Large–Scale Sparse Inverse Covariance Matrix Estimation

Matthias Bollhöfer[*] Olaf Schenk[†]

July 12, 2016

Abstract

The estimation of large sparse inverse covariance matrices is an ubitiquous statistical problem in many application areas such as mathematical finance, geology, health, or many others. Numerical approaches typically rely on the maximum likelihood estimation or its negative log-likelihood function. When the Gaussian mean random field is expected to be sparse, regularization techniques which add a sparsity prior such as the l_1-regularization have become popular to address this issue. This leads to a convex but nondifferentiable target function. Recently a quadratic approximate inverse covariance method (QUIC) was proposed. The hallmark of this method is its superlinear to quadratic convergence which makes it among the most competitive methods. In this paper we will present a sparse version of this method and we will demonstrate that using advanced sparse matrix technology, the sparse version of QUIC is easily able to deal with problems of size one million within a few minutes on modern multicore computers.

Keywords. covariance matrix, inverse covariance matrix estimation, sparse matrices, approximate inverse matrices

AMS. 65N55, 65F10, 65N22

1 Introduction

In mathematical statistics one is often faced with the problem that big data sets $y_1, \dots, y_n$, each of them of length $p \gg n$, are given but the underlying distribution is unknown. Even if one assumes that the distribution is Gaussian, i.e., $y_1, \dots, y_n \in \mathcal{N}(\mu, \Sigma)$, the expected value $\mu \in \mathbb{R}^p$ and in particular the covariance matrix $\Sigma \in \mathbb{R}^{p \times p}$ are unknown. Therefore one has to estimate the quantities. This is the typical objective of the maximum likelihood method which attempts to maximize the likelihood function $L(\Theta)$, where $\Theta = \Sigma^{-1}$. For better optimization one usually uses the (negative) logarithm $l(\Theta)$ of the likelihood function $L(\Theta)$.

When the Gaussian Markov random field is sparse, one may modify $l(\Theta)$ such that a sparse Θ is preferred. A common way (cf., e.g., [3, 14, 35]) to enforce sparsity is to regularize $l(\Theta)$ by an l_1 penalty term which then yields some target function $f_\lambda(\Theta)$. Since this regularized function f_λ is convex, there are many approaches from convex optimization to minimize the function. Among these there are blockwise descent methods [3, 10, 14, 28], (inexect) interior point methods [4, 21, 35], alternating linearization [29], iterative thresholding [27], projected subgradients [11], greedy-type descent methods [30], and, more recently, second-order methods [2, 9, 17, 18, 25]. In particular second order methods have become attractive because of their fast convergence.

In this paper we are especially interested in the QUIC method from [17] which has been shown to be among the most competitive methods for minimizing f_λ. This method is based on a second-order Taylor expansion of the differentiable part of f_λ and it has been proved in [17] that this method is superlinear convergent, and experimentally, even quadratic convergence can be observed.

[*]Institute of Computational Mathematics, TU Braunschweig, D-38106 Braunschweig, Germany (m.bollhoefer@tu-bs.de).

[†]Institute of Computational Science, Faculty of Informatics, Universitá della Svizzera italiana, Switzerland (olaf.schenk@usi.ch).

This makes this method particularly attractive, though it is designed for the use of dense matrices Σ and Θ which limits the range of application problem currently to problems in the range of 10^4, which is still very impressive. In [18] a version called BigQUIC has been proposed to deal with large-scale problems. The idea here is to avoid dense matrix computations and to provide a memory–efficient version of QUIC that computes part of the matrices on demand, when needed. Another approach using hierarchical matrices was recently proposed in [2]. In this paper we are going to present a sparse version of the QUIC algorithm where the matrices use compressed sparse column storage and sparse matrix technologies to deal with large-scale problems. This approach is feasible, when beside Θ Σ is also approximately sparse. Our approach is based on sparse (incomplete) Cholesky decompositions for Θ and factorized approximate inverse techniques to compute $W \approx \Theta^{-1}$. Besides, as we will demonstrate, an approximate sparse representation of the empirical covariance matrix associated with the given samples $y_1, \ldots, y_n$ plays a major computational role in designing an efficient sparse QUIC method.

The paper is organized as follows. In Section 2 we give a short summary about the mathematical problem of sparse inverse covariance estimation and its formulation as a convex optimization problem. In Section 3 we briefly review the QUIC method; after that we give in Section 4, the three major challenges with respect to the numerical kernel of this method as well as existing approaches to deal with these. It will present for each of these problems two separate approaches using state-of-the-art sparse matrix techniques. These will be demonstrated in Section 5 using some large-scale sparse inverse covariance estimation examples. We will demonstrate that on a modern computer with a single node and 60 cores we are easily able to solve these problems within a few minutes.

2 Sparse inverse covariance estimation

In many application problems one is often faced with the following problem: given n data samples $y_1, \ldots, y_n \in \mathcal{N}(\mu, \Sigma)$ from a p-variate Gaussian distribution with covariance matrix Σ and mean value μ, the only information that we have are the random samples but we would like to know Σ or even $\Theta = \Sigma^{-1}$ which are not known to us. We will assume throughout the paper that $p \gg n$. This situation arises quite frequently in big data problems and just to increase the number of samples to construct Σ is not feasible. The usual approach in mathematical statistics is the maximum likelihood method. We define by

$$Y := \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix}$$

the matrix of matrix samples and by

$$\hat{\mu} = \frac{1}{n} \sum_{j=1}^{n} y_i, \; S = \frac{1}{n} \sum_{i=1}^{n} [Y - \mu] [Y - \mu]^T$$

the sample arithmetic mean $\hat{\mu}$ and the associated sample covariance matrix S (here we use $\frac{1}{n}$ rather than $\frac{1}{n-1}$ in the covariance matrix for simplicity). The difference $Y - \mu \equiv Y - \mu \; (1 \cdots 1)$ is considered to be taken from each column of Y. The likelihood function is given by

$$L(\Theta) = \left[\frac{\det \Theta}{2\pi} \right]^{\frac{n}{2}} \cdot \exp \left(-\frac{1}{2} \operatorname{tr}([Y - \hat{\mu}]^T \Theta [Y - \hat{\mu}]) \right).$$

As usual in maximum likelihood estimation one takes the logarithm and, to turn this into a minimization problem, the sign is flipped and additive and multiplicative constants are omitted. This leads to the minimization of the (negative) log-likelihood function

$$g(\Theta) = -\log(\det \Theta) + \operatorname{tr}(S\Theta). \tag{1}$$

To enforce sparsity of Θ one usually adds a sparsity prior to g. Requiring that Θ has to be sparse can be read as the associated Gaussian Markov random field is sparse. This sparsity constraint yields an l_1-regularized function

$$f_\lambda(\Theta) = g(\Theta) + \lambda|\Theta|_1, \tag{2}$$

where $\lambda > 0$ is an a priori chosen parameter and $|\Theta|_1 = \sum_{i,j=1}^{p} |\theta_{ij}|$ refers to the elementwise 1-norm. The constrained minimization of g (resp., f_λ) is also referred to as Lasso-type problem and since g is strictly convex and f_λ is still convex, there exist several optimization methods to minimize f_λ such as block-wise coordinate descent methods (graphical lasso) [3,10,14,28], (inexact) interior point methods [21,35], alternating linearization [29], iterative thresholding [27], projected subgradients [11], and greedy-type descent methods [30]. These approaches have in common that they are first-order methods. More recently, second order have been proposed such as the Newton-like method in [25] or quadratic approximation methods [17]; the latter has led to the so–called QUIC method which we will briefly describe in the next section.

3 The QUIC algorithm

The basis of the QUIC method [17] consists of locally constructing a second-order approximation for the differentiable part g of f_λ using Taylor expansion. For fixed Θ, the local quadratic approximation $\tilde{g}(\Delta)$ of $g(\Theta + \Delta)$ reads as

$$g(\Theta + \Delta) \approx \tilde{g}(\Delta) = \operatorname{tr}((S - W)\Delta) + \frac{1}{2}\operatorname{tr}(W\Delta W\Delta) - \log(\det\Theta) + \operatorname{tr}(S\Theta),$$

where $W = \Theta^{-1}$. Up to a constant, this yields a local approximation

$$h(\Delta) \equiv \operatorname{tr}((S - W)\Delta) + \frac{1}{2}\operatorname{tr}(W\Delta W\Delta) + \lambda|\Theta + \Delta|_1$$

of $f_\lambda(\Theta + \Delta)$. Rather than minimizing h for all Δ, the authors have proposed to apply a sequence of one-dimensional minimization steps of type

$$h(\Delta + \mu(e_i e_j^T + e_j e_i^T)),$$

where Δ refers to the already completed updates, e_i, e_j refer to suitably chosen unit vectors and μ is the parameter to be computed. Interestingly, it has been shown in the same article that it suffices to select the sequence of indices $(i_1, j_1), \ldots, (i_k, j_k)$ only from those entries (i, j) such that $|s_{ij} - w_{ij}| \geqslant \lambda$ or $\theta_{ij} \neq 0$. A quite realistic expectation is that this set of indices is usually significantly less than p^2. Each one-dimensional step (i, j) requires, in particular, the values of s_{ij} and w_{ii}, w_{jj} and w_{ij} as well as the ith and jth columns of W. Moreover, Δ and θ_{ij} are required. At this point we skip presenting the detailed formula for computing μ and kindly refer to [17] for further details. Once the complete sequence is computed, the collection Δ of all one-dimensional steps is used to update Θ by $\Theta' = \Theta + \alpha\Delta$. Here α is chosen as 2^{-m} and α is reduced until Θ' is positive definite and the associated f_λ satisfies an additional Armijo-type criterion.

Without going into further details of the QUIC code, it is obvious that the following tasks are part of the algorithm:

1. The empirical covariance matrix S is referenced for every (i, j) from the sequence, this includes in particular (i, j) such that $|s_{ij}| > \lambda$, e.g., when W is diagonal.
2. In order to verify whether Θ' is positive definite or not, an algorithm is required to test the positive definiteness of Θ'.
3. The computation of $f_\lambda(\Theta)$ requires a method for computing $\log(\det\Theta)$.
4. Finally, for setting up the active set $(i_1, j_1), \ldots, (i_k, j_k)$ the entries of $W = \Theta^{-1}$ are required, in particular, for detecting $|s_{ij} - w_{ij}| > \lambda$, but also for computing each one-dimensional update. The latter requires each column w_i, w_j for computing μ for every (i, j) from the active set sequence.

We will next describe how these numerical challenges are treated by existing algorithms.

4 Large-scale challenges

The original QUIC algorithm is designed to work with dense matrices, therefore the sample covariance matrix S is directly passed as a dense matrix to the algorithm, the positive definiteness as well as $\log(\det \Theta)$ are computed via the dense Cholesky decomposition. Using the dense Cholesky-decomposition, $W = \Theta^{-1}$ is easily inverted. This numerical core part is performed using LAPACK and BLAS.

More recently, in [18] a large-scale version BIGQUIC of the QUIC algorithm has been presented with the major objective to save memory and to deal with a million variables. The hallmark of the BIGQUIC algorithm is to avoid memory consumption and therefore the $\log(\det \Theta)$ is computed via a recursion formula [18] which allows to compute the determinant by reducing it to solving linear systems and to check positive-definiteness. Similarly, W is not computed in total but on demand using the conjugate gradient method. In addition, the entries of S are only computed when needed. To improve efficiency, a further blocking strategy is applied to the sequence of one-dimensional updates in order to recycle the computed quantities more often.

In [2] a version of the QUIC algorithm using hierarchical matrices is presented. Here the major idea is to represent all matrices in $\mathcal{H}$ format, to compute the Cholesky decomposition and the inverse matrix using $\mathcal{H}$ matrix arithmetic.

We will now present our approach to deal with the QUIC method for large-scale systems. We will present numerical methods that allow for the use of state-of-the-art sparse matrix technology. These are employed to efficiently deal with the following tasks:

1. (Approximate) sparse representations of the large entries of the empirical covariance matrix S.
2. Detect the positive definiteness of Θ and $\log(\det \Theta)$ using a sparse (approximate) Cholesky decomposition.
3. Compute a sparse approximate inverse matrix W.

To be efficient, these tasks certainly require that the underlying statistical problem possesses certain sparsity properties, e.g., the Gaussian Markov random field (i.e. Σ^{-1}) is assumed to be sparse, but in addition we certainly need $W \approx \Sigma$ to be at least approximately sparse and that the entries $|s_{ij}| \geqslant \lambda$ can be represented by a sparse matrix. Whenever this is fulfilled, sparse matrix technologies can be efficiently applied as we will demonstrate in the following.

4.1 Sparse Representation of the Sample Covariance Matrix

Given the initial statistical data $Y = [y_1, \ldots, y_n]$ and its mean value $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} y_i$, let us recall that S is formally given by

$$S = \frac{1}{n} Z Z^T, \text{ where } Z = Y - \hat{\mu}$$

and the difference $Y - \hat{\mu}$ is understood to be taken by columns. Obviously, in our case S is large-scale and low-rank (since $p \gg n$), S is a symmetric positive semidefinite matrix and, theoretically, when $n \to \infty$ we would have $S \to \Sigma$. Certainly we will by far not have n large enough to see this convergence. Therefore, even if Σ were approximately sparse it does not mean that S has to be approximately sparse as well, but S could have a significant number of entries that have to be considered as noise. However, taking into account only the entries s_{uv} such that $u = v$ or $|s_{uv}| \geqslant \lambda$ (e.g., $\lambda = 0.5$) gives hope that at least these entries can be represented by a sparse matrix. We also like to emphasize that usually it is not known a priori at which positions the large entries are located. This certainly makes it harder to develop an efficient algorithm for an approximate sparse representation. Taking all this into account we will now present two algorithms to compute an initial sparse representation $\tilde{S}$ of all s_{uv} such that $u = v$ or $|s_{uv}| \geqslant \lambda$. To simply compute these entries does not interfere with computing some more entries of S on request, e.g., the computation of the active set requires to compare $|s_{uv} - w_{uv}|$ for all nonzero entries of W and certainly, given the pattern of W we could easily compute S at the nonzero pattern of W, if not yet present. In this case, the pattern of where to compute entries of S is predetermined by W and we do not have to further guess entries of S outside the pattern of W, except those cases where $|s_{uv}| \geqslant \lambda$, which remains the major difficulty.

The first algorithm to compute $\tilde{S}$ is straightforward. The product $\frac{1}{n}ZZ^T$ can be easily computed using level-3-BLAS. Since the amount of memory for computing this is considerably high, we compute this product in chunks of size k, i.e., we set $Z^T = [C_1, \dots, C_m]$, where $C_1, \dots, C_m \in \mathbb{R}^{n,k}$. Possibly C_m has fewer columns $p - (m-1)k \leqslant k$ if p is not a multiple of k.

Algorithm 1 Deterministic Computation of $S \geqslant \lambda$

Require: $Y \in \mathbb{R}^{p,n}$, $\lambda > 0$, $k \in \mathbb{N}$.
Ensure: sparse restriction $\tilde{S} \in \mathbb{R}^{p,p}$ of S s.t. $|s_{uv}| \geqslant \lambda$ or $u = v$

```
1: $\hat{\mu} := \frac{1}{n}\sum_{j=1}^{p} y_j$, $Z := Y - \hat{\mu}$, partition $Z^T = [C_1, \dots, C_m]$ s.t. $C_j \in \mathbb{R}^{p,k}$
2: for $j = 1 : m$ do
3:     denote by $\tilde{Z} = (z_{uv})_{u>(j-1)k,v}$ the block lower triangular part of $Z$
4:     compute $D_j = \frac{1}{n}\tilde{Z}C_j$.
5:     for $i = 1 : k$ do
6:         sparsify $i$th column of $D_j$ s.t. only $s_{uu}$ and $|s_{uv}| \geqslant \lambda$ are saved to $\tilde{S}$
7:     end for
8: end for
```

In practice, we will use $k = 256$ for simplicity to compute a sufficiently large chunk of S. Certainly, different values of k were possible and we did not investigate which size k would lead to an optimal computation time. We sketch the computation of $\tilde{S}$ via Algorithm 1 in Figure 1.

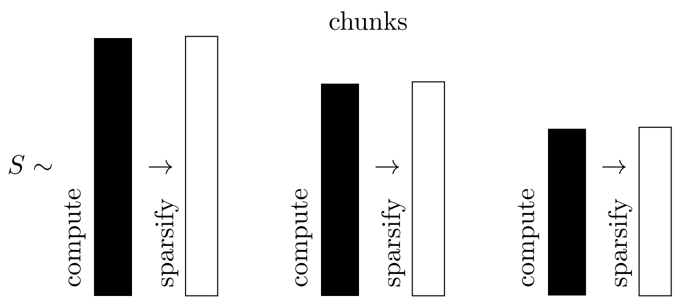


Figure 1: Sketch of Algorithm 1.

We like to note that Algorithm 1 can be easily parallelized with a large number of cores c. But even on a high-performance system the total amount of computation time remains on the order of $\mathcal{O}(\frac{p^2 n}{c})$, maybe with a small constant if all architecture-dependent properties are used and we do not see that in general a deterministic algorithm can significantly reduce the complexity. We therefore present a second random-based algorithm which may in practice consume less time, in particular when the size p is getting large.

To break the complexity $\mathcal{O}(p^2 n)$, as a first step we will make use of a column compression technique. This approach is motivated by probing techniques [6–8, 34] to efficiently compute sparse matrices by matrix-vector products using significantly fewer vectors than the size p of the matrix. Here the idea is relatively easy when the pattern of the underlying matrix is known as indicated in the following trivial pattern example of Figure 2. Now there are the two following difficulties with the probing approach in our case:

1. We do not know the pattern of S in advance.

$$\begin{pmatrix} * & * & & & & & & & \\ * & * & * & & & & & & \\ & * & * & * & & & & & \\ & & * & * & * & & & & \\ & & & * & * & * & & & \\ & & & & * & * & * & & \\ & & & & & * & * & * & \\ & & & & & & * & * & * \\ & & & & & & & * & * \end{pmatrix} \begin{pmatrix} 1 \\ \\ \\ \\ 1 \\ \\ \\ \\ 1 \end{pmatrix} = \begin{pmatrix} * \\ * \\ \\ * \\ * \\ * \\ \\ * \\ * \end{pmatrix}$$

Figure 2: Column compression $Sg = d$ via probing.

2. The matrix S is usually not sparse. This is not a contradiction to our assumption that the covariance matrix Σ itself is approximately sparse. Indeed, the number n of samples used to form S is usually much less than its size p.

We now describe the main idea of a heuristic algorithm for computing S using probing vectors. Some technical details we will describe after that. In order to deal with the first difficulty we randomly choose l numbers from $\{1, \ldots, p\}$ and denote this set by $\mathcal{J}$. Then we set

$$g = \sum_{v \in \mathcal{J}} e_v, \tag{3}$$

where e_v denotes the v–th unit vector and compute $d = Sg$. Choosing the entries of $\mathcal{J}$ randomly, there is a good chance that for $v \in \mathcal{J}$ the associated columns Se_v of S do not overlap, it least if S were sparse. Next we will deal with the second difficulty. Even if on the average, the t largest entries in each column $v \in \mathcal{J}$ of S do not overlap, noisy entries will surely accumulate as we increase the size l of $\mathcal{J}$. In particular since $n \ll p$, S is far away from Σ, therefore the noise is likely to add up to contributions greater than λ when l gets larger. For this reason we simply sort the entries of $d = Sg$ in modulus in decreasing order and only keep the $F \cdot t \cdot l$ largest entries, where $F > 1$ refers to some failure factor, allowing more entries than we expect to be greater than λ. These remaining $F \cdot t \cdot l$ entries of d are associated with some index set $\mathcal{I}$ and the only thing we have to do now is to recompute s_{uv}, for all $u \in \mathcal{I}$ and $v \in \mathcal{J}$ to cross–check which of these entries really satisfy $|s_{uv}| \geqslant \lambda$. These entries are then kept and build the approximate empirical covariance matrix $\tilde{S}$.

After having given a sketch of the major idea, we will now comment on some details of this probing method. Although we certainly do not know t in advance, we can start with a relatively pessimistic large initial guess for t. While computing columns of S step by step, we uncover more and more entries s_{uv} such that $|s_{uv}| \geqslant \lambda$. This allows to adapt t throughout the computation. Similarly, starting with an initial guess F we can easily compare the number of entries s_{uv} that were successfully computed with the number of entries that were left over after sorting. This also allows to adaptively modify F. It is also clear that analogous to Algorithm 1 we can compute multiple columns $G = [g_1, \ldots, g_k]$ simultaneously to exploit dense linear algebra kernels. Since each g_i in (3) is a sum of l unit vectors, the formal product $C = Z^T G = \left(Z^T g_i\right)_{i=1,\ldots,k}$ is easily achieved for each i summing up only those columns v of Z^T such that $v \in \mathcal{J}_m$. Algorithm 2 states the major frame of the randomized computation of the sparsified sample covariance matrix. The adjustment of the parameter l has not yet been discussed. This will be done now based on a simplified cost model.

For the computational cost of Algorithm 2 we initially note that the formal product $C = Z^T G$ costs $\mathcal{O}(nl \cdot k)$ during a single loop since we exploit the special pattern of G. Thus this product is significantly cheaper than computing $D = \frac{1}{n} ZC$ which costs $\mathcal{O}(pn \cdot k)$ locally. From this one can immediately conclude that the computation of D using dense linear algebra kernels (level-3-BLAS) is dominating the computation time up to step 12. Assuming that l is a constant, the total matrix–matrix product D accumulated over the outer while–loop costs $\mathcal{O}(\varepsilon pn \cdot \frac{p}{l})$ which roughly

Algorithm 2 Randomized Computation of $S \geqslant \lambda$ Using Column Compression

```
Require: $Y \in \mathbb{R}^{p,n}$, $\lambda > 0$, $k \in \mathbb{N}$.
Ensure: sparse restriction $\tilde{S} \in \mathbb{R}^{p,p}$ of $S$ s.t. $|s_{uv}| \geqslant \lambda$ or $u = v$
 1: $\hat{\mu} := \frac{1}{n}\sum_{j=1}^{p} y_j$, $Z := Y - \hat{\mu}$, $\mathcal{C} = \{1, \ldots, p\}$
 2: compute $s_{uu}$, $u = 1, \ldots, p$.
 3: while $\mathcal{C} \neq \emptyset$ do
 4:   $G = [g_1, \ldots, g_k] = 0 \in \mathbb{R}^{p,k}$
 5:   {for each column $i$ pick $l$ unused indices randomly:}
 6:   for $i = 1 : k$ do
 7:     $\mathcal{J}_i = \emptyset$
 8:     for $j = 1 : l$ do
 9:       pick $r \in \mathcal{C}$, $\mathcal{C} = \mathcal{C} \setminus \{r\}$, $\mathcal{J}_i = \mathcal{J}_i \cup \{r\}$
10:       $g_i = g_i + e_r$
11:     end for
12:   end for
13:   {compute $k$ compressed columns of $S$:}
14:   set $C = Z^T G$ and compute $D = \frac{1}{n} ZC$ and let $D = [d_1, \ldots, d_k]$
15:   {for each compressed column $k$ detect the neighbouring structure:}
16:   for $i = 1 : k$ do
17:     {filter $d$ s.t. only the largest entries remain:}
18:     exclude the elements of $\mathcal{J}_i$ from $d_i$
19:     sort the remaining entries of $|d_i|$ in decreasing order
20:     keep the largest $F \cdot t \cdot l$ in modulus and denote the associated indices by $\mathcal{I}_i$
21:     {for each large off-diagonal index $u$ search for one associated column $v$:}
22:     $\hat{\mathcal{J}}_i := \mathcal{J}_i$
23:     for $u \in \mathcal{I}_i$ do
24:       for $v \in \hat{\mathcal{J}}_i$ do
25:         recompute the exact value $s_{uv}$
26:         if $|s_{uv}| \geqslant \lambda$, store $s_{uv}$, remove $v$ from $\hat{\mathcal{J}}_i$ and stop as soon as $t$ entries are detected in
            column $v$ of $S$.
27:       end for
28:     end for
29:   end for
30:   adjust $t, F, l$
31: end while
```

reads as compressing l columns simultaneously and ε is some small constant (e.g., 10^{-2}) that takes into account the high performance of the dense linear algebra kernel. The recomputation of s_{uv} in steps 16-29 locally costs $\mathcal{O}(Ftl^2 n \cdot k)$ which results in an overall cost of $\mathcal{O}(Ftl^2 n \cdot \frac{p}{l})$.

After we have motivated a simple cost model for the matrix–matrix computation and the recomputation of s_{uv}, we use these simplified models to define an equilibrated value of l. Assuming that F, t, ε are constant, the optimal performance is achieved whenever the two most time-consuming parts coincide, i.e., whenever we have

$$\varepsilon pn \cdot \frac{p}{l} = Ftl^2 n \cdot \frac{p}{l},$$

which is satisfied for choosing $l = \sqrt{\frac{\varepsilon p}{Ft}}$. We will use this formula for l in Algorithm 2. In this ideal scenario the total amount of computation time would cost $\mathcal{O}(p^{3/2} n \sqrt{Ft\varepsilon})$ which is much cheaper than $\mathcal{O}(p^2 n)$ and there is hope that for large–scale problem sizes p, the randomized Algorithm 2 with column compression can outperform the deterministic Algorithm 1.

We finally like to point out that the computation of chunks of columns of S with or without column compression is easily performed using multithreaded level-3-BLAS. Similarly, the sparsification of each computed column will be done in parallel using OpenMP.

After we have discussed how the first (and major) obstacle of deriving a sparse and approximate representation of S is performed, we will next discuss the second part which consists of computing an (approximate) factorization of Θ as computed iteratively in the minimization process of $f_\lambda(\Theta)$.

4.2 Sparse matrix factorization

In this subsection we will discuss, given an update $\Theta' = \Theta + \alpha\Delta$, how to detect that Θ' is still positive definite. Provided that Θ' is symmetric and positive definite and satisfies Armijo-type criterion with respect to the decrease of $f_\lambda(\Theta')$, we discuss how to compute $\log(\det \Theta')$. Here we will concentrate on two variants based on sparse matrices. The first algorithm is simply computing the Cholesky decomposition for Θ' and returning an error message if it fails. Among many numerical software packages that allow for fast sparse Cholesky decompositions (cf. e.g. [5, 16, 19, 26, 31]), we decide to use the CHOLMOD [5] factorization which is obtained by default when using the `chol` MATLAB's function. We note that this algorithm is sequential, however it uses level-3-BLAS and therefore includes some multithreading on that level which mildly improves its parallel performance.

As an alternative to a sparse Cholesky decomposition we use an incomplete LDL^T factorization following the ideas from [15]. The main motivation for using an incomplete factorization here is to save memory rather than to decrease computation time. Indeed, modern sparse direct methods use a deep machinery of technologies which makes it hard to beat these kind of methods, except if the incomplete factorization produces factors with drastically less fill–in. But the latter may save memory which could become a significant issue in sparse inverse covariance matrix estimation. For the incomplete LDL^T approach, symmetric maximum weight matchings [12,13] are performed in a preprocessing step in order to improve the diagonal dominance followed by a fill-reducing ordering on the compressed graph such as [1, 20]. Finally a left–looking approximate LDL^T factorization with 1×1 and 2×2 pivots is performed similarly to [22]. Certainly, for symmetric positive definite matrices, this amount of work were not necessary, but since $\Theta' = \Theta + \alpha\Delta$ is not guaranteed to be positive definite, we prefer to use an indefinite approach. It is clear that in the simplest cases, checking whether the diagonal entries of Θ' are at least positive, one can easily skip (incomplete) factorizations once this property is violated. Otherwise, positive definiteness can be read off from D. We are aware that using a drop tolerance, the information could be unsafe due to dropping, however, similar to [32] we did not observe this in our experiments. This may be caused by the choice of our drop tolerance τ. To be precise, define

$$\rho := |f_\lambda(\Theta') - f_\lambda(\Theta)|/|f_\lambda(\Theta')| \tag{4}$$

as the relative error between subsequent QUIC iteration steps. Then we use $\tau = 0.1\rho$, i.e., for safety reasons τ is chosen one order of magnitude less than the relative accuracy ρ. To choose τ one order of magnitude less is also intended to prevent the target function f_λ from being perturbed too much. In order to avoid extreme values of $|f_\lambda(\Theta') - f_\lambda(\Theta)|/|f_\lambda(\Theta')|$, we also make sure that ρ is always chosen such that $\rho \leqslant 10^{-1}$ is the maximum tolerance and, at the other bound, we use $\rho \geqslant \text{tol}$, where tol is the user–defined accuracy as passed to the QUIC method (in our experiments we will use the default value $\tau = 10^{-6}$).

In total we like to point out that both approaches could be uniformly represented by

$$\Pi^T QAQ\Pi \approx LDL^T, \tag{5}$$

where Π is a suitable permutation matrix, Q is a diagonal scaling matrix, L is unit triangular, and D is (block) diagonal with diagonal entries of size 1×1 or 2×2. Once, D is discovered to be positive definite we could reduce it to a (scalar) diagonal matrix and in this case we easily obtain $\log(\det \Theta') = \sum_{i=1}^{p} (\log d_{ii} - 2\log q_{ii})$ as by-product of the Cholesky decomposition.

4.3 Sparse approximate inverse representation

As a last step to introduce sparse matrix computation into the QUIC algorithm we will discuss two approaches to approximately compute $W \approx \Theta^{-1}$ during the iterative minimization of $f_\lambda(\Theta)$.

The first approach which we will discuss simply utilizes the given factorization (5). Having an (approximate) Cholesky decomposition available we certainly reuse the given factorization in order to compute an (approximate) inverse $W \approx \Theta^{-1}$, which we will discuss next. Given some tolerance ε we can approximately compute

$$A^{-1} \approx Q\Pi L^{-T} D^{-1} L^{-1} \Pi^T Q$$

by setting $L = I - E$ and writing the inverse of L as a Neumann series $L^{-1} = I + E + E^2 + \cdots + E^{p-1}$. Using Horner's scheme and the tolerance ε we successively compute

$$^{i}L_1 = I + E, {}^{i}L_{k+1} = {}^{i}L_k \; E + I, \; k = 1, 2, 3, \ldots.$$

We define $\varepsilon := 0.1\rho$ with ρ from (4). In each step k we can sparsify the columns of ${}^{i}L_k$ by using a finer tolerance 0.1ε and we stop the expansion as soon as the element-wise error between the elements of two neigboring polynomials ${}^{i}L_{k+1} - {}^{i}L_k$ drops below the tolerance ε. Finally we use

$$A^{-1} \approx {}^{i}A = Q\Pi \, {}^{i}L^T \; D^{-1} \, {}^{i}L \; \Pi^T Q,$$

say with some relative dropping $|{}^{i}a_{uv}| \leqslant \varepsilon \; {}^{i}a_{uu} \; {}^{i}a_{vv}$ to build the final approximation ${}^{i}A$. The beneficial property of the Neumann-based approach is its ease and its simplicity that allow for straightforward parallelization. This is because the successive computation using Horner's scheme can easily be performed in parallel using OpenMP on all columns given k. The intermediate sparsification with smaller tolerance 0.1ε is intended to prevent the Neumann series from producing too much fill-in. On the downside all entries are eventually computed up to some tolerance ε and the algorithm to compute the sequence of one-dimensional updates $h(\mu) = h(\Delta + \mu(e_i e_j^T + e_j e_i^T))$ may theoretically lead to inaccurate updates.

As an alternative approach to compute an approximate inverse, we now present a second approach. Here the idea is first to compute an accurate inverse at those positions where necessary, and then later on to fill-up the remaining positions by a less accurate inverse such that the inverse is accurate enough to meet the conditions of the sequence of one-dimensional minimization steps. Looking at the strategy to select the active set $(i_1, j_j), \ldots, (i_k, j_k)$ the sequence of indices is chosen by $\theta_{ij} \neq 0$ or $|s_{ij} - w_{ij}| \geqslant \lambda$. This certainly forces initially to choose (i, j) such that $|s_{ij}| \geqslant \lambda$ whenever $w_{ij} = 0$. Assuming that $w_{ij} \neq 0$ if $\theta_{ij} \neq 0$, it is therefore necessary to have the entries of W precisely at those positions where $\theta_{ij} \neq 0$. As consequence of the optimization process, after one iteration we will already have that $\theta_{ij} \neq 0$ if $|s_{ij}| \geqslant \lambda$. Therefore we expect that the pattern of $|S|$ subject to λ is included in the pattern of Θ (at least after the first iteration step) and to find a new active set, it is likely that it is sufficient to have W available for the pattern of Θ or, say, at the pattern of its Cholesky factor. This observation leads directly to the idea of using a selective inverse [23,24,33] rather than an approximate inverse. In the symmetric case, the selective inverse is easily explained as follows. Let

$$A = LDL^T, \text{ where } L = \begin{pmatrix} I & 0 \\ L_E & I \end{pmatrix}, \; D = \begin{pmatrix} D_B & 0 \\ 0 & D_C \end{pmatrix}.$$

From this it follows that

$$(6) \qquad (LDL^T)^{-1} = \begin{pmatrix} D_B^{-1} + L_E^T(D_C^{-1}L_E) & -L_E^T D_C^{-1} \\ -D_C^{-1}L_E & D_C^{-1} \end{pmatrix} \approx \begin{pmatrix} D_B^{-1} + L_E^T G_E & -G_E^T \\ -G_E & G_C \end{pmatrix},$$

where $G_C = D_C^{-1}$ and G_E coincides with $D_C^{-1}L_E$ only in those rows that are required to compute $L_E^T \; (D_C^{-1}L_E)$ accurately. This forces the equality in (6) everywhere in the $(1,1)$ and $(2,2)$ block and selectively in those rows of the $(2,1)$ block (resp. $(1,2)$ block), where G_E is computed. Applying this approach successively from the lower right corner to the upper left corner yields the exact inverse at selected positions, at least in the case of a direct solver (this can be verified using the notion of the elimination tree). We like to point out that computing the selective inverse is on a comparable order to the computational cost computing the Cholesky decomposition having the same fill–in. Once the selective inverse is computed, we decide how to select the active set $(i_1, j_1), \dots, (i_k, j_k)$ based on the computed selective inverse $W \approx \Theta^{-1}$. After the set is defined, we sparsify W back to the diagonal entries and active set $(i_1, j_1), \dots, (i_k, j_k)$. Since now we can compute the subgradient, we compute a refined approximate inverse W using again the Neumann series, but only using a different threshold $\hat{\varepsilon} = 0.1\hat{\rho}$ which we will briefly explain in the following. Following [18], we define the subgradient via

$$\nabla^S_{ij} f_\lambda(\Theta) := \begin{cases} s_{ij} - w_{ij} + \operatorname{sign}(\theta_{ij})\lambda & \text{if } \theta_{ij} \neq 0; \\ \operatorname{sign}(s_{ij} - w_{ij}) \max(|s_{ij} - w_{ij}| - \lambda, 0) & \text{if } \theta_{ij} = 0. \end{cases} \tag{7}$$

Theorem 2 in [18] states that the approximate Hessian has to approximate the exact Hessian up to $\mathcal{O}(|\nabla^S_{ij} f_\lambda(\Theta)|_1)$ in order to obtain superlinear to quadratic convergence. Note that the entries of the Hessian refer to the entries $w_i^T \Delta w_j$, where w_i, w_j correspond to the ith and jth columns of W. Assuming that $|\Delta|_1 = \mathcal{O}(|\Theta|_1)$, a natural bound $\hat{\rho}$ for the entries of W would be

$$\hat{\rho} := |\nabla^S_{ij} f_\lambda(\Theta)|_1 / |\Theta|_1 \tag{8}$$

in order to ensure that the entries of the approximate Hessian $w_i^T \Delta w_j$ are sufficiently accurate.

We conclude this subsection mentioning that for both approaches the Neumann series is parallelized using OpenMP.

5 Numerical experiments

A Matlab implementation of the sparse QUIC algorithm was developed to illustrate the robustness of the approach and examine its practical nature. We begin by discussing a few implementational issues in chronological order of the steps of the algorithm and then describe its performance on a varied set of test problems. We will demonstrate that using modern sparse matrix technologies we are able to extend the QUIC method easily to sparse large-scale problems computing the solution within a reasonable amount of time. In our numerical experiments we will compare the QUIC method [17], the BigQUIC method [18] as well as our sparse implementation of QUIC for which we will use the abbreviation SQUIC.

The numerical experiments are carried out on a single node with 1 TB main memory and 4 Intel Xeon E7-4880 v2 @ 2.5 GHz processors each of them having 12 cores on a socket leading to 60 cores in total. Each approach uses all 60 cores, in particular the multithreaded BLAS as used in MATLAB will make use of them. Likewise, implementation in OpenMP of parts of the algorithms as outlined for BigQUIC in [18] and described in the previous sections for SQUIC will make use of 60 cores.

We will conduct three experiments. In these experiments we will prescribe the exact solution Σ^{-1} for testing and use fixed sample size $n = 500$ whereas the size p of the covariance matrix will vary over several orders of magnitude, $p = 10^k$, $k = 2, 3, 4, 5, 6$. As default tolerance all algorithms will use $\epsilon + 10^{-6}$ which is the default value for QUIC. We allow BigQUIC to use up to 80 GB of memory (default was 8 GB). This did not have a major influence on BigQUIC in our experiments.

Example 1 *We will use the tridiagonal matrix* $\Sigma^{-1} = \text{trid}[-0.5, 1.25, -0.5]$ *and we will vary* λ *by* $\lambda = 1.0, 0.6, 0.5, 0.4, 0.3$. *We like to point out that by default the QUIC method uses* $\lambda = 0.5$ *and that the precise choice of an optimal* λ *would be worth discussing in a separate paper. For this example, following [2, 17],* $\lambda = 0.5$ *results in the best recovery rate.*

Example 2 *We will use the pentadiagonal matrix*

$$\Sigma^{-1} = \text{band}[-0.25, -0.25, 1.25, -0.25, -0.25]$$

and we will vary λ *by* $\lambda = 1.0, 0.6, 0.5, 0.4, 0.3, 0.2$. *Beside the default value* $\lambda = 0.5$ *of the QUIC method we like to note that* $\lambda = 0.3$ *yields the best success rate.*

Example 3 *As a final example we use a random example*[1] *from [2]. Here Σ^{-1} is generated randomly having approximately 4 nonzeros per row such that 99% of the entries are clustered in blocks of size 20. Similary to [2], we use a relatively small value for λ, i.e., we set $\lambda = 0.03$.*

Following [2, 4] we use the success rate F as a measure of how well the inverse covariance matrix Σ^{-1} is recovered by Θ from the variants of the QUIC method. To be precise, we denote by TP the number true positive entries in Θ, FP refers to the number of false positive entries in Θ. Similarly, TN denotes true negative entries and FN the false negative entries. Based on this notation we define

$$\text{precision } P = \frac{\text{TP}}{\text{TP}+\text{FP}}, \text{ recall } R = \frac{\text{TP}}{\text{TP}+\text{FN}},$$

and finally

$$\text{success } F = \frac{2PR}{P+R} \in [0,1].$$

We will only plot F and obviously the larger the F the more successful is the recovery of Σ^{-1} by Θ as computed by the variants of the QUIC method.

For our sparse implementation of the QUIC method we have in total three major templates to be implemented in sparse arithmetic, as described previously. For each of these objectives we have presented two alternatives. Let us summarize them as follows:

1. Generation of the empirical covariance matrix S:
   (a) deterministic approach;
   (b) Randomized approach using column compression.
2. Positive definiteness of Θ and $\log(\det \Theta)$:
   (a) Sparse Cholesky decomposition;
   (b) incomplete LDL^T decomposition using the tolerance $\tau = 0.1\rho$, where ρ is defined via (4).
3. Approximate inverse $W \approx \Theta^{-1}$ using the given factorization:
   (a) truncated Neumann series using relative error $\varepsilon = 0.1\rho$.;
   (b) selective inversion, sparsification to active set and refined inverse using Neumann series and the threshold $\hat{\varepsilon} = 0.1\hat{\rho}$. with $\hat{\rho}$ from (8)

In order to distinguish between these 8 variants of SQUIC, we will denote them as SQUIC(aaa), etc., to indicate which method is chosen.

5.1 Tridiagonal example

As a first example we consider the case when Σ^{-1} is tridiagonal in Example 1. Note that since Σ^{-1} is strongly diagonal dominant, the elements σ_{ij} of Σ tend to be small the larger $|i-j|$. This problem allows for a sparse QUIC approach and we will demonstrate the effectiveness of sparse matrix technologies. Figure 3 compares the computation time when using $\lambda = 0.5$ (which is considered to be almost optimal for this problem). Only the SQUIC variants can be applied to $p = 10^5$, $p = 10^6$. Notice the double-logarithmic scalings in Figure 3! Already for $p = 10^4$ the sparse QUIC versions are faster by two orders of magnitude than QUIC or BigQUIC. The SQUIC implementations are easily able to handle the problem for $p = 10^6$. To see the difference between the SQUIC versions more precisely we only display a window between $p = 10^5$ and $p = 10^6$ in Figure 4. Here one can easily see that the SQUIC(b**)[2] implementations are up to twice as fast as the SQUIC(a**) implementations.

[1]We like to thank Jonas Ballani for providing the code for generating Σ^{-1}.

[2]Here * is used for either a and b.

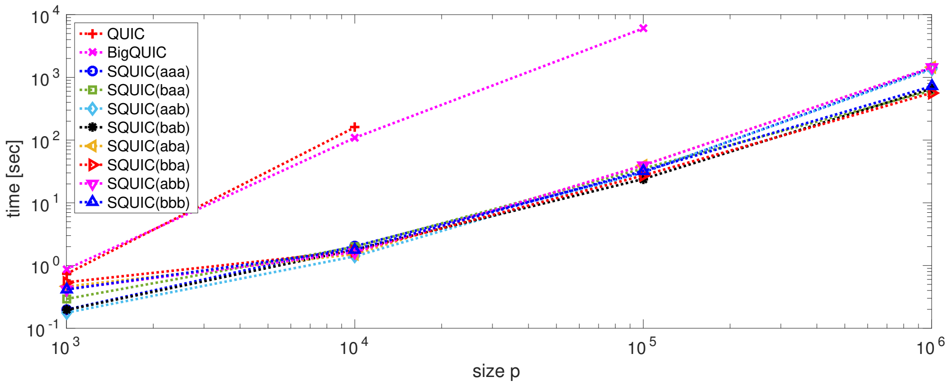


Figure 3: Computation time QUIC-type algorithms, Tridiagonal example 1.

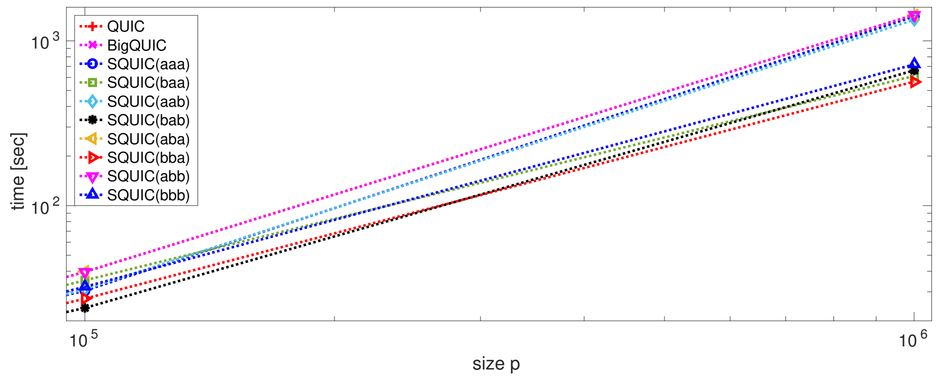


Figure 4: Detailed computation time SQUIC algorithms, Tridiagonal example 1.

Next we state the success rate for all competing algorithms for $p = 10^4$, when QUIC and BigQUIC are still part of the competition. Here Figure 5 shows that QUIC and all variants of SQUIC are more or less identical, except BigQUIC whose recovery rate is initially slightly behind but catches up as λ becomes less. Note also that we have reversed the horizontal scaling, since numerically it seems to be much more natural to start with a large λ and then decrease it. We note that our previous experiments in Figure 3, 4 refer to the specific value $\lambda = 0.5$ and we would like to point out that the choice of an optimal λ is worth writing a separate article. This can certainly not be covered in this article.

For large-scale problems such as $p = 10^6$ we are only able to compare the SQUIC variants. Here we like to draw the attention of the reader to the fact that reducing λ too much drastically increases the fill–in. This also affects the SQUIC algorithms, in particular, for $\lambda = 0.3$ the SQUIC(a**) implementations quickly lead to relatively dense matrices. Therefore we only display the algorithms that were able to treat $p = 10^6$ and $\lambda \geqslant 0.3$. Figure 6 states that the two remaining SQUIC(a**) algorithms lead to highest success ratio as long as λ is large enough and the memory consumption is small enough, but suddenly they fall back behind the SQUIC(b**) methods for $\lambda = 0.3$ which reveals the problem between fill–in of the iterates and obtaining a sufficient success rate. Although it sounds surprising that the two remaining SQUIC(a**) are eventually less accurate, though working with the exact empirical covariance matrix S s.t. $|s_{ij}| \geqslant 0.3$, this strange effect is explained by the larger amount of numerical errors working with denser matrices W that need to be approximated numerically. In contrast to that, the SQUIC(b**) work with a compressed version of S only, but this leads to a sparser representation and therefore to a higher success rate

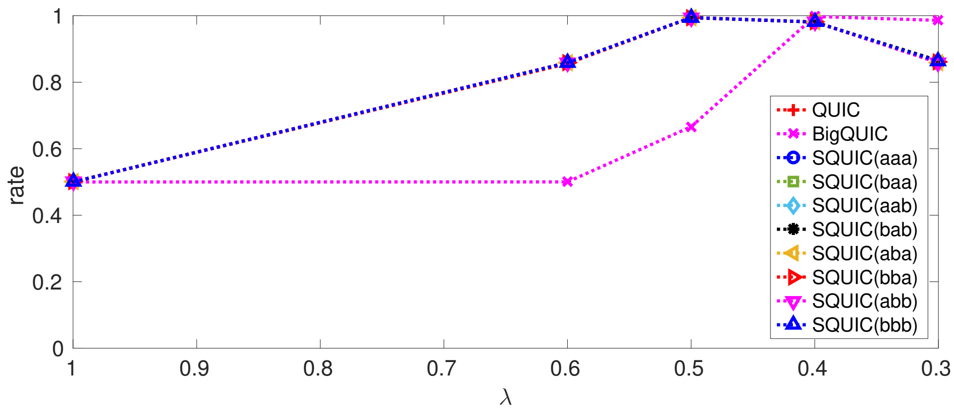


Figure 5: Success rate of QUIC-type algorithms for $p = 10^4$, Tridiagonal example 1.

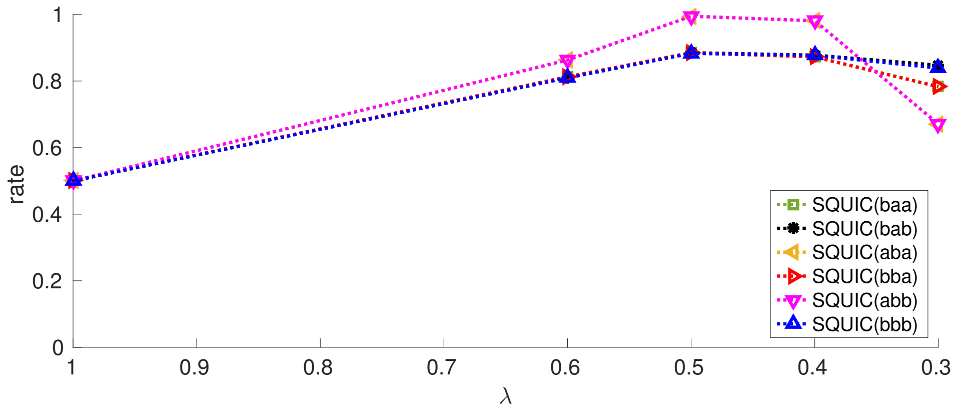


Figure 6: Success rate of some SQUIC algorithms for $p = 10^6$, Tridiagonal example 1.

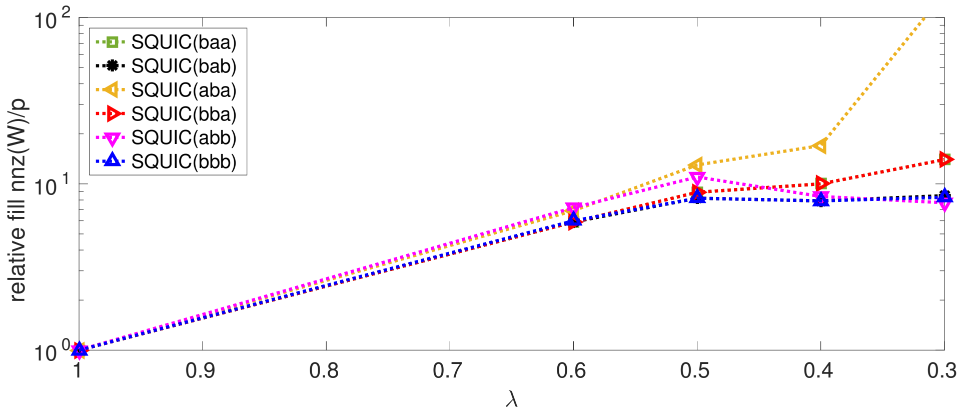


Figure 7: Relative fill SQUIC some algorithms, $p = 10^6$, Tridiagonal example 1.

for small λ.

Certainly, a natural question that arises is about the consumed memory. Figure 7 illustrates

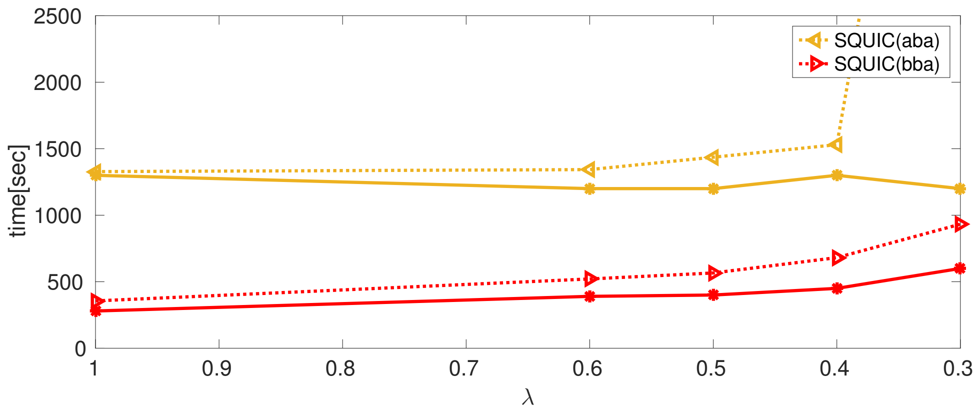


Figure 8: Computational amount for the initial S, $p = 10^6$, Tridiagonal example 1.

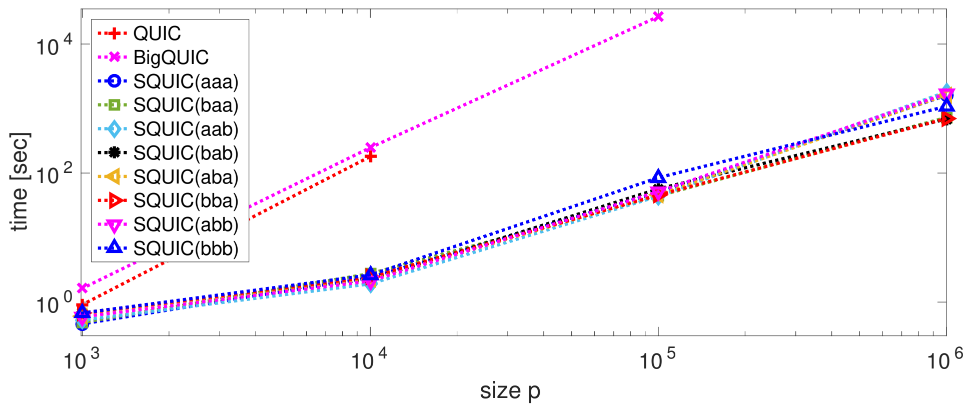


Figure 9: Computation time QUIC-type algorithms, Pentadiagonal example 2.

that, in particular, the SQUIC(b**) variants are still competitive while at the same time their success rate remains close to the best rate.

Finally for this first example, we like to mention that in several cases the computation of the sparse approximate representation of the empirical covariance matrix S consumes a significant amount of computation time. We show for two variants SQUIC(*ba) based on the exact deterministic algorithm SQUIC(aba) and on the randomized method SQUIC(bba) how much time is consumed for computing S. As shown in Figure 8, the convex optimization process often enough consumes only minor computation time compared with the generation of a sparse representation of S. Certainly, if the application-dependent, part of the relevant pattern of S is known, the algorithm could be accelerated even further.

5.2 Pentadiagonal example

For the pentadiagonal example 2 we get for $\lambda = 0.3$ a similar computation time profile as for the first example and $\lambda = 5$. We demonstrate this in Figure 9. Similarly to the first example, we also compare the success of each algorithm for $p = 10^4$ in Figure 10, where all methods are still able to compete and the results demonstrate a similar behavior from $\lambda = 1.0$ down to $\lambda = 0.3$ (again with BigQUIC being an exception). For $\lambda = 0.2$ the fill-in already significantly increases which explains the slightly greater differences there. The significant differences are also revealed when taking a

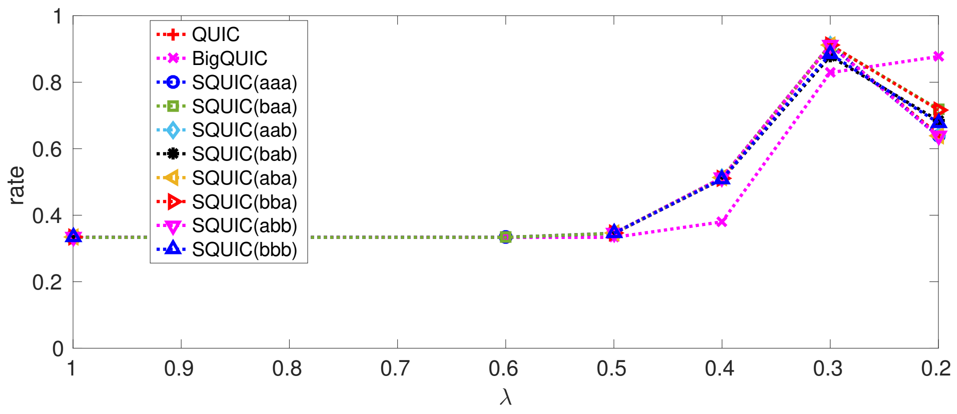


Figure 10: Success rate of QUIC-type algorithms for $p = 10^4$, Pentadiagonal example 2.

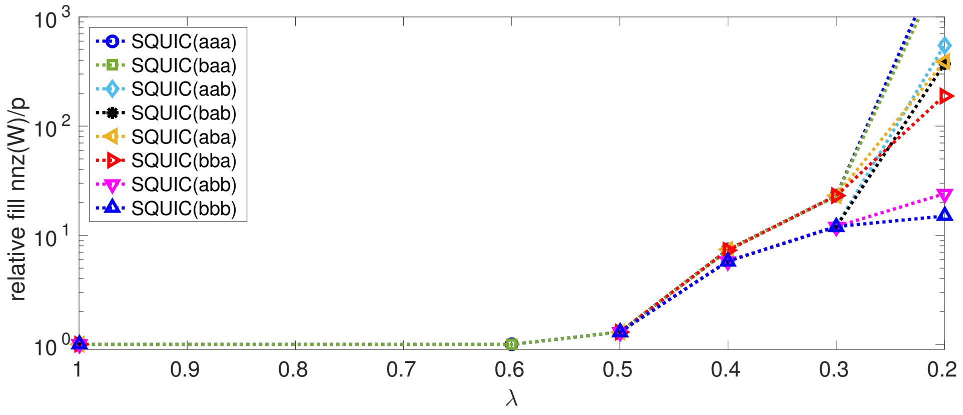


Figure 11: Relative fill SQUIC algorithms, $p = 10^4$, Pentadiagonal example 2.

closer look at the amount of memory as shown in Figure 11. The versions based on incomplete factorization and selective inversion seem to lead to a better memory usage. But certainly one has to keep in mind that selective inversion also limits the active set that can be used and possibly reduces the overall success. This has been observed for larger $p = 10^5$. Therefore the use of either approximate or selective inversion is twofold.

5.3 Random example

We now discuss the random example 3. Again QUIC, BigQUIC, and the different versions of SQUIC will be compared, yet we have to mention that in this example BigQUIC performed very poorly. We do not know the precise cause for this failure and maybe blame this to the random structure of this problem. Figure 12 displays the computational amount of work for all methods though not all algorithms were carried out for larger numbers of p if they were obviously out of competition. We also like to emphasize that the plot is logarithmic in both directions, in particular the two fastest algorithms SQUIC(bba) and SQUIC(bbb) are by more than one order of magnitude faster than the other ones for $p = 10^6$.

Before we start explaining these significant difference we also like to show the maximum relative fill $nnz(W)/p$ of W and $nnz(L)/p$ of the Cholesky factor L as computed during SQUIC. Again SQUIC(bba) and SQUIC(bbb) are by far better than all other methods as shown in Figure 13.

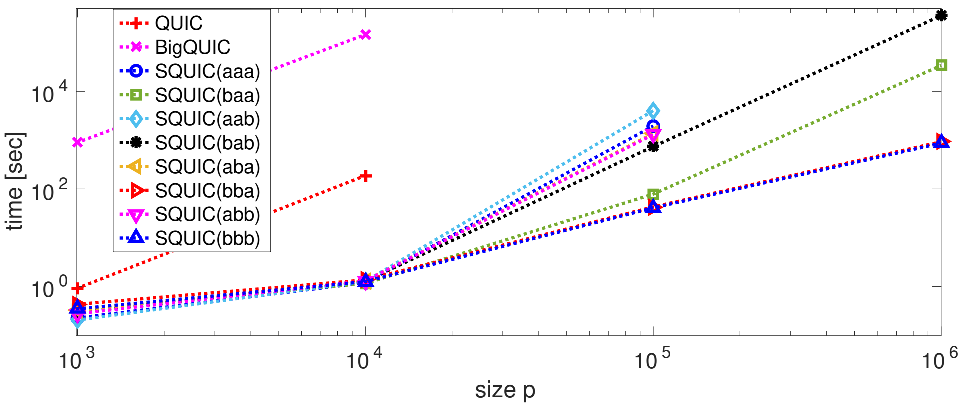


Figure 12: Computation time QUIC-type algorithms, Random example 3.

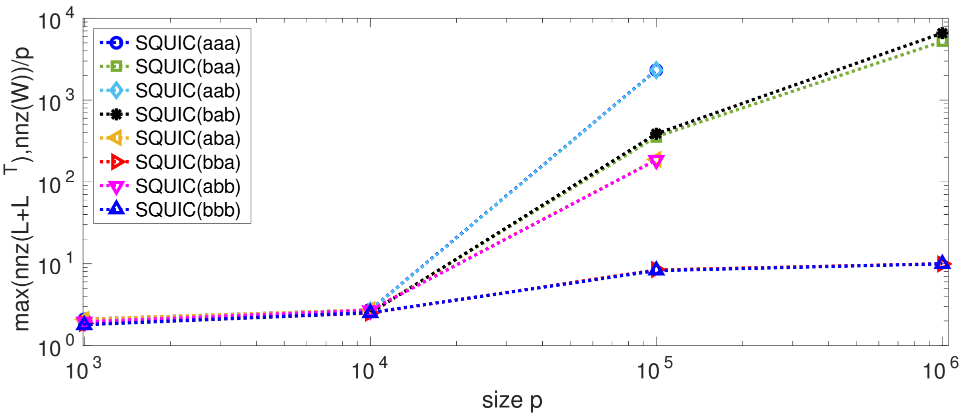


Figure 13: Maximum relative fill W and L during SQUIC, Random example 3.

After having seen the computation time and the relative fill we will now explain the effect. First of all, in contrast to the deterministic approach to compute all s_{ij} such that $|s_{ij}| \geqslant \lambda$, the randomized approach for computing the empirical covariance matrix certainly does not detect all entries of S such that $|s_{ij}| \geqslant \lambda$. This in particular affects the 1% entries which are not located within the diagonal blocks of size 20. This might be seen as a disadvantage, but since this problem has a random structure anyway, compressing the initial S is reasonable. This in turn has effects on the Cholesky decomposition and the incomplete LDL^T decomposition which perform by far more poorly when the 1% off-diagonal block entries have to be considered. Clearly, the Cholesky decomposition is more seriously slowed down than the incomplete factorization since there is no opportunity to drop some entries. When computing an approximate inverse for W, the amount of fill of the (incomplete) Cholesky factor exceeds by far the fill of the approximate inverse for larger $p = 10^5, 10^6$. This was in particular the case for all four SQUIC(*a*) variants that use the exact Cholesky decomposition. Among the other four methods based on incomplete factorizations, the versions using the deterministic computation of S were more affected (i.e., SQUIC(ab*)). This explains why only the SQUIC(bb*) methods were left over. In this case the selective/approximate inverse were sparse enough. One might argue that omitting parts of S may cause the algorithms to yield a poorer success rate. This was not observed, on the contrary, for large $p \geqslant 10^5$, the three SQUIC(b**) methods were even superior (approx. 0.45) than the three SQUIC(a**) versions (approximately 0.40). We explain this effect that having a lot of fill in the (incomplete) Cholesky

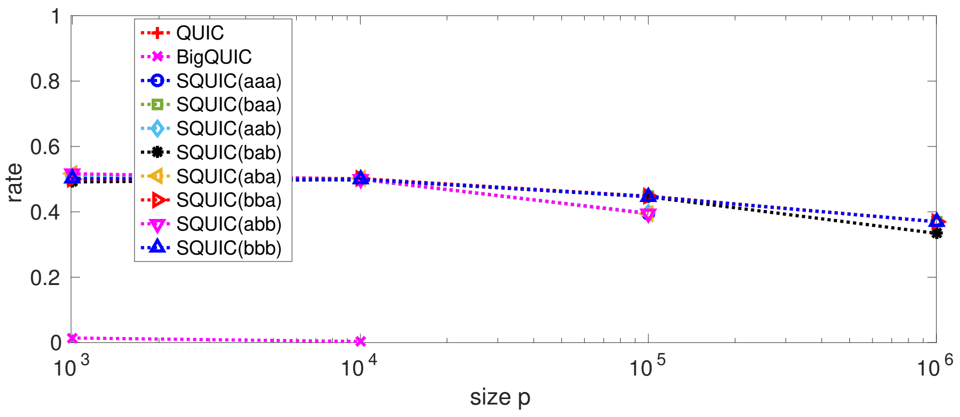


Figure 14: Success rate of QUIC-type algorithms, Random example 3.

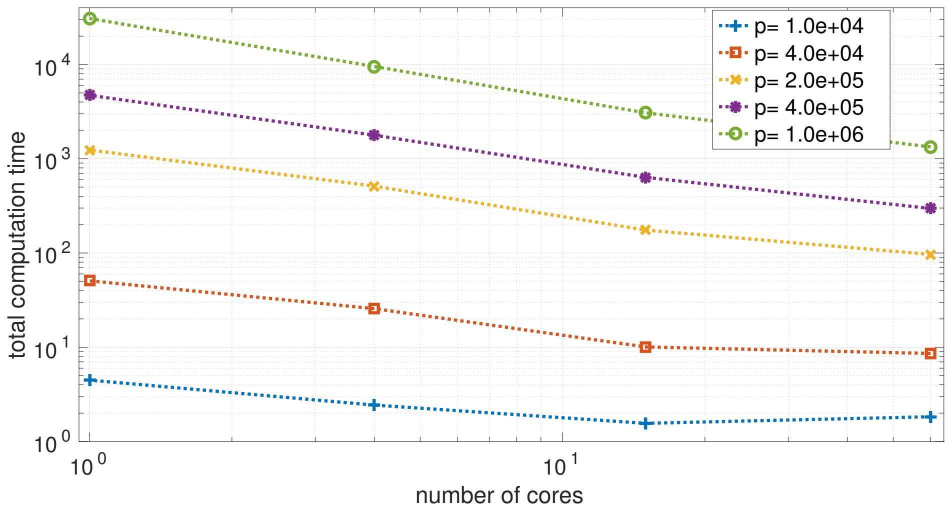
factor will also cause a larger amount of numerical rounding errors and approximation errors in W. This is displayed in Figure 14.

5.4 Parallel Performance

As part of our approach, computing in particular the sample covariance matrix $S = \frac{1}{n}\left[Y - \hat{\mu}\right]\left[Y - \hat{\mu}\right]^T$ is done in parallel. To do so, on one hand we use multi–threaded level BLAS3 for the associated matrix–matrix multiplication for each chunk of columns. On the other hand, sparsifying a chunk of columns can be done in parallel. To reveal the parallel performance of SQUIC we will consider Example 3 for the parameter $\lambda = 0.5$ and larger scales of p. In particular we demonstrate the parallel performance for $p = 10^4, 4 \cdot 10^4, 2 \cdot 10^5, 4 \cdot 10^5, 10^6$ using $1, 4, 15$ and 60 cores. The results are displayed in Figure 15. For the experiments in Figure 15 we tested the deterministic approach for generating S as well as its randomized counter part. The other two components of the algorithm are constantly chosen as the incomplete LDL^T decomposition of Θ and truncated Neumann series for generating $W \approx \Theta^{-1}$. We can see on one hand that the parallel performance scales almost linearly for the smaller number of cores while the parallel improvement is slightly worse when using larger number of cores. Besides, we observe again that the randomized version scales better with respect to the spatial dimension p. Please note that the generation of the initial S consumed almost all of the computation time. This in turn emphasizes why parallelizing this part is of particular importance.

6 Concluding remarks

In this paper, we were concerned with the computational cost in solving log-determinant optimization problems arising from the l_1-regularized Gaussian maximum likelihood estimator of a sparse inverse covariance matrix problem in high-dimensional settings. The novel aspects of the approach include our definition of the covariance matrices in the optimization method. Here, we used various advanced sparse linear algebra techniques to tackle three sub-problems as follows: we first generate the empirical covariance matrix S using a deterministic or randomized approach; secondly, we present novel techniques in QUIC to check for the positive definiteness of Θ and $\log(\det\Theta)$; and thirdly, we derive and evaluate two approximate inversion techniques based on a truncated Neumann series and a novel selected inversion method. These proposed algorithms can advance sparse inverse covariance estimation by orders of magnitude leading to scalability rates which are observed to be less than quadratic with respect to the p-variate dimension of the statistical problem. We have demonstrated that problems of size $p = 10^6$ can be easily computed within minutes on a single compute node. We showed that our method is highly comparable with respect



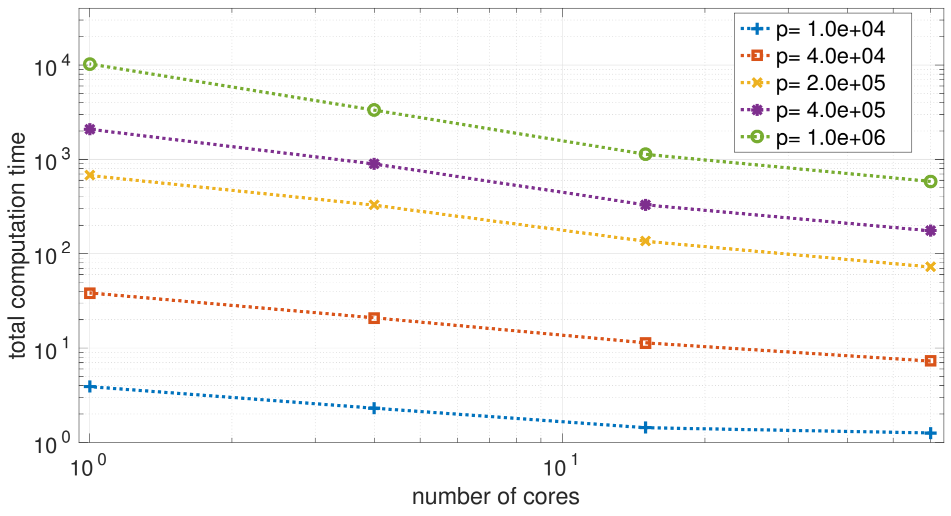


Figure 15: Parallel performance of SQUIC(aba) (top) and SQUIC(bba) (bottom), Tridiagonal example 1.

to solution quality with a state-of-the-art optimization algorithm, and it significantly outperforms the conventional approach in terms of storage and CPU time for the larger problem instances in our tests.

Interestingly, the computation of the empirical covariance matrix is often observed to be a major computational obstacle. Luckily, on large-scale parallel architectures having thousands of cores, this bottleneck can be easily bypassed or at least downscaled. In addition, a randomized algorithm with observed better numerical scalability remains as alternative. It is clear that this approach is extremely successful for and restricted to those application areas where the covariance matrix and its inverse are at least approximately sparse.

Acknowledgement

The authors of this paper like to gratefully thank the authors of [17] for their wonderful QUIC method which was the basis for the sparse matrix approach.

References

[1] P. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Analysis and Applications, 17 (1996), pp. 886–905.

[2] J. Ballani and D. Kressner, *Sparse inverse covariance estimation with hierarchical matrices*, tech. rep., EPFL Technical Report, 2014.

[3] O. Banerjee, L. E. Ghaoui, and A. d'Aspremont, *Model selection through sparse maximum likelihood estimation for multivariate gaussian or binary data*, The Journal of Machine Learning Research, 9 (2008), pp. 485–516.

[4] T. Cai, W. Liu, and X. Luo, *A constrained l_1 minimization approach to sparse precision matrix estimation*, Journal of the American Statistical Association, 106 (2011), pp. 594–607.

[5] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, *Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate*, ACM Transactions on Mathematical Software, 35 (2008), pp. 22:1–22:14.

[6] T. Coleman, B. Garbow, and J. Moré, *FORTRAN subroutines for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 346–347.

[7] T. Coleman and J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 20 (1983), pp. 187–209.

[8] A. Curtis, M. Powel, and J. Reid, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl., 13 (1974), pp. 117–119.

[9] J. Dahl, L. Vandenberghe, and V. Roychowdhury, *Covariance selection for non–chordal graphs via chordal embedding*, Optimization Methods and Software, 23 (2008), pp. 501–520.

[10] A. d'Aspremont, O. Banerjee, and L. E. Ghaoui, *First-order methods for sparse covariance selection*, SIAM J. Matrix Analysis and Applications, 30 (2008), pp. 56–66.

[11] J. Duchi, S. Gould, and D. Koller, *Projected subgradient methods for learning sparse Gaussians*, in Proceedings of the Twenty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-08), 2008, pp. 153–160.

[12] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM J. Matrix Analysis and Applications, 20 (1999), pp. 889–901.

[13] I. S. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. Matrix Analysis and Applications, 27 (2005), pp. 313–340.

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

[15] M. Hagemann and O. Schenk, *Weighted matchings for the preconditioning of symmetric indefinite linear systems*, SIAM J. Scientific Computing, (2006), pp. 403–420.

[16] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems*, Parallel Computing, 28 (2002), pp. 301–321.

[17] C.-J. Hsieh, M. A. Sustik, I. S. Dhillon, and P. K. Ravikumar, *Sparse inverse covariance matrix estimation using quadratic approximation*, in Advances in Neural Information Processing Systems, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, eds., vol. 24, Neural Information Processing Systems Foundation, 2011, pp. 2330–2338.

[18] C.-J. Hsieh, M. A. Sustik, I. S. Dhillon, P. K. Ravikumar, and R. A. Poldrack, *BIG & QUIC: Sparse inverse covariance estimation for a million variables*, in Advances in Neural Information Processing Systems, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, eds., vol. 26, Neural Information Processing Systems Foundation, 2013, pp. 3165–3173.

[19] D. Irony, G. Shklarski, and S. Toledo, *Parallel and fully recursive multifrontal supernodal sparse Cholesky*, Future Generation Computer Systems — Special issue: Selected numerical algorithms archive, 20 (2004), pp. 425–440.

[20] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Scientific Computing, 20 (1998), pp. 359–392.

[21] L. Li and K.-C. Toh, *An inexact interior point method for l_1-regularized sparse covariance selection*, Mathematical Programming Computation, 2 (2010), pp. 291–315.

[22] N. Li, Y. Saad, and E. Chow, *Crout versions of ILU for general sparse matrices*, SIAM J. Scientific Computing, 25 (2004), pp. 716–728.

[23] L. Lin, J. Lu, L. Ying, R. Car, and W. E, *Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems*, Commun. Math. Sci., 7 (2009), pp. 755–777.

[24] L. Lin, C. Yang, J. C. Meza, J. Lu, L. Ying, and W. E, *SelInv — an algorithm for selected inversion of a sparse symmetric matrix*, ACM Trans. Math. Software, (2010).

[25] F. Oztoprak, J. Nocedal, S. Rennie, and P. A. Olsen, *Newton-like methods for sparse inverse covariance estimation*, Advances in Neural Information Processing Systems, 25 (2012), pp. 755–763.

[26] J. K. P. R. Amestoy, I. S. Duff and J.-Y. L'Excellent, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal of Matrix Analysis and Applications, 23 (2001), pp. 15–41.

[27] B. Rolfs, B. Rajaratnam, D. Guillot, I. Wong, and A. Maleki, *Iterative thresholding algorithm for sparse inverse covariance estimation*, Advances in Neural Information Processing Systems, 25 (2012), pp. 1574–1582.

[28] J. Rothman, P. Bickel, E. Levina, and J. Zhu, *Sparse permutation invariant covariance estimation*, Electron. J. Stat., 2 (2008), pp. 494–515.

[29] K. Scheinberg and I. Rish, *Learning sparse Gaussian Markov networks using a greedy coordinate ascent approach*, in Machine Learning and Knowledge Discovery in Databases, J. Balczar, F. Bonchi, A. Gionis, and M. Sebag, eds., vol. 6323 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2010, pp. 196–212.

[30] ——, *Learning sparse Gaussian Markov networks using a greedy coordinate ascent approach*, in Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part III, 2010, pp. 196–212.

[31] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Journal of Future Generation Computer Systems, 20 (2004), pp. 475–487.

[32] O. Schenk, A. Wächter, and M. Weiser, *Inertia-revealing preconditioning for large-scale nonconvex constrained optimization*, SIAM J. Sci. Comput., 31 (2008), pp. 939–960.

[33] K. Takahashi, J. Fagan, and M.-S. Chin, *Formation of a sparse bus impedance matrix and its application to short circuit study*, IEEE Power Engineering Society, 1973, pp. 63–69.

[34] J. M. Tang and Y. Saad, *A probing method for computing the diagonal of a matrix inverse*, Numerical Linear Algebra with Applications, 19 (2012), pp. 485–501.

[35] M. Yuan and Y. Lin, *Model selection and estimation in the Gaussian graphical model*, Biometrika, 94 (2007), pp. 19–35.