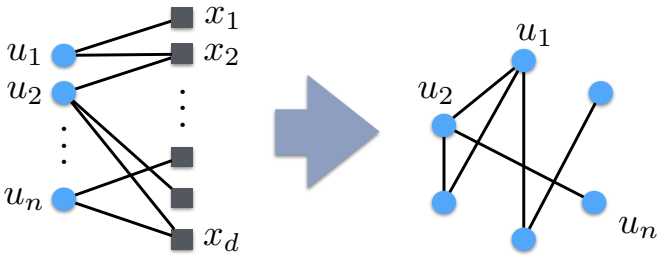


Figure 7.1: The above bipartite graph G_u (left) links an update u_i to a variable x_j when an update needs to access (read or write) the variable. From G_u we obtain the conflict graph G_c (right), whose max degree is Δ. If G_c is sufficiently sparse, we expect that it is possible to parallelize updates without too many conflicts. CYCLADES exploits this intuition.

In Algorithm 7.1 each set $\mathcal{S}_i$ is a subset of the coordinate indices of x, and each function f_i only operates on the subset $\mathcal{S}_i$ of coordinates, i.e., both its domain (read set) and co-domain (write set) are inside $\mathcal{S}_i$), and u_i is a local update function that computes a vector with support on $\mathcal{S}_i$ using as input $x_{\mathcal{S}_i}$ and f_i. Moreover, T is the total number of iterations, and $\mathcal{D}$ is the distribution with support $\{1, \ldots, n\}$ from which we draw i. As we explain in Appendix 7.A, several machine learning and optimization algorithms belong to the SU algorithmic family, such as stochastic gradient descent (SGD), with or without weight decay and regularization, variance-reduced learning algorithms like SAGA and SVRG, and even some combinatorial graph algorithms.

The Stochastic Updates algorithm is also a specialization of the iterative transformation framework to transactions with the same read and write sets, i.e., $\mathcal{S}_i = R(f_i) = W(f_i)$. Specifically, using the change function $\Lambda_i(x_{\mathcal{S}_i}, f_i) = u_i(x^{(t)}_{\mathcal{S}_i}, f_i)$ and transformation function $T_{i,k}(x, \lambda_{\mathcal{S}_i}) = \mathbf{1}_{\{k \notin \mathcal{S}_i\}} x_k + \mathbf{1}_{\{k \in \mathcal{S}_i\}} \lambda_k$, we can rewrite (2.3) as

$$x_k^{(t+1)} = \begin{cases} T_{i,k}(x^{(t)}, \Lambda_i(x^{(t)}_{\mathcal{S}_i}, f_i)) & \text{if } k \in \mathcal{S}_i \\ x_k^{(t)} & \text{otherwise} \end{cases} \tag{7.1}$$

$$= \begin{cases} [u_i(x^{(t)}_{\mathcal{S}_i}, f_i))]_k & \text{if } k \in \mathcal{S}_i \\ x_k^{(t)} & \text{otherwise} \end{cases} \tag{7.2}$$

The Updates Conflict Graph A useful construction for our developments is the conflict graph between updates, which can be generated from the bipartite graph between the updates and the model variables. We define these graphs below, and provide an illustrative sketch in Figure 7.1.

Definition 7.1. *We denote as G_u the bipartite update-variable graph between the updates $u_1, \ldots, u_n$ and the d model variables. In G_u an update u_i is linked to a variable x_j, if u_i requires to read or write x_j. We let E_u denote the number of edges in the bipartite graph, and also denote as Δ_L the left max vertex degree of G_u, and as $\overline{\Delta}_L$ its average left degree.*

Definition 7.2. *We denote by G_c a conflict graph on n vertices, each corresponding to an update u_i. Two vertices of G_c are linked with an edge, if and only if the corresponding updates share at least one variable in the bipartite-update graph G_u. We also denote as Δ the max vertex degree of G_c.*

We stress that the conflict graph is never actually constructed, but serves as a useful concept for understanding CYCLADES.

Our Main Result By exploiting the structure of the above graphs and through a light-weight and careful sampling and allocation of updates, CYCLADES is able to guarantee the following result for SU algorithms, which we establish in the following sections.

Theorem 7.1 (informal)**.** *Let us consider an SU algorithm $\mathcal{A}$ defined through n update rules, where the conflict max degree between the n updates is Δ, and the sampling distribution $\mathcal{D}$ is uniform with (or without) replacement from $\{1, \ldots, n\}$. Moreover, assume that we wish to run $\mathcal{A}$ for $T = \Theta(n)$ iterations, and that $\frac{\Delta_L}{\overline{\Delta}_L} \leq \sqrt{n}$. Then on up to $P = \tilde{O}(\frac{n}{\Delta \cdot \Delta_L})$ cores,* CYCLADES *guarantees a $\widetilde{\Omega}(P)$ speedup over $\mathcal{A}$, while outputting the same solution x as $\mathcal{A}$ would do after the same random set of T iterations.*[2]

We will now provide some simple examples of how the above parameters, and guarantees translate for specific problem cases.

Many machine learning applications often seek to minimize the empirical risk

$$\min_x \frac{1}{n} \sum_{i=1}^{n} \ell_i(a_i^T x)$$

where a_i represents the ith data point, x is the model we are trying to fit, and ℓ_i is a loss function that tells us how good of a fit a model is with respect to data point i. Several problems can be formulated in the above way, such as logistic regression, least squares, support vector machines (SVMs) for binary classification, and others. If we attempt to solve the above problem using SGD (with or without regularization), or via variance reduction techniques

[2] $\widetilde{\Omega}(\cdot)$ and $\widetilde{O}(\cdot)$ hide polylog factors.

like SVRG and SAGA, then (as we show in Appendix 7.A) the sparsity of the u_i updates is determined by the gradient of a single sampled data point i. For the aforementioned problems, this will be proportional to $\left(\frac{d}{du}\ell_i(u)\big|_{u=a_i^T x}\right) a_i$, hence the sparsity of the update is defined by the non-zero support of datapoint a_i. In the induced bipartite update-variable graph of this problem, we have $\Delta_L = \max_i ||a_i||_0$, and the maximum conflict degree Δ is the maximum number of data points a_i that share at least one of the d features. As a toy example, let $\frac{n}{d} = \Theta(1)$ and let the non-zero support of a_i be of size n^δ and uniformly distributed. Then, one can show that with overwhelmingly high probability $\Delta = \widetilde{O}(n^{1/2+\delta})$ and hence CYCLADES achieves an $\widetilde{\Omega}(P)$ speedup on up to $P = \widetilde{O}(n^{1/2-2\delta})$ cores.

Consider the following generic minimization problem

$$\min_{x_1,\ldots,x_{m_1}} \min_{y_1,\ldots,y_{m_2}} \sum_{i=1}^{m_1}\sum_{j=1}^{m_2} \phi_{i,j}(x_i, y_j)$$

where $\phi_{i,j}$ is a convex function of a scalar. The above generic formulation captures several problems like matrix completion and matrix factorization [115] (where $\phi_{i,j} = (A_{i,j} - x_i^T y_j)^2$), word embeddings [9] (where $\phi_{i,j} = A_{i,j}(\log(A_{i,j}) - \|x_i + x_j\|_2^2 - C)^2$), graph k-way cuts [116] (where $\phi_{i,j} = A_{i,j}\|x_i - x_j\|_1$), and others. Let $m_1 = m_2 = m$ for simplicity, and assume that we aim to minimize the above by sampling a single function $\phi_{i,j}$ and then updating x_i and y_j using SGD. Here, the number of update functions is proportional to $n = m^2$, and for the above setup each gradient update with respect to the sampled function $\phi_{i,j}(x_i, y_j)$ is only interacting with the variables x_i and y_j, i.e., only two variable vectors out of the $2m$ many (i.e., $\Delta_L = 2$). Moreover, the previous imply a conflict degree of at most $\Delta = 2m$. In this case, CYCLADES can provably guarantee an $\widetilde{\Omega}(P)$ speedup for up to $P = O(m)$ cores.

In our experiments we test CYCLADES on several problems including least squares, classification with logistic models, matrix factorization, and word embeddings, and several algorithms including SGD, SVRG, and SAGA. We show that in most cases it can significantly outperform the HOGWILD! implementation of these algorithms, if the data is sparse.

We would like to note, that there are several cases where there might be a few outlier updates with extremely high conflict degree. In Appendix 7.E, we prove that if there are no more than $O(n^\delta)$ vertices of high conflict degree Δ_o, and the rest of the vertices have max degree at most Δ, then the result of Theorem 7.1 still holds in expectation.

In the following section, we establish the technical results behind CYCLADES and provide the details behind our parallelization framework.

7.3 CYCLADES: Shattering Dependencies

CYCLADES consists of three computational components as shown in Figure 7.2.

It starts by sampling (according to a distribution $\mathcal{D}$) a number of B updates from the graph shown in Figure 7.1, and assigns a label to each of them (a processing order). We note that in practice the sampling is done on the bipartite graph, which avoids the need to

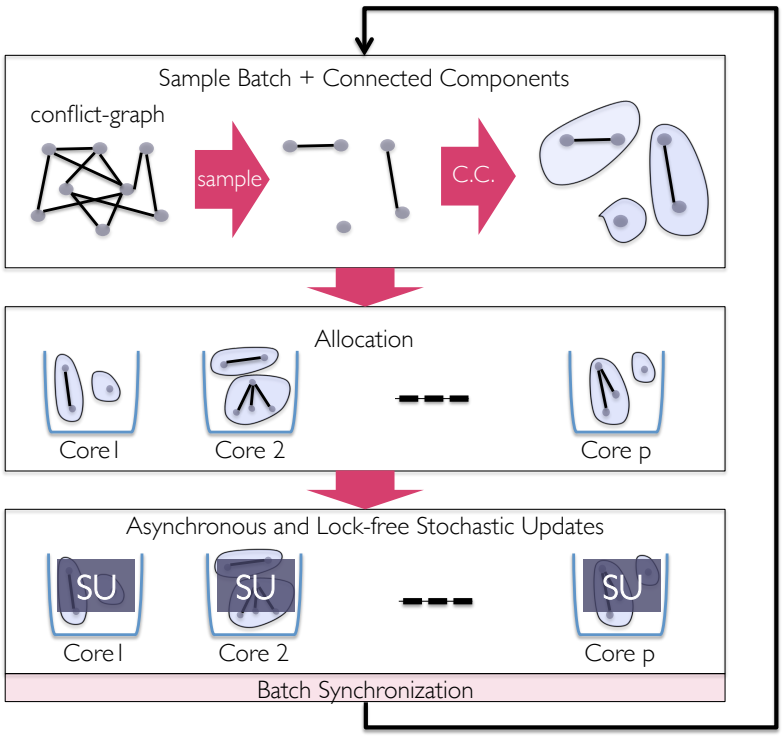


Figure 7.2: CYCLADES carefully samples updates, then finds conflict-groups, and allocates them across cores. Then, each core asynchronously updates the shared model, without incurring any read/write conflicts. This is possible by processing all the conflicting updates within the same core. After the processing of a batch is completed, the above is repeated, for as many iterations as required.

actually construct the conflict graph. After sampling, it computes the connected components of the sampled subgraph induced by the B sampled updates, to determine the conflict groups. Once the conflicts groups are formed, it allocates them across P cores. Finally, each core processes locally the conflict groups of updates that it has been assigned, following the order that each update has been labeled with. The above process is then repeated, for as many iterations as needed.

The key component of CYCLADES is to carry out the sampling in such a way that we have as many connected components as possible, and all of them of small size, provably. In the next subsections, we explain how each part is carried out, and provide theoretical guarantees for each of them individually, which we combine at the end of this section for our main theorem.

Frugal sampling shatters conflicts. A key technical aspect that we exploit in CYCLADES is that appropriate sampling and allocation of updates can lead to near optimal parallelization of sparse SU algorithms. To do that we expand upon the following result established in [82].

Theorem 7.2. *Let G be a 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 sampled vertices. Then, the largest connected component of G' has size at most $\frac{4}{\epsilon^2}\log n$, with high probability.*

The above result pays homage to the giant component phase transition phenomena in random Erdos-Renyi graphs. What is surprising is that a similar phase transition can apply for any given graph!

Adapting to ML-friendly sampling procedures. In practice, for most SU algorithms of interest, the sampling distribution of updates is either with or without replacement from the n updates. As it turns out, morphing Theorem 7.2 into a with-/without-replacement result is not straightforward. We defer the analysis needed to Appendix 7.B, and present our main theorem about graph sampling here.

Theorem 7.3. *Let G be a graph on n vertices, with maximum vertex degree Δ. Let us sample $B = (1-\epsilon)\frac{n}{\Delta}$ vertices with or without replacement, and define as G' the induced subgraph on the sampled vertices. Then, the largest connected component of G' has size at most $O(\frac{\log n}{\epsilon^2})$, with high probability.*

The key idea from the above theorem is that if one samples no more than $B = (1-\epsilon)\frac{n}{\Delta}$ vertices, then there will be at least $O\left(\epsilon^2 B/\log n\right)$ conflict groups to allocate across cores, all of size at most $O\left(\log n/\epsilon^2\right)$. Moreover, since there are no conflicts between different conflict-groups, the processing of updates per any single group will never interact with the variables corresponding to the updates of another conflict group.

The next step of CYCLADES is to form and allocate the connected components (CCs) across cores, and do so efficiently. We address this in the following subsection. In the following, for simplicity we carry our analysis for the with-replacement sampling case, but it can be readily extended to the without-replacement sampling case.

Identifying Groups of Conflict via CCs In CYCLADES, we sample batches of updates of size $B = (1-\epsilon)\frac{n}{\Delta}$ multiple times, and for each batch we need to identify the conflict groups across the updates. Let us refer to G_u^i as the subgraph induced by the ith sampled batch of updates on the update-variable bipartite graph G_u. In the following we always assume that we sample at most $n_b = c \cdot \frac{\Delta}{1-\epsilon}$ batches, where $c \geq 1$ is a constant that does not depend on n. This number of batches results in a constant number of passes over the dataset.

Identifying the conflict groups in G_u^i can be done with a connected components (CC) algorithm. The main question we need to address is what is the best way to parallelize this graph partitioning part. There are two avenues that we can take for this, depending on the number of cores P at our disposal. We can either parallelize the computation of the CCs of a single batch (i.e., compute the CCs of G_u^i on P cores), or we can compute in parallel the CCs of all n_b batches, by allocating the sampled graphs G_u^i to cores, so that each of them can

compute the CCs of its allocated subgraphs. Depending on the number of available cores, one technique can be better than the other. In Appendix 7.C we provide the details of this part, and prove the following result:

Lemma 7.4. *Let the number of cores by bounded as $P = O(\frac{n}{\Delta\Delta_L})$, and let $\frac{\Delta_L}{\bar{\Delta}_L} \leq \sqrt{n}$. Then, the overall computation of CCs for $n_b = c \cdot \frac{\Delta}{1-\epsilon}$ batches, each of size $B = (1-\epsilon)\frac{n}{\Delta}$, costs no more than $O(\frac{E_u \log^2 n}{P})$.*

Allocating Updates to Cores Once we compute the CCs (i.e., the conflicts groups of the sampled updates), we have to allocate them across cores. Once a core has been assigned with CCs, it will process the updates included in these CCs, according to the order that each update has been labeled with. Due to Theorem 7.3, each connected component will contain at most $O(\frac{\log n}{\epsilon^2})$ updates. Assuming that the cost of the j-th update in the batch is w_j, the cost of a single connected component $\mathcal{C}$ will be $w_{\mathcal{C}} = \sum_{j \in \mathcal{C}} w_j$. To proceed with characterizing the maximum load among the P cores, we assume that the cost of a single update u_i, for $i \in \{1, \ldots, n\}$, is proportional to the out-degree of that update —according to the update-variable graph G_u— times a constant cost which we shall refer to as κ. Hence, $w_j = O(d_{L,j} \cdot \kappa)$, where $d_{L,j}$ is the degree of the j-th left vertex of G_u. In Appendix 7.D we establish that a near-uniform allocation of CCs according to their weights leads to the following guarantee.

Lemma 7.5. *Let the number of cores by bounded as $P = O(\frac{n}{\Delta\Delta_L})$, and let $\frac{\Delta_L}{\bar{\Delta}_L} \leq \sqrt{n}$. Then, computing the stochastic updates across all $n_b = c \cdot \frac{\Delta}{1-\epsilon}$ batches can be performed in time $O(\frac{E \log^2 n}{P} \cdot \kappa)$, with high probability, where κ is the per edge cost for computing one of the n updates defined on G_u.*

Stitching the pieces together Now that we have described the sampling, conflict computation, and allocation strategies, we are ready to put all the pieces together and detail CYCLADES in full. Let us assume that we sample a total number of $n_b = c \cdot \frac{\Delta}{1-\epsilon}$ batches of size $B = (1-\epsilon)\frac{n}{\Delta}$, and that each update is sampled uniformly at random. For the i-th batch let us denote as $\mathcal{C}_1^i, \ldots \mathcal{C}_{m_i}^i$ the connected components on the induced subgraph G_u^i. Due to Theorem 7.3, each connected component $\mathcal{C}$ contains a number of at most $O(\frac{\log n}{\epsilon^2})$ updates, and each update carries an ID (the order of which it would have been sampled by the serial algorithm). Using the above notation, we give the pseudocode for CYCLADES in Algorithm 7.2.

Note that the inner loop that is parallelized (i.e., the SU processing loop in lines 6 – 9), can be performed asynchronously; cores do not have to synchronize, and do not need to lock any memory variables, as they are all accessing non-overlapping subset of x. This also provides for better cache coherence. Moreover, each core potentially accesses the same coordinates several times, leading to good cache locality. These improved cache locality and

Algorithm 7.2: CYCLADES

```
Input: G_u, T, B.
Sample n_b = T/B subgraphs G_u^1, ..., G_u^{n_b} from G_u
Cores compute in parallel CCs for sampled subgraphs
for batch i = 1 : n_b do
    Allocation of C_1^i, ... C_{m_i}^i to P cores
    for each core in parallel do
        for each allocated component C do
            for each update j (in order) from C do
                x_{S_j} = u_j(x_{S_j}, f_j)
Output: x
```

coherence properties experimentally lead to substantial performance gains as we see in the next section.

We can now combine the results of the previous subsection to obtain our main theorem for CYCLADES.

Theorem 7.6. *Let us assume any given update-variable graph G_u with average, and max left degree $\overline{\Delta}_L$ and Δ_L, such that $\frac{\Delta_L}{\overline{\Delta}_L} \leq \sqrt{n}$, and with induced max conflict degree Δ. Then,* CYCLADES *on $P = O(\frac{n}{\Delta \cdot \Delta_L})$ cores, with batch sizes $B = (1-\epsilon)\frac{n}{\Delta}$ can execute $T = c \cdot n$ updates, for any constant $c \geq 1$, selected uniformly at random with replacement, in time*

$$\mathcal{O}\left(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n\right),$$

with high probability.

Observe that CYCLADES bypasses the need to establish convergence guarantees for the parallel algorithm. Hence, it could be the case for many applications of interest that although we might not be able to analyze how "well" the serial SU algorithm might perform in terms of the accuracy of the solution, CYCLADES can provide black box guarantees for speedup, since our analysis is completely oblivious to the qualitative performance of the serial algorithm. This is in contrast to recent studies similar to [41], where the authors provide speedup guarantees via a convergence-to-optimal proof for an asynchronous SGD on a nonconvex problem. Unfortunately these proofs can become complicated especially on a wider range of nonconvex objectives.

In the following section we show that CYCLADES is not only useful theoretically, but can consistently outperform HOGWILD! on sufficiently sparse datasets.

7.4 Evaluation

Implementation and Setup

We implemented CYCLADES in `C++` and tested it on a variety of problems and datasets described below. We tested a number of stochastic updates algorithms, and compared against their HOGWILD! (i.e., asynchronous and lock-free) implementations — in some cases, there are no theoretical foundations for these HOGWILD! implementations, even if they work reasonably well in practice. Since CYCLADES is intended to be a general approach for parallelization of stochastic updates algorithms, we do not compare against algorithms designed and tailored for specific applications, nor do we expect CYCLADES to outperform every such highly-tuned, well-designed, specific algorithm.

Our experiments were conducted on a machine with 72 CPUs (Intel(R) Xeon(R) CPU E7-8870 v3, 2.10 GHz) on 4 NUMA nodes, each with 18 CPUs, and 1TB of memory. We ran both CYCLADES and HOGWILD! with 1, 4, 8, 16 and 18 threads pinned to CPUs on a single NUMA node (i.e., the maximum physical number of cores possible, for a single node), so that we can avoid well-known cache coherence and scaling issues across different nodes [140]. We note that distributing threads across NUMA nodes significantly increased running times for both CYCLADES and HOGWILD!, but was relatively worse for HOGWILD!. We believe this is due to the poorer locality of HOGWILD!, which results in more cross-node communication. In this paper, we exclusively focus our study and experiments on parallelization within a single NUMA node, and leave cross-NUMA node parallelization for future work, while referring the interested reader to a recent study of the various tradeoffs of ML algorithms on NUMA aware architectures [140].

In our experiments, we measure overall running times which include the overheads for computing connected components and allocating work in CYCLADES. Separately, we also measure running times for performing the stochastic updates by excluding the CYCLADES coordination overheads. We also compute the objective value at the end of each epoch (i.e., one full pass over the data). We measure the speedups for each algorithm as

$$\frac{\text{time of the parallel algorithm to reach } \epsilon \text{ objective}}{\text{time of the serial algorithm to reach } \epsilon \text{ objective}}$$

where ϵ was chosen to be the smallest objective value that is achievable by all parallel algorithms on every choice of number of threads. That is, $\epsilon = \max_{\mathcal{A},T} \min_e f(X_{\mathcal{A},T,e})$ where $X_{\mathcal{A},T,e}$ is the model learned by algorithm $\mathcal{A}$ on T threads after e epochs. The serial algorithm used for comparison is HOGWILD! running serially on one thread.

In Table 7.1 we list some details of the datasets that we use in our experiments. The stepsizes and batch sizes used for each problem are listed in Table 7.2, along with dataset and problem details. In general, we chose the stepsizes to maximize convergence without diverging. Batch sizes were picked to optimize performance for CYCLADES.

Dataset	# datapoints	# features	Density (average number of features per datapoint)	Comments
NH2010	48,838	48,838	4.8026	Topological graph of 49 Census Blocks in New Hampshire.
DBLP	5,425,964	5,425,964	3.1880	Authorship network of 1.4M authors and 4M publications, with 8.65M edges.
MovieLens	∼10M	82,250	200	10M movie ratings from 71,568 users for 10,682 movies.
EN-Wiki	20,207,156	213,272	200	Subset of English Wikipedia dump.

Table 7.1: Details of datasets used in our experiments.

Problem	Algorithm	Dataset	Hogwild! Stepsize	Cyclades Stepsize	Batch Size	Average # of connected components	Average size of connected components
Least squares	SAGA	NH2010	1×10^{-14}	3×10^{-14}	1,000	792.98	1.257
		DBLP	1×10^{-5}	3×10^{-4}	10,000	9410.34	1.062
Graph eigen	SVRG	NH2010	1×10^{-5}	1×10^{-1}	1,000	792.98	1.257
		DBLP	1×10^{-7}	1×10^{-2}	10,000	9410.34	1.062
Matrix comp	SGD	MovieLens	5×10^{-5}		5,000	1663.73	3.004
	Weighted SGD						
Word embed	SGD	EN-Wiki	1×10^{-10}		4,250	2571.51	1.653

Table 7.2: Stepsizes and batch sizes for the various learning tasks in our evaluation. We selected stepsizes that maximize convergence without diverging. We also chose batch sizes to maximize performance of Cyclades. We further list the average size of connected components and the average number of connected components in each batch. Typically there are many connected components with small average size, which leads to good load balancing for Cyclades.

Learning tasks and algorithmic setup

Least squares via SAGA The first problem we consider is least squares:

$$\min_x \frac{1}{n}\|Ax - b\|_2^2 = \min_x \frac{1}{n}\sum_{i=1}^{n}(a_i^T x - b_i)^2$$

which we will solve using the SAGA algorithm [45], an incremental gradient algorithm with faster than SGD rates on convex, or strongly convex functions. In SAGA, we initialize $g_i = \nabla f_i(x_0)$ and iterate the following two steps

$$x_{k+1} = x_k - \gamma \cdot \left(\nabla f_{s_k}(x_k) - g_{s_k} + \frac{1}{n}\sum_{i=1}^{n} g_i\right)$$
$$g_{s_k} = \nabla f_{s_k}(x_k).$$

where $f_i(x) = (a_i^T x - b_i)^2$ and $\nabla f_i(x) = 2\left(a_i^T x - b_i\right) a_i$. In the above iteration it is useful to observe that the updates can be performed in a sparse and "lazy" way. That is for any

updates where the sampled gradients ∇f_{s_k} have non-overlapping support, we can still run them in parallel, and apply the vector of gradient sums at the end of a batch "lazily". We explain the details of the lazy updates in Appendix 7.A. This requires computing the number τ_j of skipped gradient sum updates for each lazily updated coordinate j, which may be negative in HOGWILD! due to re-ordering of updates. We thresholded τ_j when needed in the HOGWILD! implementation, as this produced better convergence for HOGWILD!. Unlike other experiments, we used different stepsizes γ for CYCLADES and HOGWILD!, as HOGWILD! would often diverge with larger stepsizes. The stepsizes chosen for each were the largest such that the algorithms did not diverge. We used the DBLP and NH2010 datasets for this experiment, and set A as the adjacency matrix of each graph. For NH2010, the values of b were set to population living in the Census Block. For DBLP we used synthetic values: we set $b = A\tilde{x} + 0.1\tilde{z}$, where $\tilde{x}$ and $\tilde{z}$ were generated randomly. The SAGA algorithm was run for up to 500 epochs for each dataset.

Graph eigenvector via SVRG Given an adjacency matrix A, the top eigenvector of A^TA is useful in several applications such as spectral clustering, principle component analysis, and others. In a recent work, [72] proposes an algorithm for computing the top eigenvector of A^TA by running intermediate SVRG steps to approximate the shift-and-invert iteration. Specifically, at each step SVRG is used to solve

$$\min \frac{1}{2}x^T(\lambda I - A^TA)x - b^Tx = \min \sum_{i=1}^{n} \left(\frac{1}{2}x^T \left(\frac{\lambda}{n} I - a_i a_i^T \right) x - \frac{1}{n} b^T x \right).$$

According to [72], if we initialize $y = x_0$ and assume $\|a_i\| = 1$, we have to iterate the following updates

$$x_{k+1} = x_k - \gamma \cdot n \cdot (\nabla f_{s_k}(x_k) - \nabla f_{s_k}(y)) + \gamma \cdot \nabla f(y)$$

where after every T iterations we update $y = x_k$, and the stochastic gradients are of the form $\nabla f_i(x) = \left(\frac{\lambda}{n} I - a_i a_i^T\right) x - \frac{1}{n} b$.

We apply CYCLADES to SVRG with dense linear gradients (see Appendix 7.A) for parallelizing this problem, which uses lazy updates to avoid dense operations on the entire model x. This requires computing the number of skipped updates, τ_j, for each lazily updated coordinate, which may be negative in HOGWILD! due to re-ordering of updates. In our HOGWILD! implementation, we thresholded the bookkeeping variable τ_j (described in Appendix 7.A), as we found that this produced faster convergence. The rows of A are normalized by their ℓ_2-norm, so that we may apply the SVRG algorithm of [72] with uniform sampling. Two graph datasets were used in this experiment. The first, DBLP [78], is an authorship network consisting of 1.4M authors and 4M publications, with 8.65M edges. The second, NH2010 [131], is a weighted topological graph of 49 Census Blocks in New Hampshire, with an edge between adjacent blocks, for a total of 234K edges. We ran SVRG for 50 and 100 epochs for NH2010 and DBLP respectively.

Matrix completion via SGD In the matrix completion problem, we are given a partially observed $n \times m$ matrix M, and wish to factorize it as $M \approx UV$ where U and V are low rank matrices with dimensions $n \times r$ and $r \times m$ respectively. This may be achieved by optimizing

$$\min_{U,V} \sum_{(i,j)\in\Omega} (M_{i,j} - U_{i,\cdot}V_{\cdot,j})^2$$

where Ω is the set of observed entries, which can be approximated by SGD on the observed samples. The objective can also be regularized as:

$$\min_{U,V} \sum_{(i,j)\in\Omega} (M_{i,j} - U_{i,\cdot}V_{\cdot,j})^2 + \frac{\lambda}{2}(\|U\|_F^2 + \|V\|_F^2) = \min_{U,V} \sum_{(i,j)\in\Omega} \left((M_{i,j} - U_{i,\cdot}V_{\cdot,j})^2 + \frac{1}{|\Omega|}\frac{\lambda}{2}(\|U\|_F^2 + \|V\|_F^2) \right).$$

The regularized objective can be optimized by weighted SGD, which samples $(i,j) \in \Omega$ and updates

$$U_{i',\cdot} \leftarrow \begin{cases} (1-\gamma\lambda)U_{i,\cdot} - \gamma \cdot |\Omega| \cdot 2(U_{i,\cdot}V_{\cdot,j} - M_{i,j})(V_{\cdot,j})^T & \text{if } i = i' \\ (1-\gamma\lambda)U_{i',\cdot} & \text{otherwise} \end{cases}$$

and analogously for $V_{\cdot,j}$ In our experiments, we chose a rank of $r = 100$, and ran SGD and weighted SGD for 200 epochs. We used the MovieLens 10M dataset [63] containing 10M ratings for 10,000 movies by 72,000 users.

Word embedding via SGD Semantic word embeddings aim to represent the meaning of a word w via a vector $v_w \in \mathbb{R}^r$. In a recent work by [9], the authors propose using a generative model, and solving for the MLE which is equivalent to:

$$\min_{\{v_w\},C} \sum_{w,w'} A_{w,w'}(\log(A_{w,w'}) - \|v_w + v_{w'}\|_2^2 - C)^2,$$

where $A_{w,w'}$ is the number of times words w and w' co-occur within τ words in the corpus. In our experiments we set $\tau = 10$ following the suggested recipe of the aforementioned paper. We can approximate the solution to the above problem by SGD: we can repeatedly sample entries $A_{w,w'}$ from A and update the corresponding vectors $v_w, v_{w'}$. In this case the update is of the form as:

$$v_w = v_w + 4\gamma A_{w,w'}(\log(A_{w,w'}) - \|v_w + v_{w'}\|_2^2 - C)(v_w + v_{w'})$$

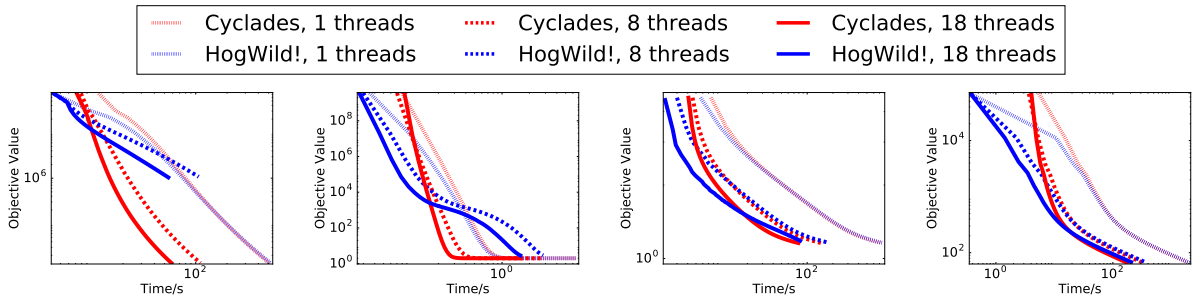
and identically for $v_{w'}$ Then, at the end of each full pass over the data, we update the constant C by its locally optimal value, which can be calculated in closed form:

$$C \leftarrow \frac{\sum_{w,w'} A_{w,w'}(\log(A_{w,w'}) - \|v_w + v_{w'}\|_2^2)}{\sum_{w,w'} A_{w,w'}}.$$

In our experiments, we optimized for a word embedding of dimension $r = 100$, and tested on a 80MB subset of the English Wikipedia dump available at [100]. The dataset contains 213K words and A has 20M non-zero entries. For our experiments, we run SGD for 200 epochs.

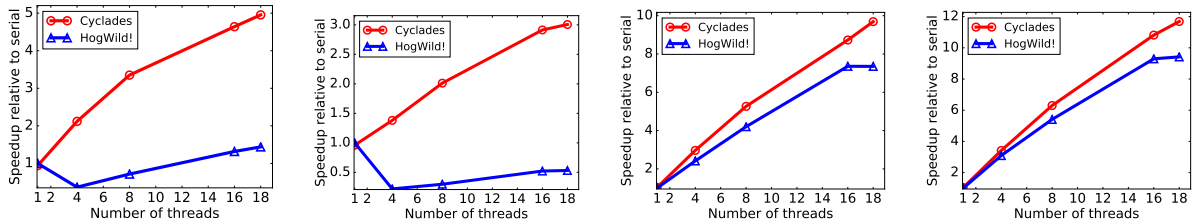
Speedup and Convergence Results

In this subsection, we present the bulk of our experimental findings. Our extended and complete set of results can be found in Appendix 7.F.



(a) Least Squares, DBLP, SAGA (b) Graph Eig., NH2010, SVRG (c) Mat. Comp., 10M, ℓ_2-SGD (d) Word2Vec, EN-Wiki, SGD

Figure 7.3: Convergence of CYCLADES and HOGWILD! in terms of overall running time with 1, 8, 16, 18 threads. CYCLADES is initially slower, but ultimately reaches convergence faster than HOGWILD!.



(a) Least Squares, DBLP, SAGA (b) Graph Eig., NH2010, SVRG (c) Mat. Comp., 10M, ℓ_2-SGD (d) Word2Vec, EN-Wiki, SGD

Figure 7.4: Speedup of CYCLADES and HOGWILD! versus number of threads. On multiple threads, CYCLADES always reaches ϵ objective faster than HOGWILD!. In some cases CYCLADES is faster than HOGWILD! even on 1 thread, due to better cache locality. In Figs. 7.4a and 7.4b, CYCLADES exhibits significant gains, since HOGWILD! suffers from asynchrony noise for which we had to use comparatively smaller stepsizes to prevent divergence.

Least squares When running SAGA for least squares, we found that HOGWILD! was divergent with the large stepsizes that we were using for CYCLADES (Figure 7.6). Thus, in the multi-thread setting, we were only able to use smaller stepsizes for HOGWILD!, which resulted in slower convergence than CYCLADES, as seen in Figure 7.3a. The effects of a smaller stepsize for HOGWILD! are also manisfested in terms of speedups in Figure 7.4a, since HOGWILD! takes a longer time to converge to an ϵ objective value.

Graph eigenvector The convergence of SVRG for graph eigenvectors is shown in Figure 7.3b. CYCLADES starts off slower than HOGWILD!, but always produces results equivalent to the convergence on a single thread. Conversely, HOGWILD! does not exhibit the same behavior on multiple threads as it does serially—in fact, the error due to asynchrony causes HOGWILD! to converge slower on multiple threads. This effect is clearly seen on Figs. 7.4b, where HOGWILD! fails to converge faster than the serial counterpart, and CYCLADES attains a significantly better speedup on 16 threads.

Matrix completion and word embeddings Figures 7.3c and 7.3d show the convergence for the matrix completion and word embeddings problems. CYCLADES is initially slower than HOGWILD! due to the overhead of computing connected components. However, due to better cache locality and convergence properties, CYCLADES is able to reach a lower objective value in less time than HOGWILD!. In fact, we observe that CYCLADES is faster than HOGWILD! when both are run serially, demonstrating that the gains from (temporal) cache locality outweigh the coordination overhead of CYCLADES. These results are reflected in the speedups of CYCLADES and HOGWILD! (Figs. 7.4c and 7.4d). CYCLADES consistently achieves a better speedup ($9 - 10\times$ on 18 threads) compared to that of HOGWILD! ($7 - 9\times$ on 18 threads).

Runtime Breakdown

Partitioning and allocation costs The cost of partitioning and allocation for CYCLADES is given in Table 7.3, relatively to the time that HOGWILD! takes to complete one epoch of stochastic updates (i.e., a single pass over the dataset). For matrix completion and the graph eigenvector problem, on 18 threads, CYCLADES takes the equivalent of 4-6 epochs of HOGWILD! to complete its partitioning, as the problem is either very sparse or the updates are expensive. For solving least squares using SAGA and word embeddings using SGD, the cost of partitioning is equivalent to 11-14 epochs of HOGWILD! on 18 threads. However, we point out that partitioning and allocation is a one-time cost which becomes cheaper with more stochastic update epochs. Additionally, we note that this cost can become amortized quickly due to the extra experiments one has to run for hyperparameter tuning, since the graph partitioning is identical across different stepsizes one might want to test.

# threads	Least Squares SAGA NH2010	Least Squares SAGA DBLP	Graph Eig. SVRG NH2010	Graph Eig. SVRG DBLP	Mat. Comp. SGD MovieLens	Mat. Comp. Weighted SGD MovieLens	Word2Vec SGD EN-Wiki
1	1.9155	2.2245	0.9039	0.9862	0.7567	0.5507	0.5299
4	4.1461	4.6099	1.6244	2.8327	1.8832	1.4509	1.1509
8	6.1157	7.6151	2.0694	4.3836	3.2306	2.5697	1.9372
16	11.7033	13.1351	3.2559	6.2161	5.5284	4.6015	3.5561
18	11.5580	14.1792	4.7639	6.7627	6.1663	5.5270	3.9362

Table 7.3: Cost of partitioning and allocation. The table shows the ratio of the time that CYCLADES consumes for partition and allocation over the time that HOGWILD! takes for 1 full pass over the dataset. On 18 threads, CYCLADES takes between 4-14 HOGWILD! epochs to perform partitioning. Note however, this computational effort is only required once per dataset.

Stochastic updates running time When performing stochastic updates, CYCLADES has better cache locality and coherence, but requires synchronization after each batch. Table 7.4 shows the time for each method to complete a single pass over the dataset, only with respect to stochastic updates (i.e., here we factor out the partitioning time). In most cases, CYCLADES is faster than HOGWILD!. In the cases where CYCLADES is not faster, the overheads of synchronizing outweigh the gains from better cache locality and coherency. However, in some of these cases, synchronization can help by preventing errors due to asynchrony that lead to worse convergence, thus allowing CYCLADES to use larger stepsizes and maximize convergence speed.

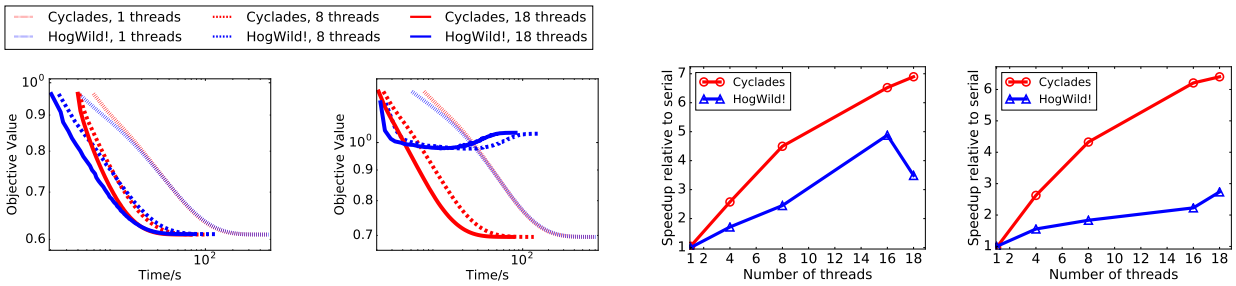
# threads	Mat. Comp. SGD MovieLens		Mat. Comp. ℓ_2-SGD MovieLens		Word2Vec SGD EN-Wiki		Graph Eig. SVRG NH2010		Graph Eig. SVRG DBLP		Least Squares SAGA NH2010		Least Squares SAGA DBLP	
	Cyc	Hog	Cyc	Hog	Cyc	Hog	Cyc	Hog	Cyc	Hog	Cyc	Hog	Cyc	Hog
1	**2.76**	2.87	**3.69**	3.84	**9.85**	10.25	0.07	0.07	11.54	**11.50**	0.04	0.04	**5.01**	5.25
4	**1.00**	1.17	**1.27**	1.51	**2.98**	3.35	0.04	0.04	**4.60**	4.81	0.03	0.03	**1.93**	1.96
8	**0.57**	0.68	**0.71**	0.86	**1.61**	1.89	0.03	0.03	**2.86**	3.03	0.01	0.01	1.04	**1.03**
16	**0.35**	0.40	**0.42**	0.48	**0.93**	1.11	0.02	0.02	**2.03**	2.15	0.01	0.01	0.59	**0.55**
18	**0.32**	0.36	**0.37**	0.40	**0.86**	1.03	0.02	0.02	**1.92**	2.01	0.01	0.01	0.52	**0.51**

Table 7.4: Time, in seconds, to complete one epoch (i.e. full pass of stochastic updates over the data) by CYCLADES and HOGWILD!. Lower times are highlighted in boldface. CYCLADES is usually faster than HOGWILD!, due to its better cache locality and coherence properties.

Diminishing stepsizes

In the previous experiments we used constant stepsizes. Here, we investigate the behavior of CYCLADES and HOGWILD! in the regime where we decrease the stepsize after each epoch. In particular, we ran the matrix completion experiments with SGD (with and without regularization), where we multiplicatively updated the stepsize by 0.95 after each epoch. The convergence and speedup plots are given in Figure 7.5. CYCLADES is able to achieve a speedup of up to $6-7\times$ on $16-18$ threads. On the other hand, HOGWILD! is performing

worse comparatively to its performance with constant stepsizes (Figure 7.4c). The difference is more significant on regularized SGD, where we have to perform lazy updates (Appendix 7.A), and HOGWILD! fails to achieve the same optimum as CYCLADES with multiple threads. Thus, on 18 threads, HOGWILD! obtains a maximum speedup of 3×, whereas CYCLADES attains a speedup of 6.5×.



(a) Convergence, SGD (b) Convergence, ℓ_2-SGD (c) Speedup, SGD (d) Speedup, ℓ_2-SGD

Figure 7.5: Convergence and speedups for SGD and weighted SGD with diminishing stepsizes for the matrix completion on the MovieLens dataset. In this case, CYCLADES outperforms HOGWILD! by achieving up to 6-7x speedup, when HOGWILD! achieves at most 5x speedup for 16-18 threads. For the weighted SGD algorithm, we used lazy updates (Appendix 7.A), in which case HOGWILD! on multiple threads gets to a worse optimum.

Binary Classification and Dense Coordinates

Filtering %	# filtered features	# remaining features
0.048%	1,555	3,228,887
0.047%	1,535	3,228,907
0.034%	1,120	3,229,322
0.028%	915	3,229,527
0.016%	548	3,229,894

Figure 7.8: Filtering of features in URL dataset. with a total of 3,230,442 features before filtering. The maximum percentage of features filtered is less than 0.05%.

In addition to the above experiments, here we explore settings where CYCLADES is expected to perform poorly due to the inherent density of updates (i.e., for data sets with dense features). In particular, we test CYCLADES on a classification problem for text based data, where a few features appear in most data points. Specifically, we run classification for the URL dataset [99] contains $\sim$ 2.4M URLs, labeled as either benign or malicious, and 3.2M features, including bag-of-words representation of tokens in the URL.

For this classification task, we used a logistic regression model, trained using SGD. By its power-law nature, the dataset consists of a small number of extremely dense features which occur in nearly all updates. Since CYCLADES explicitly avoids all conflicts, for these dense cases it will have a schedule of SGD updates that leads to poor speedups. However, we observe that most conflicts are caused by a small percentage of the densest features. If these features are removed from the dataset, CYCLADES

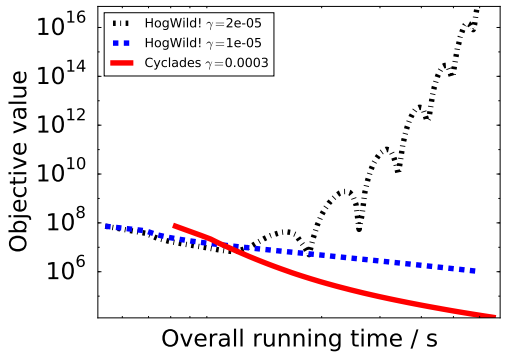


Figure 7.6: Convergence of CYCLADES and HOGWILD! on least squares using SAGA, with 16 threads, on DBLP dataset. HOGWILD! diverges with $\gamma > 10^{-5}$; thus, we were only able to use a smaller step size $\gamma = 10^{-5}$ for HOGWILD! on multiple threads. For HOGWILD! on 1 thread (and CYCLADES on any number of threads), we could use a larger stepsize of $\gamma = 3 \times 10^{-4}$.

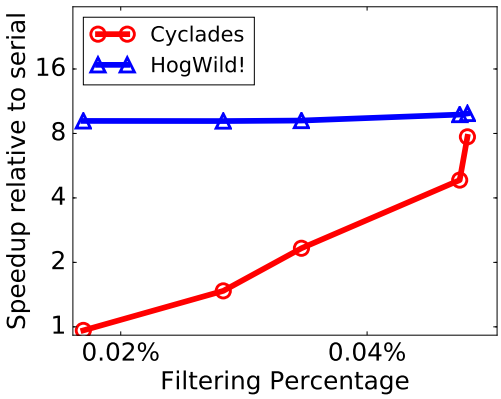


Figure 7.7: Speedups of CYCLADES and HOGWILD! on 16 threads, for different percentage of dense features filtered. When only a very small number of features are filtered, CYCLADES is almost serial. However, as we increase the percentage from 0.016% to 0.048%, the speedup of CYCLADES improves and almost catches up with HOGWILD!.

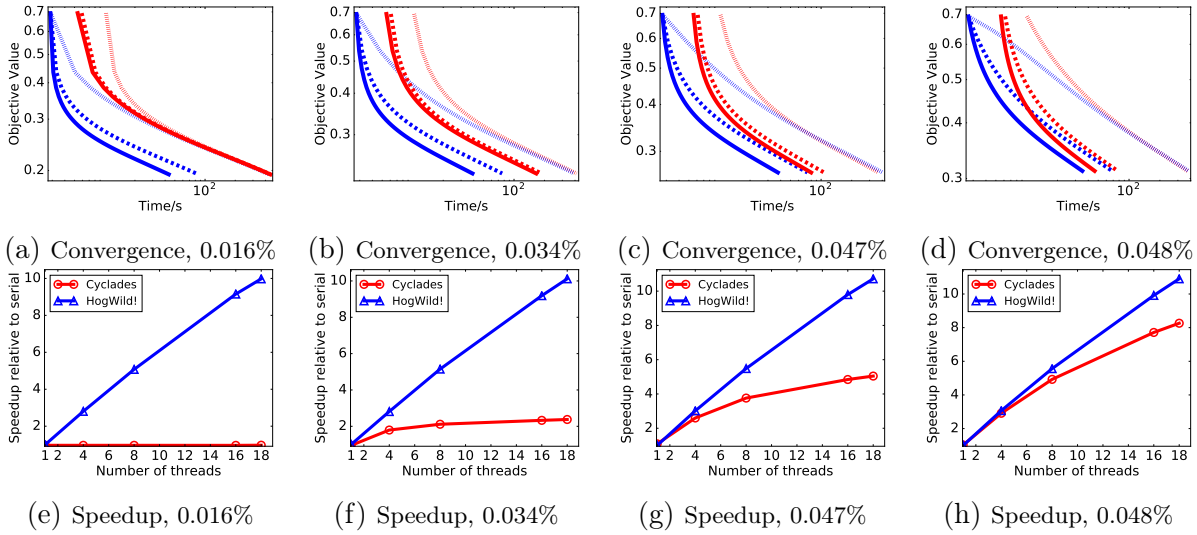
is able to obtain much better speedups. To that end, we ran CYCLADES and HOGWILD! after filtering the densest 0.016% to 0.048% of features. The number of features that are filtered are shown in Table 7.8.

The speedups that are obtained by CYCLADES and HOGWILD! on 16 threads for different filtering percentages are shown in Figure 7.7. Full results of the experiment are presented in Figure 7.9 and in Appendix 7.F. CYCLADES fails to get much speedup when nearly all the features are used, however, as more dense features are removed, CYCLADES obtains a better speedup, almost equalling HOGWILD!'s speedup when 0.048% of the densest features are filtered.

7.5 Additional Related work

Parallel stochastic optimization has been studied under various guises, with literature stretching back at least to the late 60s [36]. The end of Moore's Law coupled with recent advances in parallel and distributed computing technologies have triggered renewed interest [144, 145, 59, 2, 120, 71] in the theory and practice of this field. Much of this contemporary work is built upon the foundational work of Bertsekas, Tsitsiklis et al. [18, 130].

[116] introduced HOGWILD!, a completely lock-free and asynchronous parallel stochastic gradient descent (SGD), in shared-memory multicore systems. Inspired by HOGWILD!'s



(a) Convergence, 0.016% (b) Convergence, 0.034% (c) Convergence, 0.047% (d) Convergence, 0.048%

(e) Speedup, 0.016% (f) Speedup, 0.034% (g) Speedup, 0.047% (h) Speedup, 0.048%

Figure 7.9: Convergence and speedups of CYCLADES and HOGWILD! on 1, 4, 8, 16, 18 threads, for different percentage of dense features filtered.

success at achieving nearly linear speedups for a variety of machine learning tasks, several authors developed other lock-free and asynchronous optimization algorithms, such as parallel stochastic coordinate descent [94, 92]. Additional work in first order optimization and beyond [50, 133, 66, 67, 56], extending to parallel iterative linear solvers [93, 10], has further demonstrated that linear speedups are generically possible in the asynchronous shared-memory setting. Moreover, [118] proposes an analysis for asynchronous parallel, but dense, SVRG, under assumptions similar to those found in [116]. The authors of [41] offer a new analysis for the "coordinate-wise" update version of HOGWILD! using martingales, with similar assumptions to [116], that however can be applied to some non-convex problems. Furthermore, [90] presents an analysis for stochastic gradient methods on smooth, potentially nonconvex functions. Finally, [113] introduces a new framework for analyzing coordinate-wise fixed point stochastic iterations.

Recently, [101] provided a new analysis for asynchronous stochastic optimization by interpreting the effects of asynchrony as noise in the iterates. This perspective of asycnhrony as noise in the iterates was used in [110] to analyze a combinatorial graph clustering algorithm.

Parallel optimization algorithms have also been developed for specific subclasses of problems, including L1-regularized loss minimization [28] and matrix factorization [115].

Calvin [129] and Bohm [54] are general purpose deterministic databases which execute transactions in batches. Both Calvin and Bohm assign an ordering to transactions prior to their execution – the 'determinism' refers to this agreed ordering across processors, which is used to minimize contention during the execution phase. While both databases are serializable, the user has little or no control over the ordering which the transactions are processed. In contrast, CYCLADES guarantees that transactions (or stochastic updates) respect the user-given ordering. This is of particular importance for many stochastic update algorithms which

require that the updates are executed with a uniformly random permutation.

Additionally, Calvin, Bohm, and Cyclades differ significantly in their execution methodologies. Calvin replicates transactions to be executed at every replica where elements of the write set are found. Bohm partitions transactions across processors, but also allows processors to process other transactions that are blocking their execution. Both approaches result in possible redundant processing and could lead to poor cache coherency[3]. CYCLADES partitions both transactions and memory when performing the stochastic updates, and hence has no redundant work nor cache contention. In exchange, CYCLADES may suffer from poorer load balance in higher contention workloads to maintain this better partitioning. As an important consequence of our partitioning choice, we are able to provide theoretical guarantees on CYCLADES's speedup, which are not available in either [129] or [54].

7.6 Discussions

We presented CYCLADES, a general framework for lock-free parallelization of stochastic optimization algorithms, while maintaining serializability and determinism. Our framework can be used to parallelize a large family of stochastic updates algorithms in a conflict-free manner, thereby ensuring the parallelized algorithm produces the same result as its serial counterpart. Theoretical properties, such as convergence rates, are therefore preserved by the CYCLADES-parallelized algorithm, and we provide a single unified theoretical analysis that guarantees near linear speedups.

By eliminating conflicts across processors within each batch of updates, CYCLADES is able to avoid all asynchrony errors and conflicts, and leads to better cache locality and cache coherence than HOGWILD!. These features of CYCLADES translate to near linear speedups in practice, where it can outperform HOGWILD!-type of implementations by up to a factor of $5\times$, in terms of speedups

In the future, we intend to explore hybrids of CYCLADES with HOGWILD!, pushing the boundaries of what is possible in a shared-memory setting. We are also considering solutions for scaling *out* in a distributed setting, where the cost of communication is significantly higher.

7.A Algorithms in the Stochastic Updates family

Here we show that several algorithms belong to the Stochastic Updates (SU) family. These include the well-known stochastic gradient descent, iterative linear solvers, stochastic PCA and others, as well as combinations of weight decay updates, variance reduction methods, and more. Interestingly, even some combinatorial graph algorithms fit under the SU umbrella,

[3] Bohm optimizes for cache coherency during the *concurrency control* phase, when the database is partitioned across processors, but not during the *execution* phase, when work (but not data) is partitioned.

such as the maximal independent set, greedy correlation clustering, and others. We visit some of these algorithms below.

Stochastic Gradient Descent (SGD) Given n functions $f_1, \ldots, f_n$, one often wishes to minimize the average of these functions:

$$\min_x \frac{1}{n} \sum_{i=1}^{n} f_i(x).$$

A popular algorithm to do so —even in the case of non-convex losses— is the stochastic gradient descent:

$$x_{k+1} = x_k - \gamma_k \cdot \nabla f_{i_k}(x_k).$$

In this case, the distribution $\mathcal{D}$ for each sample i_k is usually a with or without replacement uniform sampling among the n functions. For this algorithm the conflict graph between the n possible different updates is completely determined by the support of the gradient vector $\nabla f_{i_k}(x_k)$.

Weight decay and regularization Similar to SGD, in some cases we might wish to regularize the objective with an ℓ_2 term and solve instead the following optimization:

$$\min_x \frac{1}{n} \sum_{i=1}^{n} f_i(x) + \frac{\eta}{2}||x||_2^2.$$

In this case, the update is a weighted version of the "non-regularized" SGD:

$$x_{k+1} = (1 - \gamma_k \eta) \cdot x_k - \gamma_k \cdot \nabla f_{i_k}(x_k).$$

The above stochastic update algorithm can be also be written in the SU language. Although here for each single update the entire model has to be updated with the new weights, we show below that with a simple technique it can be equivalently expressed so that each update is sparse and the support is again determined by the gradient vector $\nabla f_{i_k}(x_k)$.

First order techniques with variance reduction Variance reduction is a technique that is usually employed for (strongly) convex problems, where we wish to minimize the variance of SGD in order to achieve better rates. A popular way to do variance reduction is either through SVRG or SAGA, where a "memory" factor is used in the computation of each gradient update rule. For SAGA we have

$$\begin{aligned} x_{k+1} &= x_k - \gamma \cdot \left(\nabla f_{s_k}(x_k) - g_{s_k} + \frac{1}{n} \sum_{i=1}^{n} g_i \right) \\ g_{s_k} &= \nabla f_{s_k}(x_k). \end{aligned}$$

For SVRG the update rule is

$$x_{k+1} = x_k - \gamma_k \cdot (\nabla f_{i_k}(x_k) - \nabla f_{i_k}(y) + g)$$

where $g = \nabla f(y)$, and y is updated every T iterations of the previous form to be equal to the last x_k iterate. Again, although at first sight the above updates seem to be dense, we show below how we can equivalently rewrite them so that the update-conflict graph is completely determined by the support of the gradient vector $\nabla f_{i_k}(x_k)$.

Combinatorial graph algorithms Interestingly, even some combinatorial graph problems can be phrased in the SU language: greedy correlation clustering (The Pivot Algorithm [5]) and the maximal independent set (these are in fact identical algorithms). In the case of correlation clustering, we are given a graph G vertices joined with either positive or negative edges. Here the objective is to create a number of clusters so that the number of vertex pairs that are sharing negative edges within clusters, plus the number of pairs that are sharing positive edges across clusters, is minimized. For these cases, there exists a very simple algorithm that obtains a 3 approximation for the above objective: randomly sample a vertex, create a cluster with that vertex and its neighborhood, remove that cluster from the graph, and repeat. The above procedure is amenable to the following update rule: Can we put some time indices below?

$$[x_{k+1}]_{N(v)} = \min([x_k]_{N(v)}, v)$$

where x is intialized to ∞, and at each iteration we sample v uniformly among those with label $x_v = \infty$, and $N(v)$ denotes the neighborhood of a vertex in G. Interestingly, we can directly apply the same guarantees of the main CYCLADES theorem here. An optimized implementation of CYCLADES for correlation clustering was developed in [110].

To reiterate, all of the above algorithms, various combinations of them, and further extensions can be written in the language of SU, as presented in Algorithm 7.2.

Lazy Updates

For the cases of weight decay/regularization, and variance reduction, we can reinterpret their inherently dense updates in an equivalent sparse form. Let us consider the following generic form of updates:

$$x_j \leftarrow (1 - \mu_j)x_j - \nu_j + h_{ij}(x_{\mathcal{S}_i}) \tag{7.3}$$

where $h_{ij}(x_{\mathcal{S}_i}) = 0$ for all $j \notin \mathcal{S}_i$. Each stochastic update therefore reads from the set $\mathcal{S}_i$ but writes to every coordinate. However, it is possible to make updates lazily only when they are required. Observe that if τ_j updates are made, each of which have $h_{ij}(x_{\mathcal{S}_i}) = 0$, then we

could rewrite these τ_j updates in closed form as

$$x_j = (1-\mu_j)^{\tau_j} x_j - \nu_j \sum_{k=1}^{\tau_j} (1-\mu_j)^k \tag{7.4}$$

$$= (1-\mu_j)^{\tau_j} x_j - \frac{\nu_j}{\mu_j}(1-\mu_j)\left(1-(1-\mu_j)^{\tau_j}\right). \tag{7.5}$$

This allows the stochastic updates to only write to coordinates in $\mathcal{S}_i$ and defer writes to other coordinates. This procedure is described in Algorithm 7.3. With CYCLADES it is easy to keep track of τ_j, since we know the deterministic order of each stochastic update. On the other hand, it is unclear how a HOGWILD! approach would behave with additional noise in τ_j due to asynchrony. In fact, HOGWILD! could possibly result in negative values of τ_j, and in practice, we find that it is often useful to threshold τ_j by $\max(0, \tau_j)$.

Algorithm 7.3: Lazy Stochastic Updates pseudo-algorithm

Input: x; $f_1, \ldots, f_n$; $u_1, \ldots, u_n$; $g_1, \ldots, g_n$; $\mathcal{D}$; T.
Initialize $\rho(j) = 0$.;
for $t = 1 : T$ **do**
  sample $i \sim \mathcal{D}$;
  $x_{\mathcal{S}_i}$ = read coordinates $\mathcal{S}_i$ from x;
  for $j \in \mathcal{S}_i$ **do**
    $\tau_j = t - \rho(j) - 1$.;
    $x_j \leftarrow (1-\mu_j)^{\tau_j} x_j - \nu_j \sum_{k=1}^{\tau_j} (1-\mu_j)^k$.;
    $x_j \leftarrow (1-\mu_j) x_j - \nu_j + h_{ij}(x_{\mathcal{S}_i})$.;
    $\rho(j) \leftarrow t$.
Output: x

Weight decay and regularization The weighted decay SGD update is a special case of Eq 7.3, with $\mu_j = \eta\gamma$ and $\nu_j = 0$. Eq 7.5 becomes $x_j \leftarrow (1-\eta\gamma)^{\tau_j} x_j$.

Variance reduction with sparse gradients Suppose $\nabla f_i(x)$ is sparse, such that $[\nabla f_i(x)]_j = 0$ for all x and $j \notin \mathcal{S}_i$. Then we can perform SVRG and SAGA using lazy updates, with $\mu_j = 0$. Just-in-time updates for SAG (a similar algorithm to SAGA) were introduced in [121]. For SAGA, the update Eq 7.4 becomes

$$x_j \leftarrow x_j - \gamma \tau_j g_j$$

where $g_j = \left[\frac{1}{n}\sum_{i=1}^n \mathbf{y}_{k,i}\right]_j$ is the jth coordinate of the average gradient. For SVRG, we instead use $g_j = [\nabla f(\mathbf{y})]_j$.

SVRG with dense linear gradients Suppose instead that the gradient is dense, but has linear form $[\nabla f_i(\mathbf{x})]_j = \lambda_j x_j - \kappa_j + \tilde{h}_{ij}(\mathbf{x}_{\mathcal{S}_i})$, where $\tilde{h}_{ij}(\mathbf{x}_{\mathcal{S}_i}) = 0$ for $j \notin \mathcal{S}_i$. The SVRG

stochastic update on the jth coordinate is then

$$\begin{aligned} x_j &\leftarrow x_j - \gamma \left(\lambda_j x_j - \kappa_j + \tilde{h}_{ij}(\mathbf{x}_{\mathcal{S}_i}) - \lambda_j y_j + \kappa_j - \tilde{h}_{ij}(\mathbf{y}_{\mathcal{S}_i}) + g_j \right) \\ &= (1 - \gamma\lambda_j) x_j - \gamma g_j - \gamma \left(\tilde{h}_{ij}(\mathbf{x}_{\mathcal{S}_i}) - \tilde{h}_{ij}(\mathbf{y}_{\mathcal{S}_i}) \right) \end{aligned}$$

where $g_j = [\nabla f(\mathbf{y})]_j$ as above. This fits into our framework with $\mu_j = \gamma\lambda_j$, $\nu_j = \gamma g_j$, and $h_{ij}(\mathbf{x}_{\mathcal{S}_i}) = -\gamma \left(\tilde{h}_{ij}(\mathbf{x}_{\mathcal{S}_i}) - \tilde{h}_{ij}(\mathbf{y}_{\mathcal{S}_i}) \right)$.

7.B With and Without Replacement Proofs

In this Appendix, we show how the sampling and shattering Theorem 7.2 can be restated for sampling with, or without replacement to establish Theorem 7.3.

Let us define three sequences of binary random variables $\{X_i\}_{i=1}^n, \{Y_i\}_{i=1}^n$, and $\{Z_i\}_{i=1}^n$. $\{X_i\}_{i=1}^n$ consists of n i.i.d. Bernoulli random variables, each with probability p. In the second sequence $\{Y_i\}_{i=1}^n$, a random subset of B random variables is set to 1 without replacement. Finally, in the third sequence $\{Z_i\}_{i=1}^n$, we draw B variables with replacement, and we set them to 1. Here, B is integer that satisfies the following bounds

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

Now, consider any function f, that has a "monotonicity" property:

$$f(x_1, \ldots, x_i, \ldots, x_n) \geq f(x_1, \ldots, 0, \ldots, x_n), \text{ for all } i = 1, \ldots, n.$$

Let us now define

$$\begin{aligned} \rho_X &= \Pr\left(f(X_1, \ldots, X_n) > C\right) \\ \rho_Y &= \Pr\left(f(Y_1, \ldots, Y_n) > C\right) \\ \rho_Z &= \Pr\left(f(Z_1, \ldots, Z_n) > C\right) \end{aligned}$$

for some number C, and let us further assume that we have an upper bound on the above probability

$$\rho_X \leq \delta.$$

Our goal is to bound ρ_Y and ρ_Z. By expanding ρ_X using the law of total probability we have

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

where $q_b = \Pr\left(f(X_1, \ldots, X_n) > C \,|\textstyle\sum_{i=1}^n X_i = b\right)$, denotes 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

$$\begin{aligned}\rho_Y &= \sum_{b=0}^{n} \Pr\left(f(Y_1,\ldots,Y_n) > C \,\middle|\, \sum_{i=1}^{n} Y_i = b\right) \cdot \Pr\left(\sum_{i=1}^{n} Y_i = b\right) \\ &\overset{(i)}{=} \sum_{b=0}^{n} q_b \cdot \Pr\left(\sum_{i=1}^{n} Y_i = b\right) \\ &\overset{(ii)}{=} q_B \cdot 1 \end{aligned} \tag{7.6}$$

where (i) comes form the fact that $\Pr\left(f(Y_1,\ldots,Y_n) > C \,|\sum_{i=1}^{n} Y_i = b\right)$ 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.

In the expansion of ρ_X, we can keep the $b = B$ term, and lower bound the probability to obtain:

$$\begin{aligned}\rho_X &= \sum_{b=0}^{n} q_b \cdot \Pr\left(\sum_{i=1}^{n} X_i = b\right) \\ &\geq q_B \cdot \Pr\left(\sum_{i=1}^{n} X_i = B\right) = \rho_Y \cdot \Pr\left(\sum_{i=1}^{n} X_i = B\right) \end{aligned} \tag{7.7}$$

since all terms in the sum are non-negative numbers. Moreover, since X_is are Bernoulli random variables, their sum $\sum_{i=1}^{n} X_i$ is Binomially distributed with parameters n and p. We know that the maximum of the Binomial pmf with parameters n and p occurs at $\Pr\left(\sum_i X_i = B\right)$ where B is the integer that satisfies the upper bound mentioned above: $(n+1)\cdot p - 1 \leq B < (n+1)\cdot p$. Furthermore, the maximum value of the Binomial pmf, with parameters n and p, cannot be less than the corresponding probability of a uniform element:

$$\Pr\left(\sum_{i=1}^{n} X_i = B\right) \geq \frac{1}{n}. \tag{7.8}$$

If we combine (7.7) and (7.8) we get

$$\rho_X \geq \rho_Y / n \Leftrightarrow \rho_Y \leq n \cdot \delta. \tag{7.9}$$

The above establish a relation between the without replacement sampling sequence $\{Y_i\}_{i=1}^{n}$, and the i.i.d. uniform sampling sequence $\{X_i\}_{i=1}^{n}$.

Then, for the last sequence $\{Z_i\}_{i=1}^n$ we have

$$\begin{aligned}\rho_Z &= \sum_{b=0}^{n} \Pr\left(f(Z_1,\ldots,Z_n) > C \,\middle|\, \sum_{i=1}^{n} Z_i = b\right) \cdot \Pr\left(\sum_{i=1}^{n} Z_i = b\right) \\ &\overset{(i)}{=} \sum_{b=1}^{B} q_b \cdot \Pr\left(\sum_{i=1}^{n} Z_i = b\right) \\ &\overset{(ii)}{\leq} \left(\max_{1\leq b\leq B} q_b\right) \cdot \sum_{b=1}^{B} \Pr\left(\sum_{i=1}^{n} Z_i = b\right) \\ &\overset{(iii)}{=} q_B = \rho_Y \leq n \cdot \delta,\end{aligned} \tag{7.10}$$

where (i) comes from the fact that $\Pr\left(\sum_{i=1}^n Z_i = b\right)$ is zero for $b = 0$ and $b > B$, (ii) comes by applying Hölder's Inequality, and (iii) holds since f is assumed to have the monotonicity property:

$$f(x_1,\ldots,x_i,\ldots,x_n) \geq f(x_1,\ldots,0,\ldots,x_n),$$

for any sequence of variables $x_1,\ldots,x_n$. Hence, for any $b_1 \geq b_2$

$$\Pr\left(f(Z_1,\ldots,Z_n) > C \,\middle|\, \sum_{i=1}^{n} Z_i = b_1\right) \geq \Pr\left(f(Z_1,\ldots,Z_n) > C \,\middle|\, \sum_{i=1}^{n} Z_i = b_2\right). \tag{7.11}$$

In conclusion, we have upper bounded ρ_Z and ρ_Y by

$$\rho_Z \leq \rho_Y \leq n \cdot \rho_X \leq n \cdot \delta. \tag{7.12}$$

Application to Theorem 7.3: For our purposes, the above bound Eq. (7.12) allows us to assert Theorem 7.3 for with replacement, without replacement, and i.i.d. sampling, with different constants. Specifically, for any graph G, the size of the largest connected component in the sampled subgraph can be expressed as a function $f_G(x_1,\ldots,x_n)$, where each x_i is an indicator for whether the ith vertex was chosen in the sampling process. Note that f_G is a monotone function, i.e., $f_G(x_1,\ldots,x_i,\ldots,x_n) \geq f_G(x_1,\ldots,0,\ldots,x_n)$ since adding vertices to the sampled subgraph may only increase (or keep constant) the size of the largest connected component. We note that the high probability statement of Theorem 7.2, can be restated so that the constants in front of the size of the connected components accomodate for a statement that is true with probability $1 - 1/n^\zeta$, for any constant $\zeta > 1$. This is required to take care of the extra n factor that appears in the bound of Eq. 7.12, and to obtain Theorem 7.3.

7.C Parallel Connected Components Computation

As we will see in the following, the cost of computing CCs in parallel will depend on the number of cores so that uniform allocation across them is possible, and the number of

edges that are induced by the sampled updates on the bipartite update-variable graph G_u is bounded. As a reminder we denote as G_u^i the bipartite subgraphs of the update-variable graph G_u, that is induced by the sampled updates of the i-th batch. Let us denote as E_u^i the number of edges in G_u^i.

Following the sampling recipe of our main text (i.e., sampling each update per batch uniformly and with replacement), let us assume here that we are sampling $c \cdot n$ updates in total, for some constant $c \geq 1$. Assuming that the size of each batch is $B = (1-\epsilon)\frac{n}{\Delta}$, the total number of sampled batches will be $n_b = \frac{c}{1-\epsilon}\Delta$. The total number of edges in the induced sampled bipartite graphs is a random variable that we denote as

$$Z = \sum_{i=1}^{n_b} E_u^i.$$

Observe that $\mathbb{E}Z = c \cdot E_u$. Using a simple Hoeffding concentration bound we can see that

$$\Pr\left\{|Z - cE_u| > (1+\delta)c \cdot E_u\right\} \leq 2e^{-\frac{2c^2\cdot(1+\delta)^2 E_u^2}{c\cdot n\Delta_{\mathrm{L}}^2}} \leq 2e^{-2c\cdot(1+\delta)^2\cdot\frac{n\overline{\Delta}_{\mathrm{L}}^2}{\Delta_{\mathrm{L}}^2}}$$

where Δ_{L} is the max left degree of the bipartite graph G_u and $\overline{\Delta}_{\mathrm{L}}$ is its average left degree. Now assuming that

$$\frac{\Delta_{\mathrm{L}}}{\overline{\Delta}_{\mathrm{L}}} \leq \sqrt{n}$$

we obtain

$$\Pr\left\{|Z - cE_u| > \log n \cdot c \cdot E_u\right\} \leq 2e^{-c\cdot\log^2 n}.$$

Hence, we get the following simple lemma:

Lemma 7.7. *Let $\frac{\Delta_L}{\overline{\Delta}_L} \leq \sqrt{n}$. Then, the total number of edges $Z = \sum_{i=1}^{n_b} E_u^i$ across the $n_b = \frac{c}{1-\epsilon}\Delta$ sampled subgraphs $G_u^1, \ldots, G_u^{n_b}$ is less than $O(E_u \log n)$ with probability $1 - n^{-\Omega(\log n)}$.*

Now that we have a bound on the number of edges in the sampled subgraphs, we can derive the complexity bounds for computing CCs in parallel. We will break the discussion into the not-too-many- and many-core regime.

The not-too-many cores regime. In this case, we sample n_b subgraphs, allocate them across P cores, and let each core compute CCs on its allocated subgraphs using BFS or DFS. Since each batch is of size $B = (1-\epsilon)\frac{n}{\Delta}$, we need $n_b = \lfloor {}^{c\cdot n}/_{B} \rfloor = c \cdot \lfloor \frac{\Delta}{1-\epsilon} \rfloor$ batches to cover $c \cdot n$ updates in total. If the number of cores is

$$P = O\left(\frac{\overline{\Delta}_{\mathrm{L}}}{\Delta_{\mathrm{L}}} \cdot \Delta\right),$$

then the max cost of a single CC computation on a subgraph (which is $O(B\Delta_{\mathrm{L}})$) is smaller than the average cost across P cores, which is $O(Z/P)$. This implies that a uniform allocation is possible, so that P cores can share the computational effort of computing CCs. Hence, we can get the following lemma:

Lemma 7.8. *Let the number of cores be* $P = O\left(\frac{\overline{\Delta}_L}{\Delta_L} \cdot \Delta\right)$*, and let us sample* $O(\Delta)$ *batches, where each batch is of size* $O(\frac{n}{\Delta})$*. Then, each core will not spend more than* $O\left(\frac{E_u \log n}{P}\right)$ *time in computing CCs, with high probability.*

The many-cores regime. When $P >> \frac{\overline{\Delta}_L}{\Delta_L}$ the uniform balancing of the above method will break, leaving no room for further parallelization. In that case, we can use a very simple "push-label" CC algorithm, whose cost on P cores and arbitrary graphs with E edges and max degree Δ is $O(\max\{\frac{E}{P}, \Delta\} \cdot C_{\max})$, where $C_{\max}$ is the size of the longest-shortest path, or the diameter of the graph [75]. This parallel CC algorithm is given below, where each core is allocated with a number of vertices

Algorithm 7.4: CYCLADES `push-label`

```
1 Initialize shared cc(v) variables to vertexIDs;
2 for i = 1 : length of longest shortest path do
3     for v in the allocated vertex set do
4         for all u that are neighbors of v do
5             Read cc(v) from shared memory;
6             if cc(u) > cc(v) then
7                 Update shared cc(u) ← min(cc(u), cc(v))
```

The above simple algorithm can be significantly slow for graphs where the longest-shortest path is large. Observe, that in the sampled subgraphs G_u^i the size of the shortest-longest path is always bounded by the size of the largest connected component. By Theorem 7.3 that is bounded by $O\left(\frac{\log n}{\epsilon^2}\right)$. Hence, we obtain the following lemma.

Lemma 7.9. *For any number of cores* $P = O(\frac{n}{\Delta \cdot \Delta_L})$*, computing the connected component of a single sampled graph* G_u^i *can be performed in time* $\mathcal{O}(\frac{E_u^i \log n}{P})$*, with high probability.*

Since, we are interested in the overall running time for n_b batches of the CC algorithm, we can see that the above lemma simply boils down to the following:

Corollary 7.10. *For any number of cores* $P = O(\frac{n}{\Delta \cdot \Delta_L})$*, computing the connected component for all sampled graph* $G_u^1, \ldots, G_u^{n_b}$ *can be performed in time* $\mathcal{O}(\frac{E \log^2 n}{P})$*.*

In practice it seems to be that parallelizing the CC computation using the not-too-many core regime policy is significantly more scalable.

7.D Allocating the Conflict Groups

After we have sampled a single batch (i.e., a subgraph G_u^i), and computed the CCs for it, we have to allocate the connected components of that sampled subgraph across cores. Observe

that each connected component will contain at most $\log n$ updates, each ordered according to the a serial predetermined order. Once a core has been assigned all the CCs, it will process all the updates included in the CCs according to the order that each update has been labeled with.

Now assuming that the cost of the i-th update is w_i, the cost of a single connected component $\mathcal{C}$ will be $w_{\mathcal{C}} = \sum_{i \in \mathcal{C}} w_i$. We can now allocate the CCs accross cores so that the maximum core load is minimized, using the following 4/3-approximation algorithm (i.e., an allocation with max load that is at most 4/3 times the maximum between the max weight, and the sum of weights divided by P):

Algorithm 7.5: Greedy allocation

Input: $\{w_1, \ldots, w_m\}$ % weights to be allocated
$b_1 = 0, \ldots, b_P = 0$ % empty buckets;
w = sorted stack of the weights (descending order);
for $i = 1 : m$ **do**
  $w = \text{pop}(w)$;
  add w to bucket b_i with least sum of weights;

To proceed with characterizing the maximum load among the P cores, we assume that the cost of a single update U_i is proportional to the out-degree of that update —according to the update-variable graph G_u— times a constant cost which we shall refer to as κ. Hence, $w_i = O(d_{L,i} \cdot \kappa)$, where $d_{L,i}$ is the degree of the i-th left vertex of G_u.

Observe that the total cost of computing the updates in a single sampled subgraph G_u^i is proportional to $\mathcal{O}(E_u^i \cdot \kappa)$. Moreover, observe that the maximum weight among all CCs cannot be more than $O(\Delta_L \log n \kappa)$ where Δ_L is the max left degree of the bipartite update-variable graph G_u.

Lemma 7.11. *We can allocate CCs such that the maximum load among cores is* $\mathcal{O}\left(\max\left\{\frac{E_u^i}{P}, \Delta_L \log n\right\} \cdot \kappa\right)$, *with high probability, where* κ *is the per edge cost for computing one of the* n *updates defined on* G_u.

If $P = O\left(\frac{n}{\Delta \cdot \Delta_L}\right)$ then the average weight will be larger than the maximum divided by a $\log n$ factor, and a near-uniform allocation of CCs according to their weights possible. Since, we are interested in the overall running time for n_b batches, we can see that the above lemma simply boils down to the following:

Corollary 7.12. *For any number of cores* $P = O(\frac{n}{\Delta \cdot \Delta_L})$, *computing the stochastic updates of the allocated connected component for all sampled graphs (i.e., batches)* $G_u^1, \ldots, G_u^{n_b}$ *can be performed in time* $\mathcal{O}(\frac{E \log^2 n}{P} \cdot \kappa)$.

7.E Robustness against High-degree Outliers

Here, we discuss how CYCLADES can guarantee nearly linear speedups when there is a sublinear $O(n^\delta)$ number of high-conflict updates, as long as the remaining updates have small degree.

Assume that our conflict graph G_c defined between the n update functions has a very high maximum degree Δ_o. However, consider the case where there are only $O(n^\delta)$ nodes that are of that high-degree, while the rest of the vertices have degree much smaller (on the induced subgraph by the latter vertices), say Δ. According to our main analysis, our prescribed batch sizes cannot be greater than $B = (1-\epsilon)\frac{(1-\epsilon)n}{\Delta_o}$. However, if say $\Delta_o = \Theta(n)$, then that would imply that $B = O(1)$, hence there is not room for parallelization by CYCLADES. What we will show, is that by sampling according to $B = (1-\epsilon)\frac{n-O(n^\delta)}{\Delta}$, we can on average expect a parallelization that is similar to the case where the outliers are not present in the conflict graph. For a toy example see Figure 7.10.

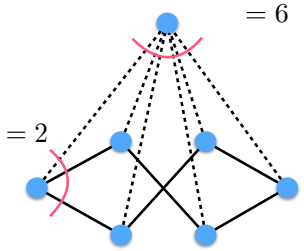


Figure 7.10: The above conflict graph has a vertex with high degree (i.e., $\Delta_o = 6$), and the remaining of the graph has maximum induced degree $\Delta = 2$. In this toy case, when we sample roughly $\frac{n-1}{\Delta} = 3$ vertices, more often than not, the large degree vertex will not be part of the sampled batch. This implies that when parallelizing with CYCLADES these cases will be as much parallelizable as if the high degree vertex was not part of the graph. Each time we happen to sample a batch that includes the max. degree vertex, then essentially we lose all flexibility to parallelize, and we have to run the serial algorithm. What we establish rigorously is that "on average" the parallelization will be as good as one would hope for even in the case where the outliers are not present.

Our main result for the outlier case follows:

Lemma 7.13. *Let us assume that there are $O(n^\delta)$ outlier vertices in the original conflict graph G with degree at most Δ_o, and let the remaining vertices have degree (induced on the remaining graph) at most Δ. Let the induced update-variable graph on these low degree vertices abide to the same graph assumptions as those of Theorem 7.6. Moreover, let the batch size be bounded as*

$$B \leq \min\left\{(1-\epsilon)\frac{n-O(n^\delta)}{\Delta},\ \ O\left(\frac{n^{1-\delta}}{P}\right)\right\}.$$

Then, the expected runtime of CYCLADES *will be* $O\left(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n\right)$.

Proof. Let w_s^i denote the total work required for batch i if that batch contains no outlier notes, and w_o^i otherwise. It is not hard to see that $w_s = \sum_i w_s^i = O\left(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n\right)$ and $w_o = \sum_i w_s^i = O\left(E_u \cdot \kappa \cdot \log^2 n\right)$ Hence, the expected computational effort by CYCLADES will be

$$w_s \cdot \Pr\{\text{a random batch contains no outliers}\} + w_o \Pr\{\text{a random batch contains outliers}\}$$

where

$$\Pr\{\text{a random batch contains no outliers}\} = \Omega\left(\left(1 - \frac{1}{n^{1-\delta}}\right)^B\right) \geq 1 - O\left(\frac{B}{n^{1-\delta}}\right) \quad (7.13)$$

Hence the expected running time will be proportional to $O\left(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n\right)$, if $O(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n) = O(E_u \cdot \kappa \cdot \log^2 n \cdot \frac{B}{n^{1-\delta}})$, which holds when $B = O\left(\frac{n^{1-\delta}}{P}\right)$. □

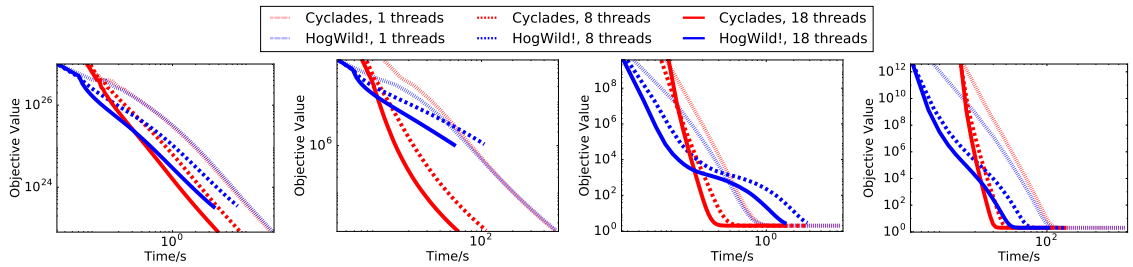
7.F Complete Experiment Results

In this section, we present the remaining experimental results that were left out for brevity from our main experimental section. In Figures 7.11 and 7.12, we show the convergence behaviour of our algorithms, as a function of the overall time, and then as a function of the time that it takes to perform only the stochastic updates (i.e., in Figure 7.12 we factor out the graph partitioning, and allocation time). In Figure 7.13, we provide the complete set of speedup results for all algorithms and data sets we tested, in terms of the number of cores. In Figure 7.14, we provide the speedups in terms of the the computation of the stochastic updates, as a function of the number of cores. Then, in Figures 7.15 – 7.18, we present the convergence, and speedups of the overal computation, and then of the stochastic updates part, for our dense feature URL data set. Finally, in Figure 7.19 we show the divergent behavior of HOGWILD! for the least square experiments with SAGA, on the NH2010 and DBLP datasets.

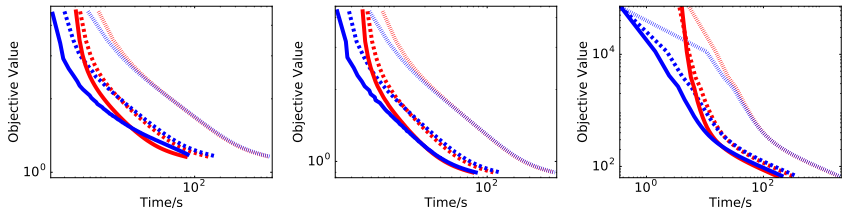
Our overall observations here are similar to the main text. One additional note to make is that when we take a closer look to the figures relative to the times and speedups of the stochastic updates part of CYCLADES (i.e., when we factor out the time of the graph partitioning part), we see that CYCLADES is able to perform stochastic updates faster than HOGWILD! due to its superior spatial and temporal access locality patterns. If the coordination overheads for CYCLADES are excluded, we are able to improve speedups, in some cases by up to 20-70% (Table 7.5). This suggests that by further optimizing the computation of connected components, we can hope for better overall speedups of CYCLADES.

	Mat. Comp. SGD MovieLens	Mat. comp ℓ_2-SGD MovieLens	Word2Vec SGD EN-Wiki	Graph Eig. SVRG NH2010	Graph Eig. SVRG DBLP	Least Squares SAGA NH2010	Least Squares SAGA DBLP
Overall Speedup	8.8010	7.6876	10.4299	2.9134	4.7927	4.4790	4.6405
Speedup of Updates	9.0453	7.9226	11.4610	3.4802	5.5533	4.6998	8.1133
% change	2.7759%	3.0580%	9.8866%	19.4551%	15.8718%	4.9285%	74.8408%

Table 7.5: Speedups of CYCLADES at 16 threads. Two versions speedups are given for each problem: (1) with the overall running time, including the coordination overheads, and (2) using only the running time for stochastic updates. Speedups using only stochastic updates are up to 20% better, which suggests we could potentially observe larger speedups by further optimizing the computation of connected components.

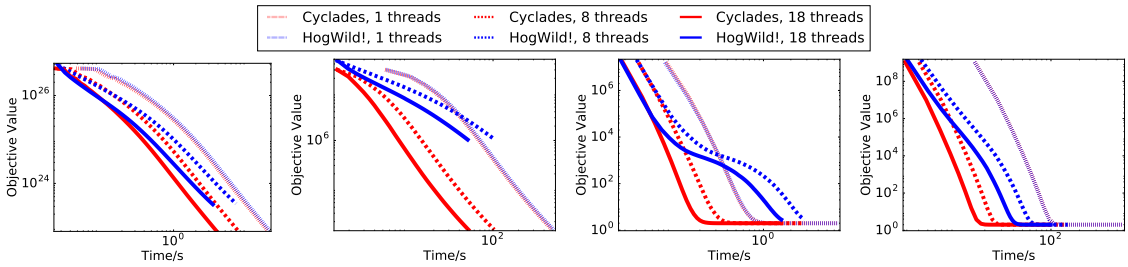


(a) LS, NH2010, SAGA (b) LS, DBLP, SAGA (c) Graph Eig., NH2010, SVRG (d) Graph Eig., DBLP, SVRG

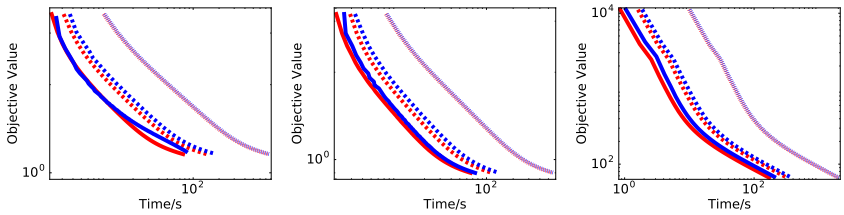


(e) Mat. Comp., 10M, ℓ_2-SGD (f) Mat. Comp., 10M, SGD (g) Word2Vec, EN-Wiki, SGD

Figure 7.11: Convergence of CYCLADES and HOGWILD! on various problems, using 1, 8, 16 threads, in terms of overall running time. CYCLADES is initially slower, but ultimately reaches convergence faster than HOGWILD!.

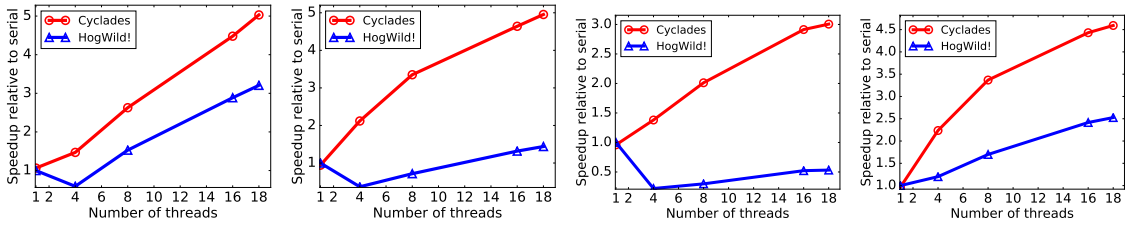


(a) LS, NH2010, SAGA (b) LS, DBLP, SAGA (c) Graph Eig., NH2010, SVRG (d) Graph Eig., DBLP, SVRG

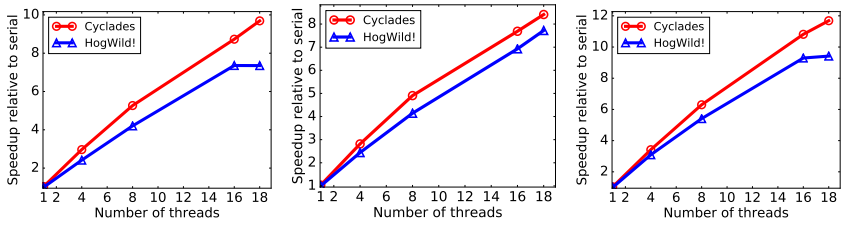


(e) Mat. Comp., 10M, ℓ_2-SGD (f) Mat. Comp., 10M, SGD (g) Word2Vec, EN-Wiki, SGD

Figure 7.12: Convergence of CYCLADES and HOGWILD! on various problems, using 1, 8, 16 threads, in terms of running time for stochastic updates.

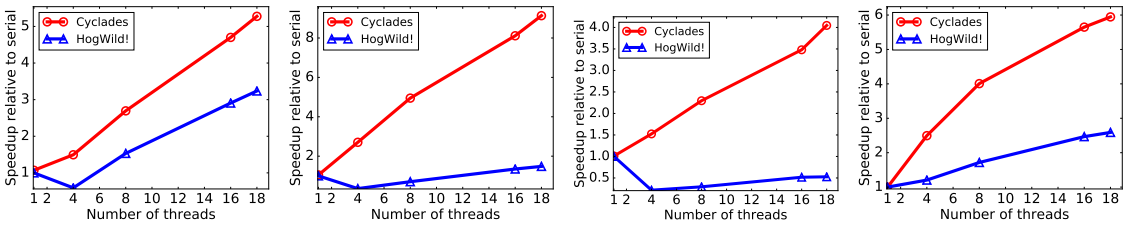


(a) LS, NH2010, SAGA (b) LS, DBLP, SAGA (c) Graph Eig., NH2010, SVRG (d) Graph Eig., DBLP, SVRG

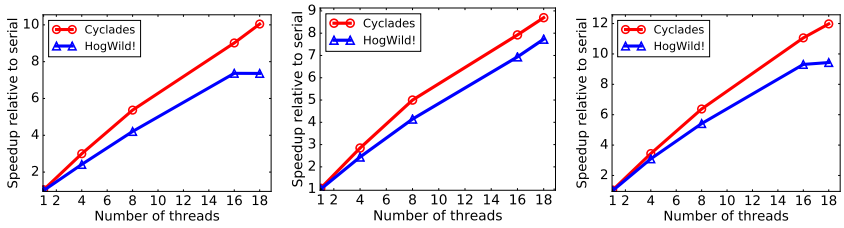


(e) Mat. Comp., 10M, ℓ_2-SGD (f) Mat. Comp., 10M, SGD (g) Word2Vec, EN-Wiki, SGD

Figure 7.13: Speedup of CYCLADES and HOGWILD! on various problems, using 1, 4, 8, 16 threads, in terms of overall running time. On multiple threads, CYCLADES always reaches ϵ objective faster than HOGWILD!. In some cases (7.13a, 7.13e, 7.13g), CYCLADES is faster than HOGWILD! on even 1 thread, as CYCLADES has better cache locality.

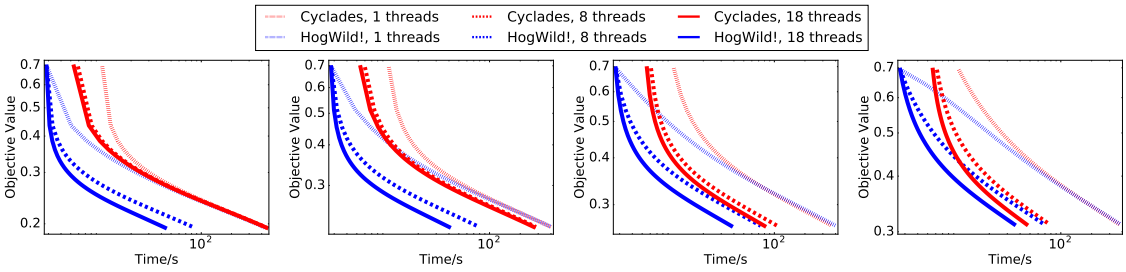


(a) LS, NH2010, SAGA (b) LS, DBLP, SAGA (c) Graph Eig., NH2010, SVRG (d) Graph Eig., DBLP, SVRG



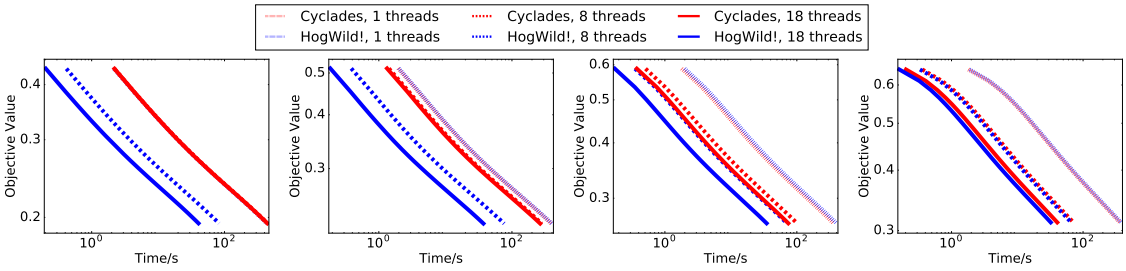
(e) Mat. Comp., 10M, ℓ_2-SGD (f) Mat. Comp., 10M, SGD (g) Word2Vec, EN-Wiki, SGD

Figure 7.14: Speedup of CYCLADES and HOGWILD! on various problems, using 1, 4, 8, 16 threads, in terms of running time for stochastic updates.



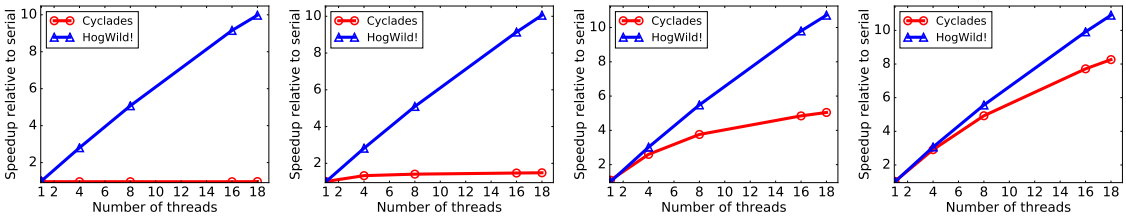
(a) 0.016% (b) 0.028% (c) 0.047% (d) 0.048%

Figure 7.15: Convergence of CYCLADES and HOGWILD! on the malicious URL detection problem, using 1, 4, 8, 16 threads, in terms of overall running time, for different percentage of features filtered.



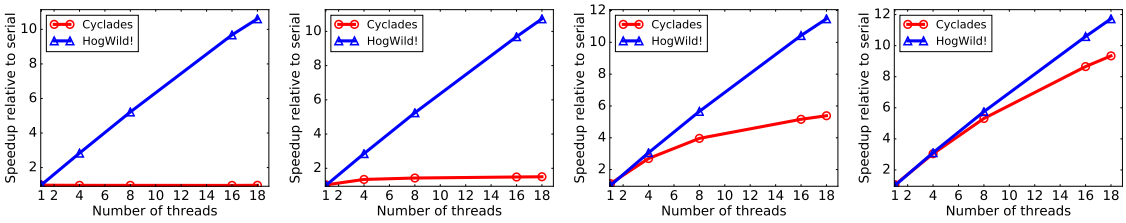
(a) 0.016% (b) 0.028% (c) 0.047% (d) 0.048%

Figure 7.16: Convergence of CYCLADES and HOGWILD! on the malicious URL detection problem, in terms of running time for stochastic updates, for different percentage of features filtered.



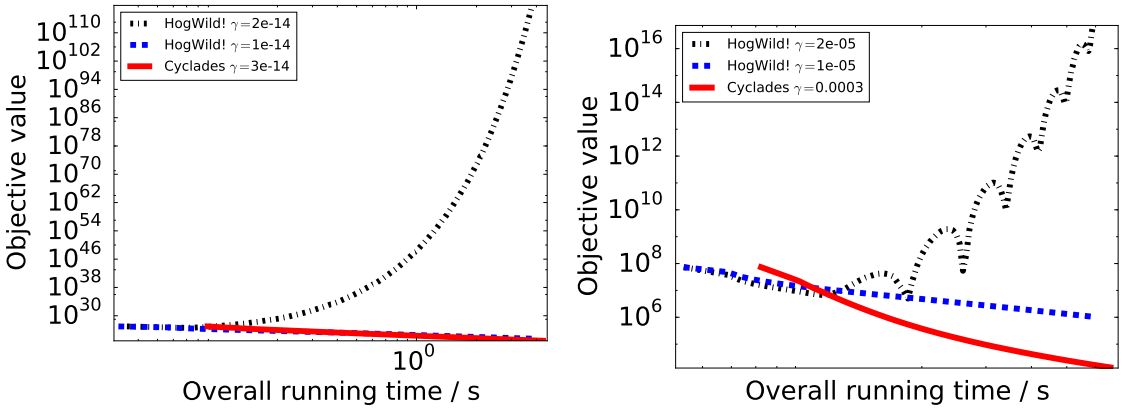
(a) 0.016% (b) 0.028% (c) 0.047% (d) 0.048%

Figure 7.17: Speedup of CYCLADES and HOGWILD! on the malicious URL detection problem, using 1, 4, 8, 16 threads, in terms of overall running time, for different percentage of features filtered.



(a) 0.016% (b) 0.028% (c) 0.047% (d) 0.048%

Figure 7.18: Speedup of CYCLADES and HOGWILD! on the malicious URL detection problem, in terms of running time for stochastic updates, for different percentage of features filtered.



(a) Least squares, NH2010, SAGA

(b) Least squares, DBLP, SAGA

Figure 7.19: Convergence of CYCLADES and HOGWILD! on least squares using SAGA, with 16 threads, on the NH2010 and DBLP datasets. CYCLADES was able to converge using larger stepsizes, but HOGWILD! often diverged with the same large stepsize. Thus, we were only able to use smaller stepsizes for HOGWILD! in the multi-threaded setting.

Chapter 8

Conclusion & Future Directions

In this dissertation, we have proposed the use of concurrency control techniques for scalable parallel machine learning. We demonstrated and evaluated our approach for parallelizing various machine learning algorithms for problems such as nonparametric unsupervised learning, graph clustering, submodular maximization, and sparse stochastic optimization.

Our proposed parallel algorithms are serializable or even deterministic, which serves to decouple the system from the algorithm. This confers the following advantages:

- *Theoretical guarantees of correctness are preserved, independently of parallelism.* Analysis of correctness of the parallel algorithm is thus simplified by a reduction to the serial algorithm. Unlike the coordination-free approach, we do not make further approximations or worsen errors for the sake of greater parallelism.

- *Output is repeatable and reproducible, independently of hardware and parallelism.* One may compare the output of the parallel algorithm against a simpler implementation of the serial algorithm, giving greater confidence that the parallel implementation is correct. Since the output is deterministic and independent of the underlying hardware, it can always be reproduced by others and independently verified.

These strong guarantees come at a cost of greater coordination overhead relative to the coordination-free approach. Moreover, while the concurrency control approach admits a relatively straightforward analysis of correctness, its complexity is shifted to the design and implementation of the parallel algorithm. However, one should note that the tight coupling between systems and algorithms in the coordination-free approach also necessitates a careful implementation in order to achieve good scalability, as has been observed in [140].

Despite the greater coordination overheads, our theoretical analyses prove that the concurrency control approach is scalable. Empirical evaluations also demonstrate that the concurrency control approach is competitive with, if not outperforming, the coordination-free approach in terms of speed, while producing higher-quality results.

Future Directions

Although we have established the feasibility of the concurrency control approach, it remains an open question as to whether there is a characterization of machine learning algorithms that are amenable to this approach. We have found some commonalities in our algorithms that afforded us opportunities for parallelism:

- A transaction that does not change the global state can be committed immediately (OCC-DP-means, OCC-OFL, OCC-BP-means).
- Sampling of discrete random variables have low probability of being affected by small changes to the sampling distribution (OCC-OFL, CC-2g).
- Transactions with sparse dependencies on the global state (i.e., small read and write sets) have low chance of conflict (Cyclades, C4).
- Monotone operations can be correctly executed without coordination (C4).

Heuristically, the algorithms we parallelized in this dissertation have the properties of *bounded changes* and / or *weak dependencies*, i.e., each transaction makes small changes to the global state with possibly low probability, and the output of each transaction is unaffected by either small changes to the global state or changes to a small subset of the global state. A formalism of this heuristic is left as future work, and could serve as a means of identifying a class of machine learning algorithms for parallelization.

Another potential direction of future work involves the interpolation between the concurrency control and coordination-free approaches. As previously noted, the concurrency control approach introduces coordination exactly at the points where the coordination-free approach makes errors. In many cases, we have both the concurrency control and coordination-free algorithms (e.g., C4/ClusterWild!, CC-2g/CF-2g, Cyclades/Hogwild!) for which we understand the scalability of the concurrency control algorithm and the error approximation of its coordination-free counterpart. One could imagine designing an intermediate approach that introduces the concurrency control coordination for the worst-offending transactions with highest errors, thereby using minimal coordination to achieve maximal control. The theoretical analysis would provide guidance for smoothly trading-off accuracy and correctness for speed and scalability.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*. 2016.

[2] Alekh Agarwal and John C Duchi. "Distributed delayed stochastic optimization". In: *Advances in Neural Information Processing Systems*. 2011, pp. 873–881.

[3] Amr Ahmed et al. "Scalable inference in latent variable models". In: *International conference on Web search and data mining (WSDM)*. Vol. 51. 2012, pp. 1257–1264.

[4] N. Ailon, R. Jaiswal, and C. Monteleoni. "Streaming K-means approximation". In: *Advances in Neural Information Processing Systems (NIPS) 22*. Vancouver, 2009.

[5] Nir Ailon, Moses Charikar, and Alantha Newman. "Aggregating inconsistent information: ranking and clustering". In: *Journal of the ACM (JACM)* 55.5 (2008), p. 23.

[6] Noga Alon et al. "Quadratic forms on graphs". In: *Inventiones mathematicae* 163.3 (2006), pp. 499–522.

[7] Tom J Ameloot, Frank Neven, and Jan Van den Bussche. "Relational transducers for declarative networking". In: *Journal of the ACM (JACM)* 60.2 (2013), p. 15.

[8] 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*. IEEE. 2009, pp. 952–963.

[9] Sanjeev Arora et al. "RAND-WALK: A latent variable model approach to word embeddings". In: *arXiv preprint arXiv:1502.03520* (2015).

[10] Haim Avron, Alex Druinsky, and Anshul Gupta. "Revisiting asynchronous linear solvers: Provable convergence rate through randomization". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE. 2014, pp. 198–207.

[11] A. Badanidiyuru and J. Vondrák. "Fast algorithms for maximizing submodular functions". In: *SODA*. 2014.

[12] Mihai Bădoiu, Sariel Har-Peled, and Piotr Indyk. "Approximate Clustering via Coresets". In: *Proc. of the 34th Annual ACM Symposium on Theory of Computing (STOC)*. Montreal, 2002.

[13] B. Bahmani et al. "Scalable Kmeans++". In: *Proc. of the 38th International Conference on Very Large Data Bases (VLDB)*. Istanbul, 2012.

[14] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. "Correlation Clustering". In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society. 2002, pp. 238–238.

[15] Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. "Clustering gene expression patterns". In: *Journal of computational biology* 6.3-4 (1999), pp. 281–297.

[16] Philip A Bernstein and Nathan Goodman. "Concurrency control in distributed database systems". In: *ACM Computing Surveys (CSUR)* 13.2 (1981), pp. 185–221.

[17] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison- Wesley, 1987.

[18] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Vol. 23. Prentice hall Englewood Cliffs, NJ, 1989.

[19] J. Bilmes. *Deep Mathematical Properties of Submodularity with Applications to Machine Learning*. NIPS Tutorial. 2013.

[20] 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*. ACM. 2012, pp. 308–317.

[21] P. Boldi and S. Vigna. "The WebGraph Framework I: Compression Techniques". In: *WWW*. 2004.

[22] P. Boldi et al. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *WWW*. ACM Press, 2011.

[23] P. Boldi et al. "UbiCrawler: A Scalable Fully Distributed Web Crawler". In: *Software: Practice & Experience* 34.8 (2004), pp. 711–726.

[24] 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*. ACM. 2014, pp. 1972–1972.

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

[26] Francesco Bonchi et al. "Chromatic correlation clustering". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2012, pp. 1321–1329.

[27] Stephen Boyd et al. "Distributed Optimization via Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3 (2010), pp. 1–122.

[28] Joseph K Bradley et al. "Parallel coordinate descent for l1-regularized loss minimization". In: *arXiv preprint arXiv:1105.5379* (2011).

[29] Tamara Broderick, Brian Kulis, and Michael I. Jordan. "MAD-Bayes: MAP-based asymptotic derivations from Bayes". In: *Proc. of the 30th International Conference on Machine Learning (ICML)*. 2013.

[30] N. Buchbinder et al. "A Tight Linear Time (1/2)-Approximation for Unconstrained Submodular Maximization". In: *FOCS*. 2012.

[31] N Cesa-Bianchi et al. "A correlation clustering approach to link classification in signed networks". In: *Annual Conference on Learning Theory*. Microtome. 2012, pp. 34–1.

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

[33] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. "Better streaming algorithms for clustering problems". In: *Proc. of the 35th Annual ACM Symposium on Theory of Computing (STOC)*. 2003.

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

[35] Shuchi Chawla et al. "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. Portland, Oregon, USA, 2015, pp. 219–228.

[36] Daniel Chazan and Willard Miranker. "Chaotic relaxation". In: *Linear algebra and its applications* 2.2 (1969), pp. 199–222.

[37] 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*. ACM. 2014, pp. 641–650.

[38] Trishul Chilimbi et al. "Project adam: Building an efficient and scalable deep learning training system". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 571–582.

[39] Cheng-Tao Chu et al. "Map-reduce for machine learning on multicore". In: *NIPS*. Vol. 6. Vancouver, BC. 2006, pp. 281–288.

[40] A. Das et al. "Google news personalization: Scalable online collaborative filtering". In: *Proc. of the 16th World Wide Web Conference*. Banff, 2007.

[41] Christopher M De Sa et al. "Taming the wild: A unified analysis of hogwild-style algorithms". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2674–2682.

[42] Christopher De Sa, Kunle Olukotun, and Christopher Ré. "Ensuring rapid mixing and low bias for asynchronous Gibbs sampling". In: *arXiv preprint arXiv:1602.07415* (2016).

[43] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[44] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in Neural Information Processing Systems*. 2012, pp. 1223–1231.

[45] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives". In: *Advances in Neural Information Processing Systems*. 2014, pp. 1646–1654.

[46] Ofer Dekel et al. "Optimal distributed online prediction". In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 713–720.

[47] Erik D Demaine et al. "Correlation clustering in general weighted graphs". In: *Theoretical Computer Science* 361.2 (2006), pp. 172–187.

[48] I. Dhillon and D.S. Modha. "A data-clustering algorithm on distributed memory multiprocessors". In: *Workshop on Large-Scale Parallel KDD Systems*. 2000.

[49] F. Doshi-Velez et al. "Large Scale Nonparametric Bayesian Inference: Data Parallelisation in the Indian Buffet Process". In: *Advances in Neural Information Processing Systems (NIPS) 22*. Vancouver, 2009.

[50] John Duchi, Michael I Jordan, and Brendan McMahan. "Estimation, optimization, and parallelism when data is sparse". In: *Advances in Neural Information Processing Systems*. 2013, pp. 2832–2840.

[51] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. "Duplicate record detection: A survey". In: *Knowledge and Data Engineering, IEEE Transactions on* 19.1 (2007), pp. 1–16.

[52] 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*. Association for Computational Linguistics. 2009, pp. 19–27.

[53] A. Ene, S. Im, and B. Moseley. "Fast clustering using MapReduce". In: *Proc. of the 17th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. San Diego, 2011.

[54] Jose M Faleiro and Daniel J Abadi. "Rethinking serializable multiversion concurrency control". In: *Proceedings of the VLDB Endowment* 8.11 (2015), pp. 1190–1201.

[55] D. Feldman, A. Krause, and M. Faulkner. "Scalable Training of Mixture Models via Coresets". In: *Advances in Neural Information Processing Systems (NIPS) 24*. Granada, 2011.

[56] Hamid Reza Feyzmahdavian, Arda Aytekin, and Mikael Johansson. "An Asynchronous Mini-Batch Algorithm for Regularized Stochastic Optimization". In: *arXiv preprint arXiv:1505.04824* (2015).

[57] A. Frank. "Submodular functions in graph theory". In: *Discrete Mathematics* 111 (1993), pp. 231–243.

[58] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: the complete book*. 2nd. Pearson Prentice Hall, 2009.

[59] Rainer Gemulla et al. "Large-scale matrix factorization with distributed stochastic gradient descent". In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2011, pp. 69–77.

[60] J. Gillenwater, A. Kulesza, and B. Taskar. "Near-optimal MAP inference for determinantal point processes". In: *Advances in Neural Information Processing Systems (NIPS)*. 2012.

[61] 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*. ACM. 2006, pp. 1167–1176.

[62] Joseph E Gonzalez et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In: *OSDI*. Vol. 12. 1. 2012, p. 2.

[63] GroupLens. *MoveLens 10M dataset*. 2009. URL: `http://grouplens.org/datasets/movielens/10m/` (visited on 01/28/2016).

[64] Joseph M Hellerstein. "The declarative imperative: experiences and conjectures in distributed logic". In: *ACM SIGMOD Record* 39.1 (2010), pp. 5–19.

[65] Q. Ho et al. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server". In: *NIPS*. 2013.

[66] Mingyi Hong. "A Distributed, Asynchronous and Incremental Algorithm for Nonconvex Optimization: An ADMM Based Approach". In: *arXiv preprint arXiv:1412.6058* (2014).

[67] Cho-Jui Hsieh, Hsiang-Fu Yu, and Inderjit S Dhillon. "PASSCoDe: Parallel ASynchronous Stochastic dual Co-ordinate Descent". In: *arXiv preprint arXiv:1504.01365* (2015).

[68] Bilal Hussain et al. *An evaluation of clustering algorithms in duplicate detection*. Tech. rep. 2013.

[69] Alexander Ihler and David Newman. *Bounding Sample Errors in Approximate Distributed Latent Dirichlet Allocation*. Tech. rep. Information and Computer Science, University of California, Irvine, 2009.

[70] Alexander Ihler and David Newman. "Understanding Errors in Approximate Distributed Latent Dirichlet Allocation". In: *Knowledge and Data Engineering, IEEE Transactions on* 24.5 (2012), pp. 952–960.

[71] Martin Jaggi et al. "Communication-efficient distributed dual coordinate ascent". In: *Advances in Neural Information Processing Systems*. 2014, pp. 3068–3076.

[72] Chi Jin et al. "Robust Shift-and-Invert Preconditioning: Faster and More Sample Efficient Algorithms for Eigenvector Computation". In: *arXiv preprint arXiv:1510.08896* (2015).

[73] Matthew J. Johnson, James Saunderson, and Alan S. Willsky. "Analyzing Hogwild Parallel Gaussian Gibbs Sampling". In: *Advances in Neural Information Processing Systems 26*. 2013, pp. 2715–2723. URL: `http://media.nips.cc/nipsbooks/nipspapers/paper_files/nips26/1267.pdf`.

[74] Rie Johnson and Tong Zhang. "Accelerating stochastic gradient descent using predictive variance reduction". In: *Advances in Neural Information Processing Systems*. 2013, pp. 315–323.

[75] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. "Pegasus: A peta-scale graph mining system implementation and observations". In: *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE. 2009, pp. 229–238.

[76] D. Kempe, J. Kleinberg, and E. Tardos. "Maximizing the spread of influence through a social network". In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 2003.

[77] G. Kim et al. "Distributed Cosegmentation via Submodular Optimization on Anisotropic Diffusion". In: *Int. Conference on Computer Vision (ICCV)*. 2011.

[78] KONECT. *DBLP network dataset*. May 2015. URL: `http://konect.uni-koblenz.de/networks/dblp-author` (visited on 01/28/2016).

[79] A. Krause and C. Guestrin. "Submodularity and its Applications in Optimized Information Gathering: An Introduction". In: *ACM Transactions on Intelligent Systems and Technology* 2.4 (2011).

[80] A. Krause and E. Horvitz. "A Utility-Theoretic Approach to Privacy in Online Services". In: *JAIR* 39 (2010).

[81] A. Krause and S. Jegelka. *Submodularity in Machine Learning – New Directions*. ICML Tutorial. 2013.

[82] Michael Krivelevich. "The Phase Transition in Site Percolation on Pseudo-Random Graphs". In: *The Electronic Journal of Combinatorics* 23.1 (2016), pp. 1–12.

[83] Michael Krivelevich. "The phase transition in site percolation on pseudo-random graphs". In: *arXiv preprint arXiv:1404.5731* (2014).

[84] Brian Kulis and Michael I. Jordan. "Revisiting k-means: New Algorithms via Bayesian Nonparametrics". In: *Proc. of 29th International Conference on Machine Learning (ICML)*. Edinburgh, 2012.

[85] R. Kumar et al. "Fast greedy algorithms in MapReduce and streaming". In: *SPAA*. 2013.

[86] Hsiang-Tsung Kung and John T Robinson. "On optimistic methods for concurrency control". In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pp. 213–226.

[87] J. Leskovec. *Stanford Network Analysis Project*. 2011. URL: `http://snap.stanford.edu/`.

[88] Mu Li et al. "Efficient mini-batch training for stochastic optimization". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2014, pp. 661–670.

[89] Mu Li et al. "Parameter Server for Distributed Machine Learning". In: *Big Learn workshop, at NIPS*. Lake Tahoe, 2013.

[90] Xiangru Lian et al. "Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization". In: *arXiv preprint arXiv:1506.08272* (2015).

[91] H. Lin and J. Bilmes. "A Class of Submodular Functions for Document Summarization". In: *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. 2011.

[92] Ji Liu and Stephen J Wright. "Asynchronous stochastic coordinate descent: Parallelism and convergence properties". In: *SIAM Journal on Optimization* 25.1 (2015), pp. 351–376.

[93] Ji Liu, Stephen J Wright, and Srikrishna Sridhar. "An asynchronous parallel randomized Kaczmarz algorithm". In: *arXiv preprint arXiv:1401.4780* (2014).

[94] Ji Liu et al. "An Asynchronous Parallel Stochastic Coordinate Descent Algorithm". In: *ICML 2014*. 2014, pp. 469–477.

[95] Ji Liu et al. "An asynchronous parallel stochastic coordinate descent algorithm". In: *arXiv preprint arXiv:1311.1873* (2013).

[96] D. Lovell et al. "ClusterCluster: Parallel Markov Chain Monte Carlo for Dirichlet Process Mixtures". In: *ArXiv e-prints* (Apr. 2013). arXiv: `1304.2302 [stat.ML]`.

[97] Yucheng Low et al. "Distributed GraphLab: a framework for machine learning and data mining in the cloud". In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727.

[98] Chenxin Ma et al. "Adding vs. Averaging in Distributed Primal-Dual Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, pp. 1973–1982.

[99] Justin Ma et al. "Identifying suspicious URLs: an application of large-scale online learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM. 2009, pp. 681–688.

[100] Matt Mahoney. *Large Text Compression Benchmark*. 2006. URL: `http://mattmahoney.net/dc/text.html` (visited on 01/28/2016).

[101] Horia Mania et al. "Perturbed iterate analysis for asynchronous stochastic optimization". In: *arXiv preprint arXiv:1507.06970* (2015).

[102] A. Meyerson. "Online Facility Location". In: *Proc. of the 42nd Annual Symposium on Foundations of Computer Science (FOCS)*. Las Vegas, 2001.

[103] A. Meyerson et al. "Clustering data streams: Theory and practice". In: *IEEE Transactions on Knowledge and Data Engineering* 15.3 (2003), pp. 515–528.

[104] B. Mirzasoleiman et al. "Distributed Submodular Maximization: Identifying Representative Elements in Massive Data". In: *Advances in Neural Information Processing Systems 26*. 2013.

[105] G.L. Nemhauser, L.A. Wolsey, and M.L. Fisher. "An analysis of approximations for maximizing submodular set functions—I". In: *Mathematical Programming* 14.1 (1978), pp. 265–294.

[106] D. Newman et al. "Distributed Inference for Latent Dirichlet Allocation". In: *Advances in Neural Information Processing Systems (NIPS) 20*. Vancouver, 2007.

[107] John Paisley, David Blei, and Michael I Jordan. "Stick-breaking Beta processes and the Poisson process". In: *Proc. of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2012.

[108] Xinghao Pan et al. "Cyclades: Conflict-free asynchronous machine learning". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2568–2576.

[109] Xinghao Pan et al. "Optimistic concurrency control for distributed unsupervised learning". In: *Advances in Neural Information Processing Systems*. 2013, pp. 1403–1411.

[110] Xinghao Pan et al. "Parallel correlation clustering on big graphs". In: *Advances in Neural Information Processing Systems*. 2015, pp. 82–90.

[111] Xinghao Pan et al. "Parallel double greedy submodular maximization". In: *Advances in Neural Information Processing Systems*. 2014, pp. 118–126.

[112] Xinghao Pan et al. "Revisiting distributed synchronous SGD". In: *arXiv preprint arXiv:1604.00981* (2016).

[113] Z. Peng et al. "ARock: an Algorithmic Framework for Asynchronous Parallel Coordinate Updates". In: *arXiv preprint arXiv:1506.02396* (2015).

[114] Gregory J Puleo and Olgica Milenkovic. "Correlation Clustering with Constrained Cluster Sizes and Extended Weights Bounds". In: *arXiv preprint arXiv:1411.0547* (2014).

[115] Benjamin Recht and Christopher Ré. "Parallel stochastic gradient algorithms for large-scale matrix completion". In: *Mathematical Programming Computation* 5.2 (2013), pp. 201–226.

[116] Benjamin Recht et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in Neural Information Processing Systems*. 2011, pp. 693–701.

[117] Ben Recht et al. "Factoring nonnegative matrices with linear programs". In: *Advances in Neural Information Processing Systems*. 2012, pp. 1214–1222.

[118] Sashank J Reddi et al. "On Variance Reduction in Stochastic Gradient Descent and its Asynchronous Variants". In: *arXiv preprint arXiv:1506.06840* (2015).

[119] C. Reed and Z. Ghahramani. "Scaling the Indian Buffet Process via Submodular Maximization". In: *Int. Conference on Machine Learning (ICML)*. 2013.

[120] Peter Richtárik and Martin Takáč. "Parallel coordinate descent methods for big data optimization". In: *arXiv preprint arXiv:1212.0873* (2012).

[121] Mark Schmidt, Nicolas Le Roux, and Francis Bach. "Minimizing finite sums with the stochastic average gradient". In: *arXiv preprint arXiv:1309.2388* (2013).

[122] A. Schrijver. *Combinatorial Optimization – Polyhedra and efficiency*. Springer, 2002.

[123] L. S. Shapley. "Cores of Convex Games". In: *International Journal of Game Theory* 1.1 (1971), pp. 11–26.

[124] M. Shindler, A. Wong, and A. Meyerson. "Fast and accurate K-means for large Datasets". In: *Advances in Neural Information Processing Systems (NIPS) 24*. Granada, 2011.

[125] Virginia Smith et al. "CoCoA: A General Framework for Communication-Efficient Distributed Optimization". In: *arXiv preprint arXiv:1611.02189* (2016).

[126] Alexander Smola and Shravan Narayanamurthy. "An architecture for parallel topic models". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 703–710.

[127] Evan R Sparks et al. "MLI: An API for distributed machine learning". In: *Data Mining (ICDM), 2013 IEEE 13th International Conference on*. IEEE. 2013, pp. 1187–1192.

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

[129] Alexander Thomson et al. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 1–12.

[130] John N Tsitsiklis, Dimitri P Bertsekas, and Michael Athans. "Distributed asynchronous deterministic and stochastic gradient optimization algorithms". In: *IEEE transactions on automatic control* 31.9 (1986), pp. 803–812.

[131] UF Sparse Matrix Collection. *DIMACS10/nh2010*. 2014. URL: `http://cise.ufl.edu/research/sparse/matrices/DIMACS10/nh2010.html` (visited on 01/28/2016).

[132] Leslie G. Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[133] Yu-Xiang Wang et al. "Asynchronous Parallel Block-Coordinate Frank-Wolfe". In: *stat* 1050 (2014), p. 22.

[134] K. Wei, R. Iyer, and J. Bilmes. "Fast Multi-stage submodular maximization". In: *Int. Conference on Machine Learning (ICML)*. 2014.

[135] Eric P Xing et al. "Petuum: A new platform for distributed machine learning on big data". In: *IEEE Transactions on Big Data* 1.2 (2015), pp. 49–67.

[136] Tianbing Xu and Alexander Ihler. "Multicore Gibbs Sampling in Dense, Unstructured Graphs". In: *Proc. of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2011.

[137] Bo Yang, William K Cheung, and Jiming Liu. "Community mining from signed social networks". In: *Knowledge and Data Engineering, IEEE Transactions on* 19.10 (2007), pp. 1333–1348.

[138] Hyokun Yun et al. "NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion". In: *arXiv preprint arXiv:1312.0193* (2013).

[139] Matei Zaharia et al. "Spark: Cluster computing with working sets". In: *Proc. of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. 2010.

[140] Ce Zhang and Christopher Ré. "DimmWitted: A study of main-memory statistical analytics". In: *Proceedings of the VLDB Endowment* 7.12 (2014), pp. 1283–1294.

[141] Wei Zhang et al. "Staleness-aware async-sgd for distributed deep learning". In: *arXiv preprint arXiv:1511.05950* (2015).

[142] Yuchen Zhang and Michael I Jordan. "Splash: User-friendly programming interface for parallelizing stochastic algorithms". In: *arXiv preprint arXiv:1506.07552* (2015).

[143] Yong Zhuang et al. "A fast parallel sgd for matrix factorization in shared memory systems". In: *Proceedings of the 7th ACM conference on Recommender systems*. ACM. 2013, pp. 249–256.

[144] Martin Zinkevich, John Langford, and Alex J Smola. "Slow learners are fast". In: *Advances in Neural Information Processing Systems*. 2009, pp. 2331–2339.

[145] Martin Zinkevich et al. "Parallelized stochastic gradient descent". In: *Advances in Neural Information Processing Systems*. 2010, pp. 2595–2603.