Large-Scale Optimization Algorithms for Sparse Conditional Gaussian Graphical Models

Calvin McCarter
Machine Learning Department
Carnegie Mellon University
calvinm@cmu.edu

Seyoung Kim
Computational Biology Department
Carnegie Mellon University
sssykim@cs.cmu.edu

Abstract

This paper addresses the problem of scalable optimization for l_1-regularized conditional Gaussian graphical models. Conditional Gaussian graphical models generalize the well-known Gaussian graphical models to conditional distributions to model the output network influenced by conditioning input variables. While highly scalable optimization methods exist for sparse Gaussian graphical model estimation, state-of-the-art methods for conditional Gaussian graphical models are not efficient enough and more importantly, fail due to memory constraints for very large problems. In this paper, we propose a new optimization procedure based on a Newton method that efficiently iterates over two sub-problems, leading to drastic improvement in computation time compared to the previous methods. We then extend our method to scale to large problems under memory constraints, using block coordinate descent to limit memory usage while achieving fast convergence. Using synthetic and genomic data, we show that our methods can solve problems with millions of variables and tens of billions of parameters to high accuracy on a single machine.

1 INTRODUCTION

Sparse Gaussian graphical models (GGMs) [2] have been extremely popular as a tool for learning a network structure over a large number of continuous variables in many different application domains including neuroscience [7] and biology [2]. A sparse GGM can be estimated as a sparse inverse covariance matrix by minimizing the convex function of l_1-regularized negative log-likelihood. Highly scalable learning algorithms such as graphical lasso [2], QUIC [3], and BigQUIC [4] have been proposed to learn the model.

In this paper, we address the problem of scaling up the optimization of sparse conditional GGM (CGGM), a model closely related to sparse GGM, to very large problem sizes without requiring excessive time or memory. Sparse CGGMs have been introduced as a discriminative extension of sparse GGMs to model a sparse network over outputs conditional on input variables [8, 10]. CGGMs can be viewed as a Gaussian analogue of conditional random field [6], while GGMs are a Gaussian analogue of Markov random field. A sparse CGGM can be estimated by minimizing a convex function of l_1-regularized negative log-likelihood. This optimization problem is closely related to that for sparse GGMs because CGGMs also model the network over outputs. However, the presence of the additional parameters in CGGMs for the functional mapping from inputs to outputs makes the optimization significantly more complex than in sparse GGMs.

Several different approaches have been previously proposed to estimate sparse CGGMs, including OWL-QN [8], accelerated proximal gradient method [11], and Newton coordinate descent algorithm [10]. In particular, the Newton coordinate descent algorithm extends the QUIC algorithm [3] for sparse GGM estimation to the case of CGGMs, and has been shown to have superior computational speed and convergence. This approach finds in each iteration a descent direction by minimizing a quadratic approximation of the original negative log-likelihood function along with l_1 regularization. Then, the parameter estimate is updated with this descent direction and a step size found by line search.

Although the Newton coordinate descent method [10] is state-of-the-art for its scalability and fast convergence, it is still not efficient enough to be applied to many real-world problems even with tens of thousands of variables. More importantly, it suffers from a large space requirement, because for very high-dimensional problems, several large

Appearing in Proceedings of the 19^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2016, Cadiz, Spain. JMLR: W&CP volume 51.

dense matrices need to be precomputed and stored during optimization. For a CGGM with p inputs and q outputs, the algorithm requires storing several $p \times p$ and $q \times q$ dense matrices, which cannot fit in memory for large p and q.

We propose new algorithms for learning l_1-regularized CGGMs that significantly improve the computation time of the previous Newton coordinate descent algorithm and also remove the large memory requirement. We first propose an optimization method, called an alternating Newton coordinate descent algorithm, for improving computation time. Our algorithm is based on the key observation that the computation simplifies drastically, if we alternately optimize the two sets of parameters for output network and for mapping inputs to outputs, instead of updating all parameters at once as in the previous approach. The previous approach updated all parameters simultaneously by forming a second-order approximation of the objective on all parameters, which requires an expensive computation of the large Hessian matrix of size $(p+q) \times (p+q)$ in each iteration. Our approach of alternate optimization forms a second-order approximation only on the network parameters, which requires the Hessian of size $q \times q$, as the other set of parameters can be updated easily using a simple coordinate descent.

In order to overcome the constraint on the space requirement, we then extend our algorithm to an alternating Newton block coordinate descent method that can be applied to problems of unbounded size on a machine with limited memory. Instead of recomputing each element of the large matrices on demand, we divide the parameters into blocks for block-wise updates such that the results of computation can be reused within each block. Block-wise parameter updates were previously used in BigQUIC [4] for learning a sparse GGM, where the block sparsity pattern of the network parameters was leveraged to overcome the space limitations. We propose an approach for block-wise update of the output network parameters in CGGMs that extends their idea. We then propose a new block-wise update strategy for the parameters for mapping inputs to outputs. In our experiments, we show that we can solve problems with a million inputs and hundreds of thousands of outputs on a single machine.

The rest of the paper is organized as follows. In Section 2, we provide a brief review of sparse CGGMs and the current state-of-the-art Newton coordinate descent algorithm [10] for learning the models. In Section 3, we propose an alternating Newton coordinate descent algorithm that significantly reduces computation time compared to the previous method. In Section 4, we further extend our algorithm to perform block-wise updates in order to scale up to very large problems on a machine with bounded memory. In Section 5, we demonstrate our proposed algorithms on synthetic and real-world genomic data.

2 BACKGROUND

2.1 The Conditional Gaussian Graphical Model

A CGGM [8, 10] models the conditional probability density of $\mathbf{x} \in \mathbb{R}^p$ given $\mathbf{y} \in \mathbb{R}^q$ as follows:

$$p(\mathbf{y}|\mathbf{x};\mathbf{\Lambda},\mathbf{\Theta}) = \exp\{-\mathbf{y}^T\mathbf{\Lambda}\mathbf{y} - 2\mathbf{x}^T\mathbf{\Theta}\mathbf{y}\}/Z(\mathbf{x}),$$

where $\mathbf{\Lambda}$ is a $q \times q$ matrix for modeling the network over $\mathbf{y}$ and $\mathbf{\Theta}$ is a $p \times q$ matrix for modeling the mapping between the input variables $\mathbf{x}$ and output variables $\mathbf{y}$. The normalization constant is given as $Z(\mathbf{x}) = (2\pi)^{q/2}|\mathbf{\Lambda}|^{-1}\exp(\mathbf{x}^T\mathbf{\Theta}\mathbf{\Lambda}^{-1}\mathbf{\Theta}^T\mathbf{x})$. Inference in a CGGM gives $p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{B}^T\mathbf{x}, \mathbf{\Lambda}^{-1})$, where $\mathbf{B} = -\mathbf{\Theta}\mathbf{\Lambda}^{-1}$, showing the connection to multivariate linear regression.

Given a mean-centered dataset of $\mathbf{X} \in \mathbb{R}^{n\times p}$ and $\mathbf{Y} \in \mathbb{R}^{n\times q}$ for n samples, and their covariance matrices $\mathbf{S_{xx}} = \frac{1}{n}\mathbf{X}^T\mathbf{X}, \mathbf{S_{xy}} = \frac{1}{n}\mathbf{X}^T\mathbf{Y}, \mathbf{S_{yy}} = \frac{1}{n}\mathbf{Y}^T\mathbf{Y}$, a sparse estimate of CGGM parameters can be obtained by minimizing l_1-regularized negative log-likelihood:

$$\min_{\mathbf{\Lambda}\succ 0,\mathbf{\Theta}} f(\mathbf{\Lambda},\mathbf{\Theta}) = g(\mathbf{\Lambda},\mathbf{\Theta}) + h(\mathbf{\Lambda},\mathbf{\Theta}), \qquad (1)$$

where $g(\mathbf{\Lambda},\mathbf{\Theta}) = -\log|\mathbf{\Lambda}| + \mathrm{tr}(\mathbf{S_{yy}}\mathbf{\Lambda} + 2\mathbf{S_{xy}}^T\mathbf{\Theta} + \mathbf{\Lambda}^{-1}\mathbf{\Theta}^T\mathbf{S_{xx}}\mathbf{\Theta})$ for the smooth negative log-likelihood and $h(\mathbf{\Lambda},\mathbf{\Theta})$=$\lambda_{\mathbf{\Lambda}}\|\mathbf{\Lambda}\|_1$+$\lambda_{\mathbf{\Theta}}\|\mathbf{\Theta}\|_1$ for the non-smooth elementwise l_1 penalty. $\lambda_{\mathbf{\Lambda}}, \lambda_{\mathbf{\Theta}} > 0$ are regularization parameters. As observed in [8, 10, 11], this objective is convex.

2.2 Optimization

The current state-of-the-art method for solving Eq. (1) for l_1-regularized CGGM is the Newton coordinate descent algorithm [10] that extends QUIC [3] for l_1-regularized GGM estimation. In each iteration, this algorithm found a generalized Newton descent direction by forming a second-order approximation of the smooth part of the objective and minimizing this along with the l_1 penalty. Given this Newton direction, the parameter estimates were updated with a step size found by line search using Armijo's rule [1].

In each iteration, the Newton coordinate descent algorithm found the Newton direction as follows:

$$\mathbf{D}_{\mathbf{\Lambda}}, \mathbf{D}_{\mathbf{\Theta}} = \underset{\Delta_{\mathbf{\Lambda}},\Delta_{\mathbf{\Theta}}}{\mathrm{argmin}}\ \bar{g}_{\mathbf{\Lambda},\mathbf{\Theta}}(\Delta_{\mathbf{\Lambda}},\Delta_{\mathbf{\Theta}}) + h(\mathbf{\Lambda}+\Delta_{\mathbf{\Lambda}}, \mathbf{\Theta}+\Delta_{\mathbf{\Theta}}), \qquad (2)$$

where $\bar{g}_{\mathbf{\Lambda},\mathbf{\Theta}}$ is the second-order approximation of g given by Taylor expansion:

$$\bar{g}_{\mathbf{\Lambda},\mathbf{\Theta}}(\Delta_{\mathbf{\Lambda}},\Delta_{\mathbf{\Theta}}) = \mathrm{vec}(\nabla g(\mathbf{\Lambda},\mathbf{\Theta}))^T\,\mathrm{vec}([\Delta_{\mathbf{\Lambda}}\ \Delta_{\mathbf{\Theta}}]) + \frac{1}{2}\,\mathrm{vec}([\Delta_{\mathbf{\Lambda}}\ \Delta_{\mathbf{\Theta}}])^T\nabla^2 g(\mathbf{\Lambda},\mathbf{\Theta})\,\mathrm{vec}([\Delta_{\mathbf{\Lambda}}\ \Delta_{\mathbf{\Theta}}]).$$

The gradient and Hessian matrices above are given as:

$$\begin{aligned}\nabla g(\mathbf{\Lambda},\mathbf{\Theta}) &= [\nabla_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta}) \quad \nabla_{\mathbf{\Theta}} g(\mathbf{\Lambda},\mathbf{\Theta})] \\ &= [\mathbf{S_{yy}} - \mathbf{\Sigma} - \mathbf{\Psi} \quad 2\mathbf{S_{xy}} + 2\mathbf{\Gamma}] \qquad (3) \\ \nabla^2 g(\mathbf{\Lambda},\mathbf{\Theta}) &= \begin{bmatrix} \nabla^2_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta}) & \nabla_{\mathbf{\Lambda}}\nabla_{\mathbf{\Theta}} g(\mathbf{\Lambda},\mathbf{\Theta}) \\ \nabla_{\mathbf{\Lambda}}\nabla_{\mathbf{\Theta}} g(\mathbf{\Lambda},\mathbf{\Theta})^T & \nabla^2_{\mathbf{\Theta}} g(\mathbf{\Lambda},\mathbf{\Theta}) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{\Sigma}\otimes(\mathbf{\Sigma}+2\mathbf{\Psi}) & -2\mathbf{\Sigma}\otimes\mathbf{\Gamma}^T \\ -2\mathbf{\Sigma}\otimes\mathbf{\Gamma} & 2\mathbf{\Sigma}\otimes\mathbf{S_{xx}} \end{bmatrix}, \qquad (4)\end{aligned}$$

where $\mathbf{\Sigma} = \mathbf{\Lambda}^{-1}$, $\mathbf{\Psi} = \mathbf{\Sigma}\mathbf{\Theta}^T\mathbf{S_{xx}}\mathbf{\Theta}\mathbf{\Sigma}$, and $\mathbf{\Gamma} = \mathbf{S_{xx}}\mathbf{\Theta}\mathbf{\Sigma}$. Given the Newton direction in Eq. (2), the parameters can be updated as $\mathbf{\Lambda} \leftarrow \mathbf{\Lambda} + \alpha\mathbf{D_\Lambda}$ and $\mathbf{\Theta} \leftarrow \mathbf{\Theta} + \alpha\mathbf{D_\Theta}$, where step size $0 < \alpha \leq 1$ ensures sufficient decrease in Eq. (1) and positive definiteness of $\mathbf{\Lambda}$.

The *Lasso* problem [9] in Eq. (2) was solved using coordinate descent. Despite the efficiency of coordinate descent for *Lasso*, applying coordinate updates repeatedly to all $q^2 + pq$ variables in $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ is costly. So, the updates were restricted to an active set of variables given as:

$$\begin{aligned}\mathcal{S}_{\mathbf{\Lambda}} &= \{(\Delta_{\mathbf{\Lambda}})_{ij} : |(\nabla_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta}))_{ij}| > \lambda_{\mathbf{\Lambda}} \vee \mathbf{\Lambda}_{ij} \neq 0\} \\ \mathcal{S}_{\mathbf{\Theta}} &= \{(\Delta_{\mathbf{\Theta}})_{ij} : |(\nabla_{\mathbf{\Theta}} g(\mathbf{\Lambda},\mathbf{\Theta}))_{ij}| > \lambda_{\mathbf{\Theta}} \vee \mathbf{\Theta}_{ij} \neq 0\}.\end{aligned}$$

Because the active set sizes $m_{\mathbf{\Lambda}} = |\mathcal{S}_{\mathbf{\Lambda}}|, m_{\mathbf{\Theta}} = |\mathcal{S}_{\mathbf{\Theta}}|$ approach the number of non-zero entries in the sparse solutions for $\mathbf{\Lambda}^*$ and $\mathbf{\Theta}^*$ over iterations, this strategy yields a substantial speedup.

To further improve the efficiency of coordinate descent, intermediate results were stored for the large matrix products that need to be computed repeatedly. At the beginning of the optimization for Eq. (2), $\mathbf{U} := \Delta_{\mathbf{\Lambda}}\mathbf{\Sigma}$ and $\mathbf{V} := \Delta_{\mathbf{\Theta}}\mathbf{\Sigma}$ were computed and stored. Then, after a coordinate descent update to $(\Delta_{\mathbf{\Lambda}})_{ij}$, row i and j of $\mathbf{U}$ were updated. Similarly, after an update to $(\Delta_{\mathbf{\Theta}})_{ij}$, row i of $\mathbf{V}$ was updated.

2.3 Computational Complexity and Scalability

Although the Newton coordinate descent method is computationally more efficient than other previous approaches, it does not scale to problems even with tens of thousands of variables. The main computational cost of the algorithm comes from computing the large $(p+q)\times(p+q)$ Hessian matrix in Eq. (4) in each application of Eq. (2) to find the Newton direction. At the beginning of the optimization in Eq. (2), large dense matrices $\mathbf{\Sigma}$, $\mathbf{\Psi}$, and $\mathbf{\Gamma}$, for computing the gradient and Hessian in Eqs. (3) and (4), are precomputed and reused throughout the coordinate descent iterations. Initializing $\mathbf{\Sigma} = \mathbf{\Lambda}^{-1}$ via Cholesky decomposition costs up to $O(q^3)$ time, although in practice, sparse Cholesky decomposition exploits sparsity to invert $\mathbf{\Lambda}$ in much less than $O(q^3)$. Computing $\mathbf{\Psi} = \frac{1}{n}\mathbf{R}^T\mathbf{R}$, where $\mathbf{R} = \mathbf{X}\mathbf{\Theta}\mathbf{\Sigma}$, requires $O(nm_{\mathbf{\Theta}} + nq^2)$ time, and computing $\mathbf{\Gamma}$ costs $O(npq + nq^2)$. After the initializations, the cost of coordinate descent update per each active variable $(\Delta_{\mathbf{\Lambda}})_{ij}$ and $(\Delta_{\mathbf{\Theta}})_{ij}$ is $O(p+q)$. During the coordinate descent for solving Eq. (2), the entire $(p+q)\times(p+q)$ Hessian matrix in Eq. (4) needs to be evaluated, whereas for the gradient in Eq. (3) only those entries corresponding to the parameters in active sets are evaluated.

A more serious obstacle to scaling up to problems with large p and q is the space required to store dense matrices $\mathbf{\Sigma}$ (size $q\times q$), $\mathbf{\Psi}$ (size $q\times q$), and $\mathbf{\Gamma}$ (size $p\times q$). In our experiments on a machine with 104 Gb RAM, the Newton coordinate descent method exhausted memory when $p+q$ exceeded 80,000.

In the next section, we propose a modification of the Newton coordinate descent algorithm that significantly improves the computation time. Then, we introduce blockwise update strategies to our algorithm to remove the memory constraint.

3 ALTERNATING NEWTON COORDINATE DESCENT

In this section, we introduce our alternating Newton coordinate descent algorithm for learning an l_1-regularized CGGM that significantly reduces computation time compared to the previous method. Instead of performing Newton descent for all parameters $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ simultaneously, our approach alternately updates $\mathbf{\Lambda}$ and $\mathbf{\Theta}$, optimizing Eq. (1) over $\mathbf{\Lambda}$ given $\mathbf{\Theta}$ and vice versa until convergence.

Our approach is based on the key observation that with $\mathbf{\Lambda}$ fixed, the problem of solving Eq. (1) over $\mathbf{\Theta}$ becomes simply minimizing a quadratic function with l_1 regularization. Thus, it can be solved efficiently using a coordinate descent method, without the need to form a second-order approximation or to perform line search. On the other hand, optimizing Eq. (1) for $\mathbf{\Lambda}$ given $\mathbf{\Theta}$ still requires forming a quadratic approximation to find a generalized Newton direction and performing line search to find the step size. However, this computation involves only $q\times q$ Hessian matrix and is significantly simpler than performing the same type of computation on both $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ jointly as in the previous approach.

3.1 Coordinate Descent Optimization for $\mathbf{\Lambda}$

Given fixed $\mathbf{\Theta}$, the problem of minimizing the objective in Eq. (1) with respect to $\mathbf{\Lambda}$ becomes

$$\operatorname*{argmin}_{\mathbf{\Lambda}\succ 0} g_{\mathbf{\Theta}}(\mathbf{\Lambda}) + \lambda_{\mathbf{\Lambda}}\|\mathbf{\Lambda}\|_1,$$

where $g_{\mathbf{\Theta}}(\mathbf{\Lambda}) = -\log|\mathbf{\Lambda}| + \operatorname{tr}(\mathbf{S_{yy}}\mathbf{\Lambda} + \mathbf{\Lambda}^{-1}\mathbf{\Theta}^T\mathbf{S_{xx}}\mathbf{\Theta})$. In order to solve this, we first find a generalized Newton direction that minimizes the l_1-regularized quadratic approximation of $g_{\mathbf{\Theta}}(\mathbf{\Lambda})$:

$$\mathbf{D_\Lambda} = \operatorname*{argmin}_{\Delta_{\mathbf{\Lambda}}} \bar{g}_{\mathbf{\Lambda},\mathbf{\Theta}}(\Delta_{\mathbf{\Lambda}}) + \lambda_{\mathbf{\Lambda}}\|\mathbf{\Lambda} + \Delta_{\mathbf{\Lambda}}\|_1, \qquad (5)$$

Algorithm 1: Alternating Newton Coordinate Descent

input : Inputs $X \in \mathbb{R}^{n\times p}$ and $Y \in \mathbb{R}^{n\times q}$; regularization parameters $\lambda_{\mathbf{\Lambda}}, \lambda_{\mathbf{\Theta}}$
output: Parameters $\mathbf{\Lambda}, \mathbf{\Theta}$
Initialize $\mathbf{\Theta} \leftarrow 0, \mathbf{\Lambda} \leftarrow I_q$
for $t = 0, 1, \ldots$ **do**
  Determine active sets $\mathcal{S}_{\mathbf{\Lambda}}, \mathcal{S}_{\mathbf{\Theta}}$
  Solve via coordinate descent:
    $D_{\mathbf{\Lambda}} = \underset{\Delta_{\mathbf{\Lambda}}}{\operatorname{argmin}} \bar{g}_{\mathbf{\Lambda},\mathbf{\Theta}}(\mathbf{\Lambda}+\Delta_{\mathbf{\Lambda}},\mathbf{\Theta}) + h(\mathbf{\Lambda}+\Delta_{\mathbf{\Lambda}},\mathbf{\Theta})$
  Update $\mathbf{\Lambda}^+ = \mathbf{\Lambda} + \alpha D_{\mathbf{\Lambda}}$, where step size α is found with line search
  Solve via coordinate descent:
    $\mathbf{\Theta}^+ = \operatorname{argmin}_{\mathbf{\Theta}}\ g_{\mathbf{\Lambda}}(\mathbf{\Theta}) + \lambda_{\mathbf{\Theta}}\|\mathbf{\Theta}\|_1$

where $\bar{g}_{\mathbf{\Lambda},\mathbf{\Theta}}(\Delta_{\mathbf{\Lambda}})$ is obtained from a second-order Taylor expansion and is given as

$$\bar{g}_{\mathbf{\Lambda},\mathbf{\Theta}}(\Delta_{\mathbf{\Lambda}}) = \operatorname{vec}(\nabla_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta}))^T \operatorname{vec}(\Delta_{\mathbf{\Lambda}}) + \frac{1}{2}\operatorname{vec}(\Delta_{\mathbf{\Lambda}})^T \nabla^2_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta}) \operatorname{vec}(\Delta_{\mathbf{\Lambda}}).$$

The $\nabla_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta})$ and $\nabla^2_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta})$ above are components of the gradient and Hessian matrices corresponding to $\mathbf{\Lambda}$ in Eqs. (3) and (4). We solve the *Lasso* problem in Eq. (5) via coordinate descent. Similar to the Newton coordinate descent method, we maintain $\mathbf{U} := \Delta_{\mathbf{\Lambda}}\mathbf{\Sigma}$ to reuse intermediate results of the large matrix-matrix product. Given the Newton direction for $\mathbf{\Lambda}$, we update $\mathbf{\Lambda} \leftarrow \mathbf{\Lambda} + \alpha\Delta_{\mathbf{\Lambda}}$, where α is obtained by line search.

Restricting the generalized Newton descent to $\mathbf{\Lambda}$ simplifies the computation significantly for coordinate descent updates, compared to the previous approach [10] that applies it to both $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ jointly. Our updates only involve $\nabla_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta})$ and $\nabla^2_{\mathbf{\Lambda}} g(\mathbf{\Lambda},\mathbf{\Theta})$, and no longer involve $\nabla_{\mathbf{\Theta}} g(\mathbf{\Lambda},\mathbf{\Theta})$ and $\nabla_{\mathbf{\Lambda}}\nabla_{\mathbf{\Theta}} g(\mathbf{\Lambda},\mathbf{\Theta})$, eliminating the need to compute the large $p \times q$ dense matrix $\mathbf{\Gamma}$ in $O(npq + nq^2)$ time. Our approach also reduces the computational cost for the coordinate descent update of each element of $\Delta_{\mathbf{\Lambda}}$ from $O(p+q)$ to $O(q)$.

3.2 Coordinate Descent Optimization for $\mathbf{\Theta}$

With $\mathbf{\Lambda}$ fixed, the optimization problem in Eq. (1) with respect to $\mathbf{\Theta}$ becomes

$$\underset{\mathbf{\Theta}}{\operatorname{argmin}}\ g_{\mathbf{\Lambda}}(\mathbf{\Theta}) + \lambda_{\mathbf{\Theta}}\|\mathbf{\Theta}\|_1, \tag{6}$$

where $g_{\mathbf{\Lambda}}(\mathbf{\Theta}) = \operatorname{tr}(2\mathbf{S_{xy}}^T\mathbf{\Theta} + \mathbf{\Lambda}^{-1}\mathbf{\Theta}^T\mathbf{S_{xx}}\mathbf{\Theta})$. Since $g_{\mathbf{\Lambda}}(\mathbf{\Theta})$ is a quadratic function itself, there is no need to form its second-order Taylor expansion or to determine a step size via line search. Instead, we solve Eq. (6) directly with coordinate descent method, storing and maintaining $\mathbf{V} := \mathbf{\Theta}\mathbf{\Sigma}$. Our approach reduces the computation time for updating $\mathbf{\Theta}$ compared to the corresponding computation in the previous algorithm [10]. We avoid computing the large $p \times q$ matrix $\mathbf{\Gamma}$, which had dominated overall computation time with $O(npq)$. Our approach also eliminates the need for line search for updating $\mathbf{\Theta}$. Finally, it reduces the cost for each coordinate descent update in $\mathbf{\Theta}$ to $O(p)$, compared to $O(p+q)$ for the corresponding computation for $\Delta_{\mathbf{\Theta}}$ in the previous method.

Our approach is summarized in Algorithm 1. We provide the details of the coordinate descent update equations in Appendix. In practice, we approximately solve Eqs. (5) and (6) by using a warm-start for $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ with the results of the previous iteration and making a single pass over the active set. This ensures decrease in the objective in Eq. (1) and reduces the overall computation time in practice.

4 ALTERNATING NEWTON BLOCK COORDINATE DESCENT

The alternating Newton coordinate descent algorithm in the previous section improves the computation time of the previous state-of-the-art method, but is still limited by the space required to store large matrices during coordinate descent computation. Solving Eq. (5) for updating $\mathbf{\Lambda}$ requires precomputing and storing $q \times q$ matrices, $\mathbf{\Sigma}$ and $\mathbf{\Psi}$, whereas solving Eq. (6) for updating $\mathbf{\Theta}$ requires $\mathbf{\Sigma}$ and a $p \times p$ matrix $\mathbf{S_{xx}}$. A naive approach to reduce the memory footprint would be to recompute portions of these matrices on demand for each coordinate update, which would be very expensive.

In this section we describe how our algorithm in the previous section can be combined with block coordinate descent to scale up the optimization to very large problems on a machine with limited memory. During coordinate descent optimization, we update blocks of $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ so that within each block, the computation of the large matrices can be cached and re-used, where these blocks are determined automatically by exploiting the sparse stucture. For $\mathbf{\Lambda}$, we extend the block coordinate descent approach in BigQUIC [4] developed for GGMs to take into account the conditioning variables in CGGMs. For $\mathbf{\Theta}$, we describe a new approach for block coordinate descent update. Our algorithm can, in principle, be applied to problems of any size on a machine with limited memory.

4.1 Blockwise Optimization for $\mathbf{\Lambda}$

4.1.1 Block Coordinate Descent Method

A coordinate-descent update of $(\Delta_{\mathbf{\Lambda}})_{ij}$ requires the ith and jth columns of $\mathbf{\Sigma}$ and $\mathbf{\Psi}$. If these columns are in memory, they can be reused. Otherwise, it is a cache miss and we should compute them on demand. $\mathbf{\Sigma}_i$ for the ith column of $\mathbf{\Sigma}$ can be obtained by solving linear system $\mathbf{\Lambda}\mathbf{\Sigma}_i = \mathbf{e}_i$

Algorithm 2: Alternating Newton Block Coordinate Descent

input : Inputs $\mathbf{X} \in \mathbb{R}^{n \times p}$ and outputs $\mathbf{Y} \in \mathbb{R}^{n \times q}$; regularization parameters $\lambda_{\mathbf{\Lambda}}, \lambda_{\mathbf{\Theta}}$
output: Parameters $\mathbf{\Lambda}, \mathbf{\Theta}$
Initialize $\mathbf{\Theta} \leftarrow 0, \mathbf{\Lambda} \leftarrow I_q$
for $t = 0, 1, \ldots$ **do**
  Determine active sets $\mathcal{S}_{\mathbf{\Lambda}}, \mathcal{S}_{\mathbf{\Theta}}$
  Partition columns of $\mathbf{\Lambda}$ into $k_{\mathbf{\Lambda}}$ blocks
    ▷ `Minimize over` $\mathbf{\Lambda}$
  Initialize $\Delta_{\mathbf{\Lambda}} \leftarrow 0$
  for $z = 1$ **to** $k_{\mathbf{\Lambda}}$ **do**
    Compute $\mathbf{\Sigma}_{C_z}$, $\mathbf{U}_{C_z}$, and $\mathbf{\Psi}_{C_z}$
    for $r = 1$ **to** $k_{\mathbf{\Lambda}}$ **do**
      if $z \neq r$ **then**
        Identify columns $B_{zr} \subset C_r$ with active elements in $\mathbf{\Lambda}$
        Compute $\mathbf{\Sigma}_{B_{zr}}$, $\mathbf{U}_{B_{zr}}$, and $\mathbf{\Psi}_{B_{zr}}$
      Update all active $(\Delta_{\mathbf{\Lambda}})_{ij}$ in (C_z, C_r)
  Update $\mathbf{\Lambda}^+ \leftarrow \mathbf{\Lambda} + \alpha \Delta_{\mathbf{\Lambda}}$, where step size α is found with line search
  Partition columns of $\mathbf{\Theta}$ into $k_{\mathbf{\Theta}}$ blocks
    ▷ `Minimize over` $\mathbf{\Theta}$
  for $r = 1$ **to** $k_{\mathbf{\Theta}}$ **do**
    Compute $\mathbf{\Sigma}_{C_r}$, and initialize $\mathbf{V} \leftarrow \mathbf{\Theta}\mathbf{\Sigma}_{C_r}$
    for row $i \in \{1, \ldots, p\}$ if $I_\phi(\mathcal{S}_{(i,C_r)})$ **do**
      Compute $(\mathbf{S}_{\mathbf{xx}})_{ij}$ for non-empty rows j in V_{C_r}
      Update all active $\mathbf{\Theta}_{ij}$ in (i, C_r)

with conjugate gradient method in $O(m_{\mathbf{\Lambda}} K)$ time, where K is the number of conjugate gradient iterations. Then, $\mathbf{\Psi}_i$ can be obtained from $\frac{1}{n}\mathbf{R}^T\mathbf{R}_i$ in $O(nq)$ time, where $\mathbf{R} = \mathbf{X}\mathbf{\Theta}\mathbf{\Sigma}$.

In order to reduce cache misses, we perform block coordinate descent, where within each block, the columns of $\mathbf{\Sigma}$ are cached and re-used. Suppose we partition $\mathcal{N} = \{1, \ldots, q\}$ into $k_{\mathbf{\Lambda}}$ blocks, $C_1, \ldots, C_{k_{\mathbf{\Lambda}}}$. We apply this partitioning to the rows and columns of $\Delta_{\mathbf{\Lambda}}$ to obtain $k_{\mathbf{\Lambda}} \times k_{\mathbf{\Lambda}}$ blocks. We perform coordinate-descent updates in each block, updating all elements in the active set within that block. Let $\mathbf{A}_{C_r}$ denote a q by $|C_r|$ matrix containing columns of $\mathbf{A}$ that corresponds to the subset C_r. In order to perform coordinate-descent updates on (C_z, C_r) block of $\Delta_{\mathbf{\Lambda}}$, we need $\mathbf{\Sigma}_{C_z}$, $\mathbf{\Sigma}_{C_r}$, $\mathbf{\Psi}_{C_z}$, and $\mathbf{\Psi}_{C_r}$. Thus, we pick the smallest possible $k_{\mathbf{\Lambda}}$ such that we can store $2q/k_{\mathbf{\Lambda}}$ columns of $\mathbf{\Sigma}$ and $2q/k_{\mathbf{\Lambda}}$ columns of $\mathbf{\Psi}$ in memory. When updating the variables within block (C_z, C_r) of $\Delta_{\mathbf{\Lambda}}$, there are no cache misses once $\mathbf{\Sigma}_{C_z}$, $\mathbf{\Sigma}_{C_r}$, $\mathbf{\Psi}_{C_z}$, and $\mathbf{\Psi}_{C_r}$ are computed and stored. After updating each $(\Delta_{\mathbf{\Lambda}})_{ij}$ to $(\Delta_{\mathbf{\Lambda}})_{ij} + \mu$, we maintain $\mathbf{U}_{C_z}$ and $\mathbf{U}_{C_r}$ by $\mathbf{U}_{it} \leftarrow \mathbf{U}_{it} + \mu\mathbf{\Sigma}_{jt}, \mathbf{U}_{jt} \leftarrow \mathbf{U}_{jt} + \mu\mathbf{\Sigma}_{it}, \forall t \in \{C_z \cup C_r\}$.

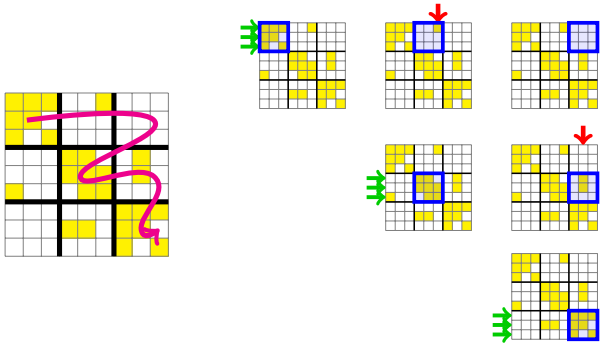


Figure 1: Schematic of block coordinate descent for $\mathbf{\Lambda}$. The $\mathbf{\Lambda}$ of size $q = 9$ is updated for each of the $k_{\mathbf{\Lambda}}^2$ blocks in turn with $k_{\mathbf{\Lambda}} = 3$. Filled elements denote the parameters in the active set. The green arrows denote rows of $\mathbf{\Sigma}$ and $\mathbf{\Psi}$ that are computed once and reused while sweeping through a row of blocks. The red arrows denote cache misses and the corresponding columns of $\mathbf{\Sigma}$ and $\mathbf{\Psi}$ need to be recomputed.

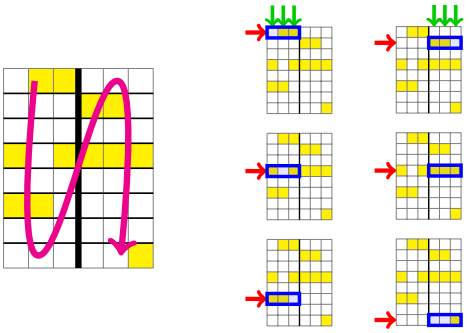


Figure 2: Schematic of block coordinate descent for $\mathbf{\Theta}$. The $\mathbf{\Theta}$ of size $p = 8, q = 6$ is updated for each of the $p \times k_{\mathbf{\Theta}}$ blocks with $k_{\mathbf{\Theta}} = 2$. Filled elements denote the parameters in the active set. Green arrows denote columns of $\mathbf{\Sigma}$ that are computed once and reused while sweeping through the column of p blocks. The red arrows denote cache misses for $(\mathbf{S}_{\mathbf{xx}})_i$.

To go through all blocks, we update blocks $(C_z, C_1), \ldots, (C_z, C_{k_{\mathbf{\Lambda}}})$ for each $z \in \{1, \ldots, k_{\mathbf{\Lambda}}\}$. Since all of these blocks share $\mathbf{\Sigma}_{C_z}$ and $\mathbf{\Psi}_{C_z}$, we precompute and store them in memory. When updating an off-diagonal block $(C_z, C_r), z \neq r$, we compute $\mathbf{\Sigma}_{C_r}$ and $\mathbf{\Psi}_{C_r}$. In the worst case, overall $\mathbf{\Sigma}$ and $\mathbf{\Psi}$ will be computed $k_{\mathbf{\Lambda}}$ times.

4.1.2 Reducing Computational Cost Using Graph Clustering

In typical real-world problems, the graph structure of $\mathbf{\Lambda}$ will exhibit clustering, with an approximately block diagonal structure. We exploit this structure by choosing a partition $\{C_1, \ldots, C_{k_{\mathbf{\Lambda}}}\}$ that reduces cache misses. Within diagonal blocks (C_z, C_z)'s, once $\mathbf{\Sigma}_{C_z}$ and $\mathbf{\Psi}_{C_z}$ are computed, there are no cache misses. For off-diagonal blocks (C_z, C_r)'s, $r \neq z$, we have a cache miss only if some vari-

ables in $\{(\Delta_{\mathbf{\Lambda}})_{ij} | i \in C_z, j \in C_r\}$ lie in the active set. We thus minimize the active set in off-diagonal blocks via clustering, following the strategy for sparse GGM estimation in [4]. In the best case, if all parameters in the active set appear in the diagonal blocks, $\mathbf{\Sigma}$ and $\mathbf{\Psi}$ are computed only once with no cache misses. We use the METIS [5] graph clustering library. Our method for updating $\mathbf{\Lambda}$ is illustrated in Figure 1.

4.2 Blockwise Optimization for $\mathbf{\Theta}$

4.2.1 Block Coordinate Descent Method

The coordinate descent update of $\mathbf{\Theta}_{ij}$ requires $(\mathbf{S_{xx}})_i$ and $\mathbf{\Sigma}_j$ to compute $(\mathbf{S_{xx}})_i^T \mathbf{V}_j$, where $\mathbf{V}_j = \mathbf{\Theta}\mathbf{\Sigma}_j$. If $(\mathbf{S_{xx}})_i$ and $\mathbf{\Sigma}_j$ are not already in the memory, it is a cache miss. Computing $(\mathbf{S_{xx}})_i$ takes $O(np)$, which is expensive if we have many cache misses.

We propose a block coordinate descent approach for solving Eq. (6) that groups these computations to reduce cache misses. Given a partition of $\{1, \ldots, q\}$ into $k_{\mathbf{\Theta}}$ subsets, $C_1, \ldots, C_{k_{\mathbf{\Theta}}}$, we divide $\mathbf{\Theta}$ into $p \times k_{\mathbf{\Theta}}$ blocks, where each block comprises a portion of a row of $\mathbf{\Theta}$. We denote each block (i, C_r), where $i \in \{1, \ldots, p\}$. Since updating block (i, C_r) requires $(\mathbf{S_{xx}})_i$ and $\mathbf{\Sigma}_{C_r}$, we pick the smallest possible $k_{\mathbf{\Theta}}$ such that we can store $q/k_{\mathbf{\Theta}}$ columns of $\mathbf{\Sigma}$ in memory. While performing coordinate descent updates within block (i, C_r) of $\mathbf{\Theta}$, there are no cache misses, once $(\mathbf{S_{xx}})_i$ and $\mathbf{\Sigma}_{C_r}$ are in memory. After updating each $\mathbf{\Theta}_{ij}$ to $\mathbf{\Theta}_{ij} + \mu$, we update $\mathbf{V}_{C_r}$ by $\mathbf{V}_{it} \leftarrow \mathbf{V}_{it} + \mu \mathbf{\Sigma}_{jt}, \forall t \in C_r$.

In order to sweep through all blocks, each time we select a $q \in \{1, \ldots, k_{\mathbf{\Theta}}\}$ and update blocks $(1, C_r), \ldots, (p, C_r)$. Since all of these p blocks with the same C_r share the computation of $\mathbf{\Sigma}_{C_r}$, we compute and store $\mathbf{\Sigma}_{C_r}$ in memory. Within each block, the computation of $(\mathbf{S_{xx}})_i$ is shared, so we pre-compute and store it in memory, before updating this block. The full matrix of $\mathbf{\Sigma}$ will be computed once while sweeping through the full $\mathbf{\Theta}$, whereas in the worst case $\mathbf{S_{xx}}$ will be computed $k_{\mathbf{\Theta}}$ times.

4.2.2 Reducing Computational Cost Using Row-wise Sparsity

We further reduce cache misses for $(\mathbf{S_{xx}})_i$ by strategically selecting partition $C_1, \ldots, C_{k_{\mathbf{\Theta}}}$, based on the observation that if the active set is empty in block (i, C_r), we can skip this block and forgo computing $(\mathbf{S_{xx}})_i$. We therefore choose a partition where the active set variables are clustered into as few blocks as possible. Formally, we want to minimize $\sum_{i,r} I_\phi(\mathcal{S}_{(i,C_r)})$, where $I_\phi(\mathcal{S}_{(i,C_r)})$ is an indicator function that outputs 1 if the active set $\mathcal{S}_{(i,C_r)}$ within block (i, C_r) is not empty.We therefore perform graph clustering over the graph $G = (V, E)$ defined from the active set in $\mathbf{\Theta}$, where $V = \{1, \ldots, q\}$ with one node for each column of $\mathbf{\Theta}$, and $E = \{(j, k) | \exists i : \mathbf{\Theta}_{ij} \in \mathcal{S}_{\mathbf{\Theta}}, \mathbf{\Theta}_{ik} \in \mathcal{S}_{\mathbf{\Theta}}\}$, connecting two nodes j and k with an edge if both $\mathbf{\Theta}_{ij}$ and $\mathbf{\Theta}_{ik}$ are in the active set. This edge set corresponds to the non-zero elements of $\mathbf{\Theta}^T\mathbf{\Theta}$, so the graph can be computed quickly in $O(m_{\mathbf{\Theta}} q)$.

We also exploit row-wise sparsity in $\mathbf{\Theta}$ to reduce the cost of each cache miss. Every empty row in $\mathbf{\Theta}$ corresponds to an empty row in $\mathbf{V} = \mathbf{\Theta}\mathbf{\Sigma}$. Because we only need elements in $(\mathbf{S_{xx}})_i$ for the dot product $(\mathbf{S_{xx}})_i^T \mathbf{V}_j$, we skip computing the kth element of $(\mathbf{S_{xx}})_i$ if the kth row of $\mathbf{\Theta}$ is all zeros. Our strategy for updating $\mathbf{\Theta}$ is illustrated in Figure 2.

Our method is summarized in Algorithm 2. See Appendix for analysis of the computational cost.

4.3 Parallelization

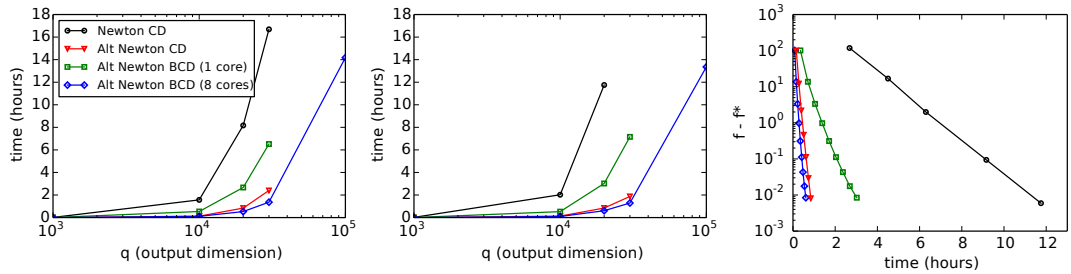
The most expensive computations in our algorithm are embarrassingly parallelizable, allowing for further speedups on machines with multiple cores. Throughout the algorithm, we parallelize matrix and vector multiplications. In addition, for block-wise $\mathbf{\Lambda}$ updates, we compute multiple columns of $\mathbf{\Sigma}_{C_z}$ and $\mathbf{\Psi}_{C_z}$ as well as multiple columns of $\mathbf{\Sigma}_{C_r}$ and $\mathbf{\Psi}_{C_r}$ for multiple cache misses in parallel, running multiple conjugate gradient methods in parallel. For block-wise $\mathbf{\Theta}$ updates, we compute multiple columns of $\mathbf{\Sigma}$ in parallel before sweeping through blocks and perform a parallel computation within each cache miss, computing elements within each $(\mathbf{S_{xx}})_i$ in parallel.

5 EXPERIMENTS

We compare the performance of our methods with the existing state-of-the-art Newton coordinate descent algorithm, using synthetic and real-world genomic datasets. All methods were implemented in C++ with parameters represented in sparse matrix format. All experiments were run on 2.6GHz Intel Xeon E5 machines with 8 cores and 104 Gb RAM, running Linux OS. We run the Newton coordinate descent and alternating Newton coordinate descent algorithms as a single thread job on a single core. For our alternating Newton block coordinate descent method, we run it on a single core and with parallelization on 8 cores.

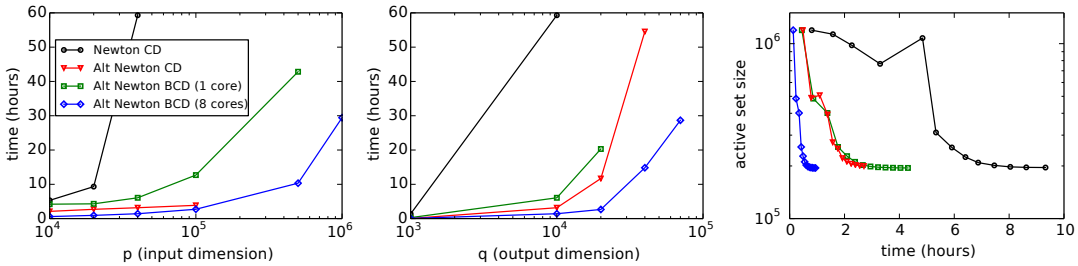
5.1 Synthetic Data Experiments

We compare the different methods on two sets of synthetic datasets, one for chain graphs and another for random graphs with clustering for $\mathbf{\Lambda}$, generated as follows. For chain graphs, the true sparse parameters $\mathbf{\Lambda}$ is set with $\mathbf{\Lambda}_{i,i-1} = 1$ and $\mathbf{\Lambda}_{i,i} = 2.25$ and the true $\mathbf{\Theta}$ is set with $\mathbf{\Theta}_{i,i} = 1$. We perform one set of chain graph experiments with $p = q$, and another set with an additional q irrelevant features unconnected to any outputs, so that $p = 2q$. For random graphs with clustering, following the procedure in [4] for generating a GGM, we set the true $\mathbf{\Lambda}$ to a graph with



(a) (b) (c)

Figure 3: Comparison of scalability on chain graphs. We vary p and q, where (a) $p = q$ and (b) $p = 2q$. The Newton coordinate descent and alternating Newton coordinate descent methods could not be run beyond the problem sizes shown due to memory constraint. (c) Convergence when $q = 20{,}000$ and $p = 40{,}000$.



(a) (b) (c)

Figure 4: Comparison of scalability on random graphs with clustering. (a) Varying p with q fixed at $10{,}000$. (b) Varying q with p fixed at $40{,}000$. (c) Active set size versus time with $p = 20{,}000$ and $q = 10{,}000$.

clusters of nodes of size 250 and with 90% of edges connecting randomly-selected nodes within clusters. We set the number of edges so that the average degree of each node is 10, with edge weights set to 1. We then set the diagonal values so that $\mathbf{\Lambda}$ is positive definite. To set the sparse patterns for $\mathbf{\Theta}$, we randomly select $100\sqrt{p}$ input variables as having edges to at least one output and distribute total $10q$ edges among those selected inputs to influence randomly selected outputs. We set the edge weights of $\mathbf{\Theta}$ to 1.

Then, we draw samples from the CGGM defined by these true $\mathbf{\Lambda}$ and $\mathbf{\Theta}$. We generate datasets with $n = 100$ samples for the chain graphs and $n = 200$ samples for random graphs with clustering. We choose $\lambda_{\mathbf{\Lambda}}$ and $\lambda_{\mathbf{\Theta}}$ so that the number of edges in the estimated $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ is close to ground truth. Following the strategy used in GGM estimation [4], we use the minimum-norm subgradient of the objective as our stopping criterion: $\|\text{grad}^S(\mathbf{\Lambda}_t, \mathbf{\Theta}_t)\|_1 < 0.01(\|\mathbf{\Lambda}\|_1 + \|\mathbf{\Theta}\|_1)$.

We compare the scalability of the different methods on chain graphs of different sizes. We show the computation times for datasets with $p = q$ in Figure 3(a) and for datasets with $p = 2q$ with q additional irrelevant features in Figure 3(b). For large problems, computation times are not shown for Newton coordinate descent and alternating Newton coordinate descent methods because they could not complete the optimization with limited memory. In addition, for large problems, alternating block coordinate descent was terminated after 60 hours of computation. We provide results on varying the sample size n in Appendix. In Figure 3(c), using the dataset with $p = 40{,}000$ and $q = 20{,}000$, we plot the suboptimality in the objective $f - f^*$ over time, where f^* is obtained by running alternating Newton coordinate descent algorithm to numerical precision. Our new methods converge substantially faster than the previous approach, regardless of desired accuracy level. We notice that as expected from the convexity of the optimization problem, all algorithms converge to the global optimum and find nearly identical parameter estimates.

In Figure 4, we compare scalability of different methods for random graphs with clustering. In Figure 4(a), we vary p, while setting q to $10{,}000$. In Figure 4(b), we vary q, fixing p to $40{,}000$. Similar to the results from chain graph, for larger problems, Newton coordinate descent and alternating Newton coordinate descent methods ran out of memory and alternating block coordinate descent was terminated after 60 hours. For all problem sizes, our alternating Newton coordinate descent algorithm significantly reduces the computation time of the previous method, the Newton coordinate descent algorithm. This gap in the computation time increases for larger problems. In Figure 4(c), we compare the convergence in sparsity pattern for the different methods as measured by the active set size, for $p = 20{,}000$ and $q = 10{,}000$. All our methods recover the optimal sparsity pattern much more rapidly than the previous approach.

Figures 3 and 4 show that our alternating Newton block coordinate descent can run on much larger problems than

Table 1: Computation time in hours on genomic dataset. '*' indicates running out of memory.

p	q	$\|\mathbf{\Lambda}^*\|_0$	$\|\mathbf{\Theta}^*\|_0$	Newton CD	Alternating Newton CD	Alternating Newton BCD
34,249	3,268	34,914	28,848	22.0	0.51	**0.24**
34,249	10,256	86,090	103,767	> 50	2.4	**2.3**
442,440	3,268	26,232	30,482	*	*	**11**

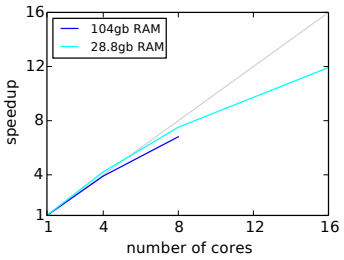


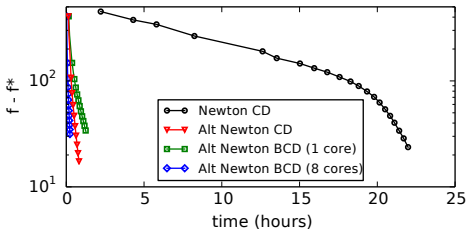
Figure 5: Speedup with parallelization for alternating Newton block coordinate descent.

any other methods, while those methods without block coordinate descent run out of memory. For example, in Figure 4(a) alternating Newton block coordinate descent could handle problems with one million inputs, while without block-wise optimization it ran out of memory when $p > 100{,}000$. We also notice that on a single core, the alternating Newton block coordinate descent is slighly slower than the same method without block-wise optimization because of the need to recompute $\mathbf{\Sigma}$ and $\mathbf{S_{xx}}$. However, it is still substantially faster than the previous method.

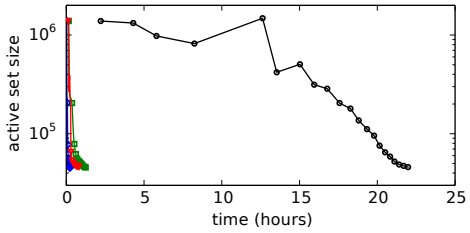
Finally, we evaluate the parallelization scheme for our alternating Newton block coordinate descent method on multi-core machines. Given a dataset generated from cluster graph with $p = 40{,}000$ and $q = 20{,}000$, in Figure 5, we show the folds of speedup for different numbers of cores with respect to a single core. We obtained about 7 times speedup on a 8-core machine with 104Gb RAM, and about 12 times speedup on a 16-core machine with 28Gb RAM. In general, we observe greater speedup on larger problems and for random graphs, because such problems tend to have more cache misses that can be computed in parallel.

5.2 Genomic Data Analysis

We compare the different methods on a genomic dataset. The dataset consists of genotypes for 442,440 single nucleotide polymorphisms (SNPs) and 10,256 gene expression levels for 171 individuals with asthma, after removing genes with variance < 0.01. We fit a sparse CGGM using SNPs as inputs and expressions as outputs to model a gene network influenced by SNPs. We also compared the methods on a smaller dataset of 34,249 SNPs from chromosome 1 and expression levels for 3,268 genes with variance > 0.1. As typically sparse model structures are of interests in this type of analysis, we chose regularization parameters so that the number of non-zero entries in each of $\mathbf{\Theta}$ and



(a)



(b)

Figure 6: Convergence results on genomic dataset. (a) Suboptimality and (b) active set size over time.

$\mathbf{\Lambda}$ at convergence was approximately 10 times the number of genes. The alternating Newton block coordinate descent was run on a 8-core machine with parallelization.

The computation time of different methods are provided in Table 1. On the largest problem, the previous approach could not run due to memory constraint, whereas our block coordinate descent converged in around 11 hours. We also compare the convergence of the different methods on the dataset with 34,249 SNPs and 3,268 gene expressions in Figure 6, and find that our methods provide vastly superior convergence than the previous method.

6 CONCLUSION

In this paper, we addressed the problem of large-scale optimization for sparse CGGMs. We proposed a new optimization procedure, called alternating Newton coordinate descent, that reduces computation time by alternately optimizing for the two sets of parameters $\mathbf{\Lambda}$ and $\mathbf{\Theta}$. Further, we extended this with block-wise optimization so that it can run on any machine with limited memory.

Acknowledgements

This material is based upon work supported by an NSF CAREER Award No. MCB-1149885, Sloan Research Fellowship, and Okawa Foundation Research Grant.

References

[1] L. Armijo et al. Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[2] J. Friedman, T. Hastie, and R. Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, 2008.

[3] C.-J. Hsieh, I. S. Dhillon, P. K. Ravikumar, and M. A. Sustik. Sparse inverse covariance matrix estimation using quadratic approximation. In *Advances in Neural Information Processing Systems 24*, pages 2330–2338. 2011.

[4] C.-J. Hsieh, M. A. Sustik, I. S. Dhillon, P. K. Ravikumar, and R. Poldrack. BIG & QUIC: Sparse inverse covariance estimation for a million variables. In *Advances in Neural Information Processing Systems 26*, pages 3165–3173. 2013.

[5] G. Karypis and V. Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[6] A. Lafferty, J. McCallum and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning*, 2001.

[7] B. Ng, R. Abugharbieh, G. Varoquaux, J. B. Poline, and B. Thirion. Connectivity-informed fMRI activation detection. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2011*, pages 285–292. Springer, 2011.

[8] K. Sohn and S. Kim. Joint estimation of structured sparsity and output structure in multiple-output regression via inverse-covariance regularization. In *Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 16. JMLR W&CP, 2012.

[9] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of Royal Statistical Society, Series B*, 58(1):267–288, 1996.

[10] M. Wytock and J. Kolter. Sparse Gaussian conditional random fields: algorithms, theory, and application to energy forecasting. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28. JMLR W&CP, 2013.

[11] X.-T. Yuan and T. Zhang. Partial Gaussian graphical model estimation. *IEEE Transactions on Information Theory*, 60(3):1673–1687, 2014.

A Appendix

A.1 Coordinate Descent Updates for Alternating Newton Coordinate Descent Method

In our alternating Newton coordinate descent algorithm, each element of $\Delta_{\mathbf{\Lambda}}$ is updated as follows:

$$(\Delta_{\mathbf{\Lambda}})_{ij} \leftarrow (\Delta_{\mathbf{\Lambda}})_{ij} - c_{\mathbf{\Lambda}} + S_{\lambda_{\mathbf{\Lambda}}/a_{\mathbf{\Lambda}}}(c_{\mathbf{\Lambda}} - \frac{b_{\mathbf{\Lambda}}}{a_{\mathbf{\Lambda}}}),$$

where $S_r(w) = \text{sign}(w)\max(|w| - r, 0)$ is the soft-thresholding operator and

$$\begin{aligned}
a_{\mathbf{\Lambda}} =& \mathbf{\Sigma}_{ij}^2 + \mathbf{\Sigma}_{ii}\mathbf{\Sigma}_{jj} + \mathbf{\Sigma}_{ii}\mathbf{\Psi}_{jj} + 2\mathbf{\Sigma}_{ij}\mathbf{\Psi}_{ij} + \mathbf{\Sigma}_{jj}\mathbf{\Psi}_{ii} \\
b_{\mathbf{\Lambda}} =& (\mathbf{S}_{\mathbf{yy}})_{ij} - \mathbf{\Sigma}_{ij} - \mathbf{\Psi}_{ij} + (\mathbf{\Sigma}\Delta_{\mathbf{\Lambda}}\mathbf{\Sigma})_{ij} + (\mathbf{\Psi}\Delta_{\mathbf{\Lambda}}\mathbf{\Sigma})_{ij} + (\mathbf{\Psi}\Delta_{\mathbf{\Lambda}}\mathbf{\Sigma})_{ji} \\
c_{\mathbf{\Lambda}} =& \mathbf{\Lambda}_{ij} + (\Delta_{\mathbf{\Lambda}})_{ij}.
\end{aligned}$$

For $\mathbf{\Theta}$, we perform coordinate-descent updates directly on $\mathbf{\Theta}$ without forming a second-order approximation of the log-likelihood to find a Newton direction, as follows:

$$\mathbf{\Theta}_{ij} \leftarrow \mathbf{\Theta}_{ij} - c_{\mathbf{\Theta}} + S_{\lambda_{\mathbf{\Theta}}/a_{\mathbf{\Theta}}}(c_{\mathbf{\Theta}} - \frac{b_{\mathbf{\Theta}}}{a_{\mathbf{\Theta}}}),$$

where

$$\begin{aligned}
a_{\mathbf{\Theta}} =& 2\mathbf{\Sigma}_{jj}(\mathbf{S}_{\mathbf{xx}})_{ii} \\
b_{\mathbf{\Theta}} =& 2(\mathbf{S}_{\mathbf{xy}})_{ij} + 2(\mathbf{S}_{\mathbf{xx}}\mathbf{\Theta}\mathbf{\Sigma})_{ij} \\
c_{\mathbf{\Theta}} =& \mathbf{\Theta}_{ij}.
\end{aligned}$$

A.2 Time Complexity Analysis of Alternating Newton Block Coordinate Descent

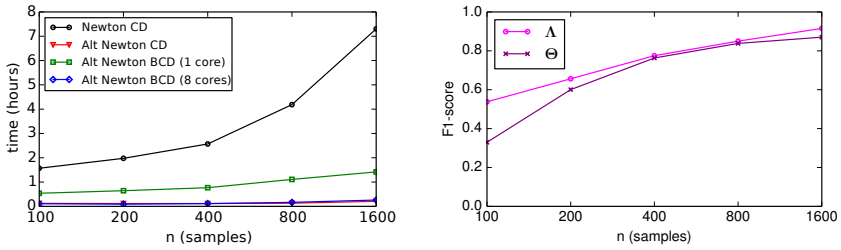
In this section we describe the time complexity of the alternating Newton block coordinate descent method. The active set sizes for $\mathbf{\Lambda}$ and $\mathbf{\Theta}$ are $m_{\mathbf{\Lambda}}$ and $m_{\mathbf{\Theta}}$, respectively. Also, K is the number of conjugate gradient iterations.

A.2.1 Cost of updating $\mathbf{\Lambda}$

The time complexity of each $\mathbf{\Lambda}$ update is dominated by the cost of precomputing columns of $\mathbf{\Sigma}$ and $\mathbf{\Psi}$. The cost of these precomputations is $O\Big(\big[1 + \frac{B_{\mathbf{\Lambda}}}{q}\big][m_{\mathbf{\Lambda}}Kq + nq^2]\Big)$ where $B_{\mathbf{\Lambda}} = \sum_{z \neq r} |\{j | i \in C_z, j \in C_r, (i,j) \in \mathcal{S}_{\mathbf{\Lambda}}\}|$ is the number of cache misses. Although the worst-case of $B_{\mathbf{\Lambda}} = k_{\mathbf{\Lambda}}q$ requires computing $\mathbf{\Sigma}$ and $\mathbf{\Psi}$ a total of $k_{\mathbf{\Lambda}}$ times, in practice, graph clustering dramatically reduces this additional cost of block-wise optimization. In the best case, when graph clustering identifies perfect block-diagonal structure in the active set, the number of cache misses $B_{\mathbf{\Lambda}} = 0$ and we incur no runtime penalty from limited memory.

A.2.2 Time Cost of Updating $\mathbf{\Theta}$

The overall runtime is dominated by the cost of precomputing columns of $\mathbf{S}_{\mathbf{xx}}$ and $\mathbf{\Sigma}$. The complexity of these operations is $O(m_{\mathbf{\Lambda}}Kq + m_{\mathbf{\Theta}}q + n\tilde{p}B_{\mathbf{\Theta}})$, where $\tilde{p}$ is the number of non-empty rows in



(a) (b)

Figure 1: Results from varying sample size n on chain graph with $p = q = 10{,}000$. (a) Comparison of computation time of different methods. (b) Comparison of edge recovery accuracy as measured by F_1-score.

$\mathbf{\Theta}$ and $B_{\mathbf{\Theta}} = \sum_{i,r} |\{i | j \in C_r, (i,j) \in \mathcal{S}_\Theta\}|$ is the number of cache misses. Without any row-wise sparsity we have $\tilde{p}{=}p$ and $B_{\mathbf{\Theta}}{=}k_{\mathbf{\Theta}}p$, so the worst-case is that $\mathbf{S_{xx}}$ is computed a total of $k_{\mathbf{\Theta}}$ times. This additional cost is substantially reduced in real datasets, where most input variables influence few or none of the outputs. In the best case, if the active set of $\mathbf{\Theta}$ has a block structure, where each group of inputs are influencing only one group of outputs, overall $\mathbf{S_{xx}}$ will be computed only once. We can further save the computation of $\mathbf{S_{xx}}$, if the entire row of $\mathbf{\Theta}$ is not in the active set, by entirely skipping the computation for the corresponding column of $\mathbf{S_{xx}}$.

A.3 Additional Results from Synthetic Data Experiments

We compare the performance of the different algorithms on synthetic datasets with different sample sizes n, using a chain graph structure with $p = q = 10{,}000$. Figure 1(a) shows that our methods run significantly faster than the previous method across all sample sizes. In Figure 1(b) we measure the accuracy in recovering the true chain graph structure in terms of F_1-score for different sample sizes n. At convergence, F_1-score was the same for all methods to three significant digits. As expected, the accuracy improves as the sample size increases.