A Parallel, Block Greedy Method for Sparse Inverse Covariance Estimation for Ultra-high Dimensions

Prabhanjan Kambadur
IBM T.J. Watson Research Center
pkambadu@us.ibm.com

Aurélie Lozano
IBM T.J. Watson Research Center
aclozano@us.ibm.com

Abstract

Discovering the graph structure of a Gaussian Markov Random Field is an important problem in application areas such as computational biology and atmospheric sciences. This task, which translates to estimating the sparsity pattern of the inverse covariance matrix, has been extensively studied in the literature. However, the existing approaches are unable to handle ultra-high dimensional datasets and there is a crucial need to develop methods that are both highly scalable and memory-efficient. In this paper, we present GINCO, a blocked greedy method for sparse inverse covariance matrix estimation. We also present detailed description of a highly-scalable and memory-efficient implementation of GINCO, which is able to operate on both shared- and distributed-memory architectures. Our implementation is able recover the sparsity pattern of $25,000$ vertex random and chain graphs with 87% and 84% accuracy in $\leq$ 5 minutes using $\leq 10GB$ of memory on a single 8-core machine. Furthermore, our method is statistically consistent in recovering the sparsity pattern of the inverse covariance matrix, which we demonstrate through extensive empirical studies.

1 Introduction

Estimating the network structure of a Gaussian Markov random field (GMRF) is a key building block in many modern applications such as biological network discovery and climate modeling. Given its applications, this task has become a classical problem in high dimensional statistics, machine learning, and optimization. Estimating the network structure of GMRF is equivalent to estimating the inverse covariance matrix of a multivariate Gaussian distribution given some independently drawn samples; in short, the non-zero entries of the inverse covariance matrix map to the underlying graph structure of the GMRF.

Appearing in Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS) 2013, Scottsdale, AZ, USA. Volume 31 of JMLR: W&CP 31. Copyright 2013 by the authors.

A popular approach to estimating sparse inverse covariance matrices is the ℓ_1-penalized maximum likelihood formulation, which results in log-determinant optimization problems. Solutions to the ℓ_1-penalized maximum likelihood formulation include block coordinate descent [3, 13], greedy coordinate descent [15], projected subgradient [2], alternating linearization [14], second order methods including an inexact interior point method [8], and a quadratic approximation-based method (QUIC) [4]. Of these solutions, QUIC has been reported to be significantly faster than the rest. QUIC achieves economy of computation by using an inner coordinate relaxation iteration to compute the Newton step and exploiting the properties of this relaxation on a function that is the sum of a convex quadratic and an ℓ_1 term.

Recently, greedy selection methods have received considerable attention as an alternative to ℓ_1-penalized methods. Greedy methods are iterative procedures that perform forward greedy feature selection steps and optional backward greedy removal steps; each step is followed by a re-estimation of the model parameters. The most popular greedy method is Orthogonal Matching Pursuit (OMP) [10], which originated from the signal-processing community. Strong theoretical performance guarantees and empirical support for greedy methods' variable selection and estimation accuracy have been provided under the linear model [17, 9]. Recently, these guarantees have been extended to the multiple kernel learning setting [16]. The most striking result for greedy methods pertains to sparse inverse covariance estimation: greedy methods need significantly fewer samples than their ℓ_1-penalized counterparts — $O(dlog(p))$ vs $O(d^2log(p))$ — to recover the true network structure with high probability, where d is the maximum degree in the true inverse covariance matrix and p its dimensionality [6].

In addition, restricted eigenvalue and smoothness requirements are much weaker for greedy methods when compared to their ℓ_1-penalized counterparts.

The focus of the current paper is to tackle the inverse covariance estimation problem for ultra-high dimensions, which is a setting that is increasingly prevalent in applications domains such as next generation sequencing and ultra-high resolution climatology. In such settings, in addition to *fast convergence*, algorithms need to be *highly scalable* and *memory efficient*. The ℓ_1 penalty does not provide a "firm" control over the sparsity of the intermediate solutions. Consequently, ℓ_1-penalized approaches, though efficient in terms of optimization convergence, may reach dense intermediate solutions, which are unacceptable memory-wise when dealing with ≥ 10000 dimensions. In addition, ultra-high dimensional problems have a low ratio of sample size (m) to dimensionality ($m \lll p$); greedy methods require fewer samples than ℓ_1-penalized approaches. Therefore, we believe that greedy methods are better-suited to tackle the problem of sparse inverse covariance estimation for ultra-high dimensional datasets. Till now, the sequential nature of the forward and backward steps in the greedy methods remained a bottleneck to their scalability. In this paper we propose to mitigate this bottleneck by introducing blocking and efficient computational techniques borrowed from high performance computing. Our approach, titled Greedy Inverse Covariance estimation, or GINCO, is able to recover the sparsity pattern of $25,000$ vertex random and chain graphs with 87% and 84% accuracy in ≤ 5 minutes using $\leq 10GB$ of main memory on a single 8-core machine. The key contributions of this paper are:

- *Blocking:* We extend traditional greedy methods to select multiple candidates in both the forward and backward phases. Blocking results in significant speedups with little or no loss in accuracy.
- *Theoretical guarantees:* We show that GINCO is statistically consistent in recovering the sparsity pattern of the inverse covariance matrix. In particular, we prove that GINCO preserves the significant improvement on the sample size requirements that is enjoyed by the traditional (*unit block*) greedy methods over ℓ_1-penalized counterparts.
- *Parallelization:* We exposit the procedure for efficient parallelization of GINCO, which results in a highly-scalable and memory-efficient implementation that can exploit both shared- and distributed-memory architectures simultaneously.
- *Empirical studies:* To demonstrate the accuracy and efficiency of GINCO, we present extensive and detailed small-scale and large-scale experiments.

2 Method

Algorithm 1: GreedyInverseCovariance

```
Input: $S$: sample covariance, $\tau$: max number of
       predictors, $b_f$: forward step block size, $b_b$:
       backward step block size, $\epsilon_f$: min forward gain, $\epsilon_b$:
       backward stop factor. $\epsilon_r$: refitting tolerance.
Output: $W$: inverse covariance, $\Sigma$: covariance, $L$:
        maximum likelihood estimate.
1  $W = I_{p,p}$; $\Sigma = I_{p,p}$; $L = \operatorname{tr} S$;
2  while true do
3      $bs$ =Min($b_f$, ($\tau-$NNZ($W$)));
4      $M$ =ForwardEvaluator($S, \Sigma, W, L, bs$);
5      if $M == \emptyset$ then break;
6      $(W_f, \Sigma_f, L_f)$ =Update($M, S, W, \Sigma, L$);
7      $(W_f, \Sigma_f, L_f)$ =Refit($W_f, \Sigma_f, L_f, S, \epsilon_r$);
8      $\delta_f = \frac{L-L_f}{|M|}$; if $\delta_f < \epsilon_f$ then break;
9      $(W, \Sigma, L) = (W_f, \Sigma_f, L_f)$;
10     while true do
11         $bs$ =Min($b_b$,NNZ($W$));
12         $M$ =BackwardEvaluator($S, \Sigma, W, L, bs$);
13         if $M == \emptyset$ then break;
14         $(W_b, \Sigma_b, L_b)$ =Update($M, S, W, \Sigma, L$);
15         $(W_b, \Sigma_b, L_b)$ =Refit($W_b, \Sigma_b, L_b, S, \epsilon_r$);
16         $\delta_b = \frac{L_b-L}{|M|}$; if $\delta_b > \delta_f\epsilon_b$ then break;
17         $(W, \Sigma, L) = (W_b, \Sigma_b, L_b)$;
18     return $(W, \Sigma, L)$;
```

Let $X_{(p,m)}$ be the matrix of observations that are assumed to be generated from a zero-mean Gaussian vector. Let S be the sample covariance matrix, which is defined as $S = \frac{1}{m}\Sigma_{k=1}^{m} X^{(k)}(X^{(k)})^T$. Let W^*, Σ^* be the true covariance and inverse covariance matrices, respectively ($W^* = \Sigma^{*-1}$). Then, the inverse covariance matrix can be estimated by greedy minimization of the Gaussian log-likelihood loss

$$L = \min_{W} [\operatorname{tr} WS - \log\det W] \tag{1}$$

A forward-backward algorithm to compute W greedily was given in [6]. In our work, we extend this method to handle block additions and deletions to the iterate W and exposit the procedure that we followed in detail. Our method, GreedyInverseCovariance() or GINCO, is outlined in Algorithm 1. Traditionally b_f and b_b, the block sizes for forward and backward phases, are 1, which indicates picking just one candidate to make non-zero at each iteration; note that $0 \leq \tau \leq \frac{p^2-p}{2}$ and $(b_f, b_b) \leq \tau$, where τ is the maximum number of candidates to select. In the first phase, we initialize all the required variables; W and Σ are initialized to be the identity ($I_{p,p}$), and L, the objective is initialized to be the trace of S. GINCO alternates between forward (Algorithm 1, lines $3-9$) and backward phases (Algorithm 1, lines $11-17$), which are similar in flavor.

In each phase, we first compute the block size (b_f/b_b), which determines the number of candidates to select for addition/deletion. The next step is to select the required number of candidates by sweeping through the list of potential candidates (ForwardEvaluator()/BackwardEvaluator()) [1]. In the forward phase, the list of potential candidates consists of all zero entries in W, whereas in the backward phase, this list consists of all non-zero entries in W; the procedure for selecting the candidates is outlined in Section 2.1. Once the set of candidates (M, set of (i, j, α_{ij})) are selected, matrices W, Σ, and the likelihood estimate L are updated (Update()). The update to W consists of setting the chosen positions to the computed values for those positions. However, efficient updates to Σ and L require care and are discussed in Section 2.2 and Section 2.3 respectively. After updating the required values, we re-estimate the value of the model parameters (W, Σ, L) to propagate the selection/removal of the current set of candidates (Refit()). Refitting is done using coordinate descent by iteratively re-estimating one position of W in each iteration; the procedure stops when the change in L is smaller than ϵ_r, the tolerance for refitting. Note that we only need to refit those entries in W that share a common row or column with any of the chosen candidates. Next, the gain/loss from adding/removing the chosen set of candidates is evaluated for significance. In the forward phase, the gain from adding the candidates is expected to be at least ϵ_f; in the backward phase, the maximum loss that we can tolerate due to removal of the chosen set of candidates is the forward gain times ϵ_b, the backward stop factor. If we have sufficient gain or minimal loss, the new set of candidates is accepted and the process is repeated. The forward phase terminates either if the maximum number of candidates (τ) are selected or if the gain from adding the current set of candidates is insufficient. The backward phase terminates either if there are no more candidates or loss from removing the current set of candidates is too high.

2.1 Candidate Selection

A key part of the greedy procedure to estimate W is to select the best candidates to add given the previous candidates; that is, we need to find α_{ij} such that $\mathrm{L}(W + \Delta_{ij})$ is minimized, where $\Delta_{ij} = \alpha_{ij}(e_{ij} + e_{ji})$. From Equation 1, we have:

$$\mathrm{L}(W+\Delta_{ij}) = \mathrm{tr}\,(W + \Delta_{ij})S - \log\det\,(W + \Delta_{ij}) \quad (2)$$

To get an equation in α_{ij}, first, consider the term $\mathrm{tr}\,(W + \Delta_{ij})S$; this can be expanded as $\mathrm{tr}\,WS + \mathrm{tr}\,\Delta_{ij}S$. Furthermore, given the structure of Δ_{ij}, $\mathrm{tr}\,\Delta_{ij}S$ is simply $2\alpha_{ij}s_{ij}$. Similarly, from the property of determinants, we know that $\det\,(W + \Delta_{ij}) =$

[1]See supplementary material for ForwardEvaluator(), BackwardEvaluator(), Update(), and Refit().

Algorithm 2: ComputeAlpha

```
Input: $S$: sample covariance matrix, $\Sigma$: current estimate
       of covariance, $(i, j)$: position
Output: $\alpha$: the value at position $(i, j)$
1  $\rho = \sigma_{ii}\sigma_{jj}$ ; $\gamma = \sigma_{ij}^2 - \sigma_{ii}\sigma_{jj}$; $\delta = \frac{1}{2s_{ij}}$;
2  if $|s_{ij}\gamma| \lll 0$ then
3      $\alpha = -\frac{s_{ij}-\sigma_{ij}}{2s_{ij}\sigma_{ij}-\gamma}$;
4      if $(1+\alpha\sigma_{ij})^2 - \alpha^2\sigma_{ii}\sigma_{jj} > 0$ then return $\alpha$;
5  else
6      $\phi = \delta^2 + \frac{\rho}{\gamma^2}$;
7      if $\phi \geq 0$ then
8          $(\alpha_{min}, \alpha_{max}) = \delta - \frac{\sigma_{ij}}{\gamma} \pm \sqrt{\phi}$;
9          $(\beta_{min}, \beta_{max}) = \frac{-1}{\sigma_{ij} \pm \sqrt{\rho}}$;
10         if $s_{ij} \geq 0$ and $\gamma > 0$ then $\alpha = \alpha_{max}$;
11         else if $s_{ij} \geq 0$ and $\gamma < 0$ then $\alpha = \alpha_{min}$;
12         else if $s_{ij} < 0$ and $\gamma > 0$ then $\alpha = \alpha_{min}$;
13         else if $s_{ij} < 0$ and $\gamma < 0$ then $\alpha = \alpha_{max}$;
14         if $\beta_{min} \leq \alpha \leq \beta_{max}$ then return $\alpha$;
15 return $\infty$
```

$(\det W)[(1 + \alpha_{ij}\sigma_{ij})^2 - \alpha_{ij}^2\sigma_{ii}\sigma_{jj}]$. Substituting these in Equation 2, we get:

$$\mathrm{L}(W + \Delta_{ij}) = \mathrm{tr}\,WS + 2\alpha s_{ij} - \log\det W - \log\left[(1+\alpha_{ij}\sigma_{ij})^2 - \alpha_{ij}^2\sigma_{ii}\sigma_{jj}\right] \quad (3)$$

Rearranging the terms and substituting $\mathrm{L}(W) = \mathrm{tr}\,WS - \log\det W$, we get:

$$\mathrm{L}(W + \Delta_{ij}) = \mathrm{L}(W) + 2\alpha_{ij}s_{ij} - \log\left[(1+\alpha_{ij}\sigma_{ij})^2 - \alpha_{ij}^2\sigma_{ii}\sigma_{jj}\right] \quad (4)$$

We see that the effect of adding Δ_{ij} to W on L(W) can be expressed as a function of α_{ij}:

$$\mathrm{F}(\alpha_{ij}) = 2\alpha_{ij}s_{ij} - \log\left[(1+\alpha_{ij}\sigma_{ij})^2 - \alpha_{ij}^2\sigma_{ii}\sigma_{jj}\right] \quad (5)$$

The value(s) of α_{ij}^*, which minimize(s) $\mathrm{F}(\alpha_{ij})$ also minimize(s) $\mathrm{L}(W + \Delta_{ij})$. To find this $\alpha_{ij}*$, we set the first derivate of F with respect to α_{ij} to 0 giving us:

$$\frac{\delta\mathrm{F}}{\delta\alpha_{ij}} = 2s_{ij} - \frac{2\sigma_{ij}(1+\alpha_{ij}\sigma_{ij}) - 2\alpha_{ij}\sigma_{ii}\sigma_{jj}}{(1+\alpha_{ij}\sigma_{ij})^2 - \alpha_{ij}^2\sigma_{ii}\sigma_{jj}} = 0 \quad (6)$$

Equation 6 is quadratic in α_{ij}, which can be rewritten and solved to get the roots:

$$[s_{ij}(\sigma_{ij}^2 - \sigma_{ii}\sigma_{jj})]\alpha_{ij}^2 + [2s_{ij}\sigma_{ij} - \sigma_{ij}^2 + \sigma_{ii}\sigma_{jj}]\alpha_{ij} + [s_{ij} - \sigma_{ij}] = 0 \quad (7)$$

$$\alpha_{ij}^* = \frac{1}{2s_{ij}} - \frac{\sigma_{ij}}{\sigma_{ij}^2 - \sigma_{ii}\sigma_{jj}} \pm \sqrt{\frac{1}{4s_{ij}^2} + \frac{\sigma_{ii}\sigma_{jj}}{(\sigma_{ij}^2 - \sigma_{ii}\sigma_{jj})^2}} \quad (8)$$

Equation 8 gives the unconstrained solution for α_{ij}^*. To decide which of the two roots to choose requires setting the second derivative of $\mathrm{F}(\alpha)$ to 0. There are two additional conditions that need to be checked to ensure numerical stability. First, for many (i, j), the value α^*_{ij} is infeasible. To ensure this, it is important to analyze the denominator of $\frac{\delta \mathrm{F}}{\delta \alpha}$ — $(1+\alpha\sigma_{ij})^2 - \alpha^2\sigma_{ii}\sigma_{jj}$ — as well. This denominator can be rewritten as $(1+\alpha_{ij}\sigma_{ij})^2 - (\alpha_{ij}\sqrt{\sigma_{ii}\sigma_{jj}})^2$, which has the roots: $\beta^*_{ij} = \frac{-1}{\sigma_{ij} \pm \sqrt{\sigma_{ii}\sigma_{jj}}}$. In short, for an α^*_{ij} to be a valid candidate for addition, it should lie between the two values of β^*_{ij}. Second, for an α^*_{ij} to be considered a candidate, we must test that $W + \Delta_{ij}$ is positive-definite. Algorithm 2 shows the procedure of computing α taking all the above factors into account. Note that Algorithm 2 first tries to compute α_{ij} as a linear approximation if $s_{ij}(\sigma^2_{ij} - \sigma_{ii}\sigma_{jj})$ is very small, which stems from Equation 7. Algorithm 1 selects *bs* candidates at a time. This is achieved by keeping the top-*bs* of all the candidates available for section in both the forward and the backward phases.

2.2 Updating Σ

When $\Delta_{ij} = (\alpha_{ij}(e_{ij} + e_{ji}))$ is added to W, $\Sigma(W^{-1})$ also needs to be updated. Because of the special structure of Δ_{ij}, we use the following (Sherman-Morrison-Woodbury) formula to update Σ:

$$\Sigma_{new} = \Sigma - \frac{\alpha(1+\alpha_{ij}\sigma_{ij})(\Sigma_i\Sigma_j^T + \Sigma_j\Sigma_i^T)}{(1+\alpha_{ij}\sigma_{ij})^2 - \alpha_{ij}^2\sigma_{ii}\sigma_{jj}} + \frac{\alpha^2(\sigma_{jj}\Sigma_i\Sigma_i^T + \sigma_{ii}\Sigma_j\Sigma_j^T)}{(1+\alpha_{ij}\sigma_{ij})^2 - \alpha_{ij}^2\sigma_{ii}\sigma_{jj}} \tag{9}$$

The above equation shows us that when Δ_{ij} is added to W, $\Sigma((W+\Delta_{ij})^{-1})$ receives a symmetric *rank-2* update, which consists of the following (scaled) symmetric *rank-1* matrices $\Sigma_i(\Sigma_j^T + \Sigma_i^T)$ and $\Sigma_j(\Sigma_i^T + \Sigma_j^T)$.

2.3 Updating Likelihood

When $\Delta_{ij} = (\alpha_{ij}(e_{ij} + e_{ji}))$ is added to W, the new likelihood $(\mathrm{L}(W + \Delta_{ij}))$ is computed using Equation 4.

2.4 Theoretical Analysis

We extend the analysis in [5] and show that GINCO recovers the true structure of the inverse covariance matrix. In particular, GINCO enjoys similar advantage as its unit-block version proposed in [6] over ℓ_1-penalized approaches. The impact of the block size appears on the condition pertaining to the restricted eigenvalue property. For ease of analysis, we consider a variant of GINCO, where the foward step breaks if $\delta_f = \sup_{(i,j)\in M} L - (L + \alpha_{i,j}e_{i,j}) < \epsilon_f$, where L is the log likelihood loss on entry to $\mathsf{ForwardEvaluator()}$ and M are the candidates selected by it. Similarly, assume that the backward step breaks if $\delta_b = \inf_{(i,j)\in M}(L - \alpha_{i,j}e_{i,j}) - L > \epsilon_b\delta_f$, where L is the log likelihood loss on entry to $\mathsf{BackwardEvaluator()}$ and M are the candidates selected for removal. We consider $\epsilon_b = 1/2$.

$$\begin{bmatrix} & 1 & 2 & 3 & 4 \\ & & 5 & 6 & 7 \\ & & & 8 & 9 \\ & & & & 10 \\ & & & & \end{bmatrix} \begin{bmatrix} & x & x & x & x \\ & & x & o & o \\ & & & o & o \\ & & & & o \\ & & & & \end{bmatrix}$$

R	Own	Rep
1	P1	-
2	P1	P2
3	P1	-
4	P1	-
5	P2	-

Figure 1: The candidate matrix on the left with numbered candidates appearing in the upper triangle. The center matrix shows how these candidates are sub-divided if there were 2 processors P1 and P2 — candidates $[1-5]$ are evaluated by P1(x) and $[6, 10]$ are evaluated by P2(o). The corresponding row-ownership table is depicted on the right. Notice that row 2 is needed by both P1 and P2; P1, being higher numbered, is given ownership of row 2, but row 2 is replicated (Rep) on P2 to minimize communication.

Assumptions Following [5], let $\rho \geq 1$ denote a constant. Let Δ be a $p \times p$ sparse, symmetric matrix with at most ηd non-zero entries per row/column, where $\eta \geq 2 + 4\rho^2(\sqrt{\frac{(\rho^2-\rho)b}{d}} + \sqrt{2})^2$. The population covariance matrix Σ^* is assumed to satisfy the restricted eigenvalue property: $C_{min}\|\Delta\|_F \leq \langle\langle\Sigma^*, \Delta\rangle\rangle \leq \rho C_{min}\|\Delta\|_F$, where, $\|\cdot\|_F$ denotes the Frobenius norm. The following theorem states the sparsistency of GINCO.

Theorem 1 *Suppose* GINCO *is run with block sizes* $b_f = b_b = b$, *stopping threshold* $\epsilon_f > (2c^2\eta/\rho^2)d\log(p)/m$, *where, d is the maximum node degree in the graphical model, and the true parameters W^* with support $\mathcal{S}^*$ satisfy* $\min_{(i,j)\in\mathcal{S}^*}|W^*_{i,j}| \geq \sqrt{8\epsilon_f/\rho^2}$, *and that the sample size* $m > Kd\log(p)$ *for some constant K. Then, with probability at least* $1 - c_1\exp(-c_2 m)$, *we have no false exclusion and no false inclusions of entries, with $c_1, c_2 > 0$ constants.*

3 Parallelization

To solve ultra-high dimensional problems, it is necessary to efficiently parallelize Algorithm 1. Our goals were: (1) achieve near-linear speedup in running time as the number of processors/threads increase, (2) maintain a near-flat memory profile as the number of processors/threads increase, and (3) minimize memory requirements by partitioning and exploiting sparsity when possible. In this section, we give a brief description of key ideas that were used to achieve these goals.

3.1 Candidate Selection

Computationally, the most expensive step in Algorithm 1 is the evaluation of candidates in the forward phase (see Section 2.1). In this step, each unselected element from the upper triangle of W is evaluated as a potential candidate for addition. This 2-dimensional forward evaluation space can be flattened and parti-

$$\begin{bmatrix} - & - & - & P1 & - \\ & - & - & P1 & - \\ & & - & P2 & - \\ & & & P2 & P2 \\ & & & & - \end{bmatrix}$$

Figure 2: An example of accumulation of row 4 from Figure 1. Processor P2 owns row 4, but it does not have all the elements as we store only the upper triangle. Instead, row 4 is formed by both P1 and P2 using Algorithm 3.

tioned amongst all the processing elements. For example, consider the $W_{5,5}$ matrix shown on the left in Figure 1: the evaluation space in this case is $[1, 10]$. If this space were to be divided between processors $P1$ and $P2$, each processor would get 5 elements each, which is shown in the center of Figure 1. $P1$ and $P2$ can now independently choose their top candidates for inclusion into W. Following this, a global reduction of the top candidates across all processors gives us the global list of top candidates. Parallelization of the backward phase is similar; at each step, we need to choose the top candidates for removal from our model from the list of selected candidates, which constitutes the backward evaluation space. However, as Σ and S are partitioned (see Section 3.2), candidates for evaluation are assigned to processors based on the ownership of relevant rows. In Figure 1, candidate 9 can only be evaluated by P2, whereas candidates 5,6, and 7 can be evaluated by either P1 or P2.

3.2 Partitioning S and Σ

To evaluate a candidate in position (i, j), we need access to Σ_{ii}, Σ_{jj}, Σ_{ij} and S_{ij} (see Algorithm 2). That is, in Figure 1, apart from the diagonal of Σ, processors $P1$ and $P2$ only require access to rows $[1, 2]$ and $[2, 5]$ of Σ and S, respectively. Therefore, both Σ and S can be partitioned by rows (with some overlap) and stored across processors to conserve space; the ownership table for the $W_{5,5}$ example is shown in the far-right of Figure 1; row 2, which is needed by both $P1$ and $P2$, is replicated. As Σ is both read and written, replicated partitioning requires us to carefully update Σ to avoid inconsistencies; therefore, we assign ownership of each row to a *distinct* processor. In Figure 1, although row 2 is replicated, ownership is assigned to higher numbered $P1$; there are no strict rules to determine ownership of replicated rows. Note that the diagonal of Σ is *fully* replicated on all processors.

3.3 Updating Σ

Update to Σ requires the rows Σ_i and Σ_j (see Equation 9); as we partition and store only the upper triangle of Σ, this step requires global participation. For

```
Algorithm 3: AccumulateRow
Input: $\Sigma_{lcl}$: local portion of $\Sigma$, $OWN$: row ownership
       ($id \rightarrow (first, last)$), $r$: row to be accumulated,
       $id$: ID of the current process.
Output: $R$: The accumulated row of $\Sigma$
1 $R_{part} = \emptyset$;
2 for $i \in OWN(id)$ do
3     if $i \geq r$ then break;
4     Append($R_{part}, \Sigma_{lcl}(i, r)$);
5 if $r \in OWN(id)$ then Append($R_{part}, \Sigma_{lcl}(r, :)$);
6 AllGatherV($R_{part}, |R_{part}|, R$);
7 return $R$
```

example, the elements needed to form Σ_4 are shown in Figure 2. In this example, row 4 belongs to $P2$; however, $P2$ needs values from $P1$ to complete row 4. In GINCO, Algorithm 3 is used to form a row; each processor owns a portion of Σ, which we refer to as Σ_{lcl}. To form row r, we traverse down column r and then traverse across row r upon hitting the entry $\Sigma_{r,r}$. The first loop (line 2 in Algorithm 3) traverses along the column; the next check ensures that we turn upon reaching the diagonal and is encountered only on the processor that owns row r. At this point, the process that owns row r fills in rest of the entries of r. Finally, all the processors send their portion of Σ_r, which is gathered and redistributed using AllGatherV() [11, 12].

Exploiting Sparsity Σ starts out being hyper-sparse ($I_{p,p}$) and it gradually becomes more dense. The number of non-zeros added by an update to Σ depends on the non-zero structure in Σ_i and Σ_j. For sparse, structured problems such as chain graphs, we can exploit sparsity to speed up these updates to Σ.

Block Update As discussed in Section 2.2, updates to Σ are *rank-2* updates, which have low compute-to-memory ratio; consequently, the update operation is memory bandwidth bound. In GINCO, we parallelize this *rank-2* update by partitioning Σ by rows, both at thread- and processor-levels. When the block size, bs, is ≥ 2, one of three strategies can be followed to make efficient updates to Σ; the choice of strategy is dictated by the block size. First, we can apply the bs *rank-2* updates to Σ one at a time, where each individual update is parallelized as discussed above. This is the strategy currently implemented in GINCO, and it is efficient for small block sizes. Second, for medium-sized blocks, we can update Σ in a tree-like fashion. That is, we can separately prepare the bs *rank-2* update matrices in parallel and merge them using an *n-ary* tree till we are left with a single *rank-2* update matrix, which is used to update Σ. This strategy reduces the time complexity of updating Σ by a logarithmic factor at the cost of increasing the memory consumption. Third, when the block sizes are large — $O(NNZ)$ — it is more efficient

Case	GINCO				QUIC
	best		worst		
	block	$F_1\%$	block	$F_1\%$	
100-R	40	89.01	99	82.91	76.56
100-C	2	80.95	31	78.79	77.52
1K-R	700	92.83	300	86.48	78.83
1K-C	10	86.13	999	79.72	77.68

Table 1: The mean accuracy ($F_1\%$) achieved by GINCO (best and worst) and QUIC (100 simulated datasets per case); for GINCO, we also list the block size at which the reported mean accuracy was obtained. GINCO was run with varying block sizes and for each block size, 100 simulations were run. For standard errors, see Figure 3.

to explicitly invert W using methods such as Cholesky factorization to get the updated Σ instead of using Equation 9. This approach has the added benefit that it allows us to use high-quality, parallel packages such as ScaLAPACK [1] to compute Σ.

4 Results

To empirically determine the behavior of GINCO, we conducted several small- and large-scale experiments. Specifically, we wanted to answer three questions. (1) Is GINCO as accurate as its ℓ_1-penalized counterparts? (2) How does block size affect accuracy and running time? (3) Is GINCO scalable and if so, what are the scaling characteristics? In this section, we provide detailed results and analyses of our experiments.

Throughout this section, we will use the F_1 score, which is the harmonic mean of precision and recall, as the measure of accuracy. Following [4], we use chain and random graphs in our experiments. Briefly, chain graphs are tri-diagonal having 1 on the principle diagonal and 0.5 on the off-diagonal. For random graphs, we first generate a sparse matrix X with randomly chosen nonzero elements equal to ± 1 and then set $W^* = I + X^T X$; I is added to ensure that W is positive definite. We control the number of non-zeros in X so that the resulting W^* has approximately $2p$ nonzero elements. S is generated using $\frac{p}{2}$ samples.

4.1 Small-scale Experiments

We designed small-scale experiments to study the impact of blocking on the accuracy of GINCO, and to compare GINCO to QUIC, its state-of-the art ℓ_1-penalized counterpart. For GINCO, we used a MATLAB prototype; for QUIC, we downloaded the mex code, as suggested in [6]. Both experiments were run on a single node of the **oxygen** cluster (Section 4.2) using MATLAB (version number 7.6.0.324).

For QUIC, we set the subgradient tolerance threshold to 10^{-6}, the maximum Newton iterations to $10p$, and

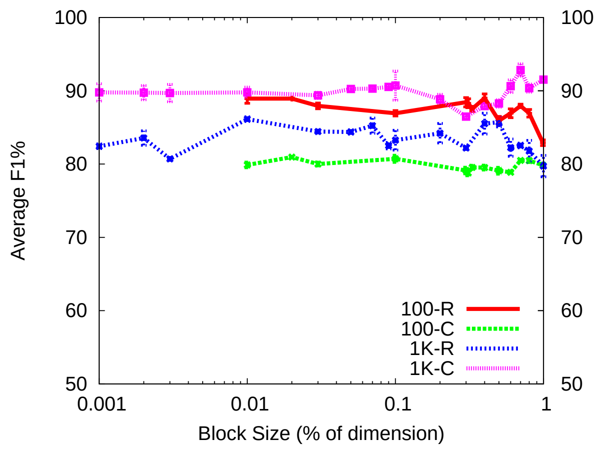


Figure 3: Accuracy ($F_1\%$) of GINCO with varying block sizes for 100-R, 100-C, 1K-R, and 1K-C. For each block size, experiments were run on 100 different simulated datasets — mean and standard deviation are provided. Block sizes are reported on the x-axis as a % of the dimension of the graph; for example, for 100-R, multiply the x-axis by 100 to get the actual block size.

the regularization parameter λ to $c \log p$, respectively; c is a tuning constant selected by 10-fold cross validation. For GINCO, we set $b_f = b_b = b$, and vary b. We set $\epsilon_f = \frac{c(d/b)\log p}{n}$, where d is the maximum degree of the graph, c is a tuning parameter selected by 10−fold cross-validation, $\epsilon_b = 0.5$ and $\epsilon_r = 10^{-6}$. Finally, we used 100 and 1000 vertex ($p = 100, 1000$) random (100-R, 1K-R) and chain (100-C, 1K-C) graphs.

We first generated 100 simulated datasets for each of the 4 graphs mentioned above and ran QUIC and GINCO using the settings described above on these datasets; for GINCO, this meant running 100 experiments for each block size. We then computed the mean and standard deviation of F_1 scores (per block size for GINCO) for each graph. Table 1 summarizes the results of our accuracy studies with both QUIC and GINCO. For GINCO, as we ran experiments with multiple block sizes, Table 1 reports the block sizes at which the best and the worst average F_1 scores (mean over 100 runs) were obtained. For example, for graph 100-R, GINCO at block size 99 had the worst mean F_1 score when averaged over the 100 simulated datasets. Figure 3 shows the mean and standard deviations of F_1 scores of GINCO for each block size. When taken together, Table 1 and Figure 3 show that GINCO, for any block size, outperforms QUIC consistently in terms of selection accuracy.

4.2 Large-scale Experiments

We implemented a shared- and distributed-memory parallel version of GINCO in C++. For shared-memory parallelism, we use PFunc [7], a lightweight

Name	Type	NNZ	QUIC	
			time	$F_1\%$
1K-R	Random	2074	6.23	78.83
1K-C	Chain	2996	6.03	77.68
4K-R	Random	7900	264.72	78.59
4K-C	Chain	11996	123.23	76.67
10K-R	Random	20516	4913.78	80.55
10K-C	Chain	29996	2997.24	79.67

Table 2: Characteristics of the graphs used in Section 4.2. "K" stands for *kilo*; for example, 1K-R is a 1000 vertex graph. We also list the running times (in seconds) and $F_1\%$ for QUIC on these graphs — the settings with which these numbers were obtained are given in Section 4.1.

and portable library that provides C and C++ APIs to express task parallelism. For distributed-memory parallelism, we use MPI [11, 12], a popular library specification for message-passing that is used extensively in high-performance computing. We ran scalability experiments on `oxygen`, a six-node cluster of dual-socket, quad-core Intel® Xeon™ E5410 machine with 32GB of RAM per-node running Linux Kernel 2.6.31-23 (total 48 cores). For compilation, we used GCC 4.4.1 with: "`-O3 -fomit-frame-pointer -funroll-loops`" in addition to PFunc 1.02 and OpenMPI 1.4.5. All the graphs used in this section were generated using the model specified in Section 4.1. Table 2 gives detailed information about each graph used in our experiments. Note that the number of vertices discovered in each experiment is equal to the number of non-zeros (NNZ) in the graph. Finally, all the numbers reported were averaged over three runs.

Parallel Scaling Figure 4 depicts results of the scaling studies conducted on the six graphs listed in Table 2 [2]. In both the pure PFunc (multi-threaded) and pure MPI case, we get near-linear speedup for all the graphs (Figures 4(a) and 4(b)). Of these, 1K-C and 1K-R are lower performing as they do not have sufficient computation to sustain linear speedups when the number of threads/processes is ≥ 4. For example, 4K-C, 4K-R, 10K-C, and 10K-R all achieve $\geq$7.5x speedup over 8 threads (Figure 4(a)), whereas 1K-C and 1K-R only achieve 6.2x and 6.8x speedup. Another factor in not achieving linear speedup is that the updates to Σ have a low compute-to-memory ratio and require global communication (Section 3.3). Although global communication can be avoided by replicating Σ on all the nodes, it is not advised as memory quickly becomes a bottleneck as graph sizes increase. For example, a 100K vertex graph needs 100GB of memory *per-node* if the matrices are not partitioned. Figure 4(c) shows the speedups when running both PFunc and MPI simultaneously. For 4K-C, 4K-R, 10K-C, and 10K-R, we achieve speedups ≥ 3.5 over 6 nodes (48 cores), whereas 1K-C and 1K-R achieve lower speedups due to the small problem size. Hybrid speedups are slightly lower than the PFunc-only and MPI-only speedups because of Amdahl's Law — all of the MPI communication happens in a single thread, which affects speedup. This is a temporary restriction as, going forward, many MPI implementations will start offering increased support for multi-threading. Note that hybridization is necessary (as opposed to running MPI everywhere) to fully exploit multi-core machines and to reduce the communication between processes, which is expensive. Finally, as the problem size increases, the sequential portions of GINCO take up a smaller portion of the overall time, which mainly consists of the $O(p^2)$ candidate evaluations; therefore, 4K-R, 4K-C, 10K-R, and 10K-C consistently achieve good speedups.

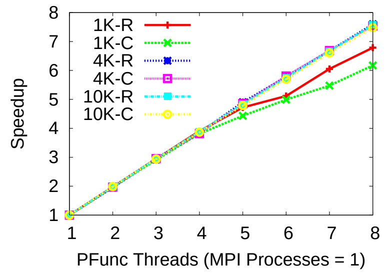
[2]Baseline time for Figures 4(a) and 4(b) is the sequential running time, whereas for Figure 4(c), the baseline time is the 8-thread running time.

Name	Block=NNZ		Block=*best*		
	Time	F_1 (%)	Time	F_1(%)	size
1K-R	.03	93.62	0.03	93.62	NNZ
1K-C	.03	80.61	5.3	86.51	4
4K-R	.28	98.23	4.5	98.27	128
4K-C	.29	83.95	8.4	87.31	128
10K-R	4.0	98.05	4.0	98.05	NNZ
10K-C	1.6	84.65	34.3	89.6	512

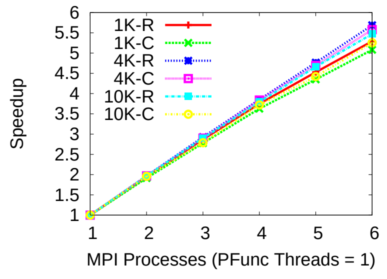
Table 3: Accuracy ($F_1\%$) and running times for (in seconds, 1 MPI process and 8 threads per-process) candidate selection of using NNZ as the block-size is compared to the best accuracy obtained for the same graphs.

Memory scaling Our implementation, keeps only one *partitioned* copy of Σ and S in memory — W is fully replicated, but as it is sparse, we store it as (i, j, v) triplets, which takes up little space. Therefore, the total memory needed by GINCO is *almost* independent of the number of nodes and threads; as more nodes are added, the memory usage per-node decreases along with the partition size of S and Σ. The total memory usage (over all processes/threads) for the graphs in Table 2 is: 1K-R and 1K-C(7MB), 4K-R and 4K-C (100MB), and 10K-R and 10K-C (700MB), which demonstrates the memory scalability of GINCO.

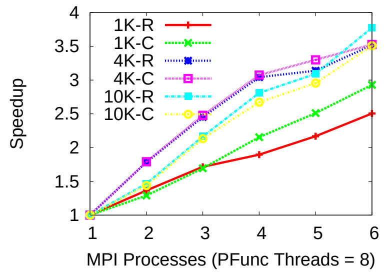
Block Scaling Block selection of candidates, either in the forward phase or the backward phase, has the same effect on running time of GINCO as parallelization, modulo the accuracy of estimation. To characterize the effect of block selection, we conducted experiments on the graphs listed in Table 2 — these results are summarized in Figure 5. We used increasing powers of 2 as the block sizes for the forward phase (from 1 to 512) — the backward phase block size was set to 1. All experiments were conducted on a single node of **oxygen** with 8 threads; the times reported are



(a) Multi-threaded



(b) MPI



(c) Hybrid

Figure 4: Speedups of GINCO for the six different graphs described in Table 2 are shown in 4(a), 4(b), and 4(c). 1K-R and 1K-C were run with block size 1, 4K-R and 4K-C with block size 4, and 10K-R and 10K-C with block size 32. The baseline times (in seconds) for Figures 4(a), 4(b), and 4(c) are: 1K-R (66,66,9.6), 1K-C (120,120,19.4), 4K-R (1039,1039,136.7), 4K-C (2032,2032,269.6), 10K-R (2527,2527,333.6), and 10K-C (4219,4219,563.8), respectively.

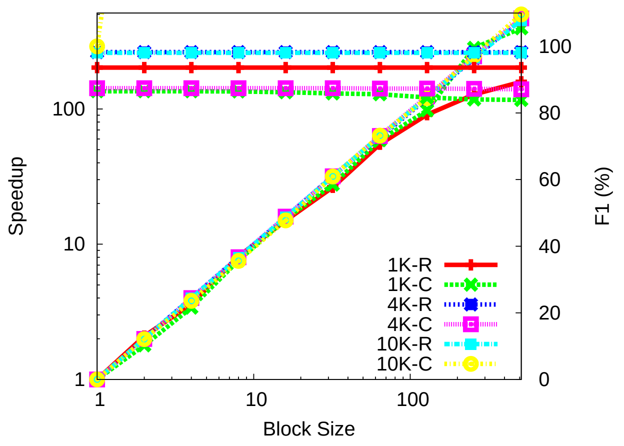


Figure 5: Performance of GINCO with varying block sizes for graphs listed in Table 2. Speedup is depicted on the left *y-axis* and accuracy of estimation is depicted on the right *y-axis*. Baseline times (in seconds, 1 MPI process and 8 threads per-process) are: 1K-R (9.8), 1K-C (18), 4K-R (538), 4K-C (1046), 10K-R(10232), and 10K-C(17150).

only for candidate selection *not* the total running time. The best accuracy listed is for the highest block size at which that accuracy was achieved. For all the graphs, as we increase the block size, we see a linear speedup in candidate selection at the cost of little or no loss in accuracy of estimation. For speedups, the only outlier is 1K-R, whose speedup for block sizes 256 and 512 is sub-linear as there are only 537 non-zeros. For random graphs, we saw nearly no loss in accuracy as the block sizes were increased; for chain graphs, the loss in accuracy was at most 1%. This result shows us that it is possible to use GINCO to solve high-dimensional problems with large block sizes relatively quickly. Table 3 compares the running times and accuracies obtained when the block size is set to NNZ of each graph with the best accuracies for that graph. Notice that 1K-R and 10K-R incur no loss in accuracy when the block size is NNZ; the chain graphs incur $\leq 6\%$ loss when compared to the best accuracy. However, the improvement in running times when the block size is NNZ is significant — we are able to obtain two orders of magnitude speedup over smaller block sizes at the cost of little or no loss in accuracy. The only outlier is 10K-R, which takes 6 seconds as using NNZ as block size triggered multiple backward phases. Interestingly, even when block size is set to NNZ, GINCO achieves better accuracy than QUIC (see Table 2), while running to completion significantly faster than QUIC. For example, GINCO, when run *sequentially* with block size of NNZ, requires 42.38 and 58.2 seconds to run to completion for 10K-R and 10K-C; QUIC requires 4913.3 and 2997.24 seconds, respectively.

5 Conclusion and Future Work

In this paper, we presented GINCO, a parallel, block greedy method for sparse inverse covariance estimation. Through extensive empirical analysis, we have demonstrated that GINCO is able to best the accuracy and running times of its competing ℓ_1-penalized counterparts. We have also demonstrated that by using blocking, we significantly reduced the running time GINCO at the cost of little or no loss in accuracy. Finally, we have presented the detailed design of our massively-parallel version of GINCO that is able to exploit both shared- and distributed-memory machines. Our parallel implementation is able to discover the network structure of random and chain graphs with 25,000 dimensions in ≤ 5 minutes using $\leq 10GB$ of main memory with high accuracy on a single 8-core machine — solutions to a problem of this ultra-high dimension have not been reported in literature.

GINCO currently implements a serialized version of block updates to Σ; this strategy is not optimal for medium or large block sizes. In the future, we would like to evaluate the alternative strategies to update Σ (see Section 3.3) to further improve the performance of GINCO for medium and large block sizes.

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[2] J. Duchi, S. Gould, and D. Koller. Projected subgradient methods for learning sparse gaussians. In *Proceedings of the Twenty-fourth Conference on Uncertainty in AI (UAI)*, 2008.

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

[4] Cho-Jui Hsieh, Matyas A. Sustik, Inderjit S. Dhillon, and Pradeep Ravikumar. Sparse inverse covariance matrix estimation using quadratic approximation. In J. Shawe-Taylor, R.S. Zemel, P. Bartlett, F.C.N. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2330–2338. http://nips.cc/, 2011.

[5] Ali Jalali, Christopher C. Johnson, and Pradeep D. Ravikumar. On learning discrete graphical models using greedy methods. *Advances in Neural Information Processing Systems (NIPS) and extended arxiv version*, 2011.

[6] Christopher C. Johnson, Ali Jalali, and Pradeep D. Ravikumar. High-dimensional sparse inverse covariance estimation using greedy methods. *Journal of Machine Learning Research - Proceedings Track*, 22:574–582, 2012.

[7] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. PFunc: Modern Task Parallelism For Modern High Performance Computing. In *ACM/IEEE conference on Supercomputing*, 2009.

[8] Lu Li and Kim chuan Toh. An inexact interior point method for l1-regularized sparse covariance selection. Technical report, National University Of Singapore, 2010.

[9] A.C Lozano, G Swirszcz, and N Abe. Group orthogonal matching pursuit for variable selection and prediction. *Neural Information Processing Systems Conference (NIPS)*, 22, 2009.

[10] S.G Mallat and Zhang. Z. Matching pusuits with time-frequency dictionaries. *IEEE Transcations on Signal Processing*, 41:3397–3415, 1993.

[11] Message Passing Interface Forum. MPI, June 1995. `http://www.mpi-forum.org/`.

[12] Message Passing Interface Forum. MPI-2, July 1997. `http://www.mpi-forum.org/`.

[13] O.Banerjee, L. El Ghaoui, and A. d'Aspremont. Model selection through sparse maximum likelihood estimation for multivariate gaussian or binary data. *Journal of Machine Learning Research*, 9:485–516, March 2008.

[14] Katya Scheinberg, Shiqian Ma, and Donald Goldfarb. Sparse inverse covariance selection via alternating linearization methods. *CoRR*, abs/1011.0097, 2010.

[15] Katya Scheinberg and Irina Rish. 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*, ECML PKDD'10, pages 196–212, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] V. Sindhwani and A.C. Lozano. Non-parametric group orthogonal matching pursuit for sparse learning with multiple kernels. *Neural Information Processing Systems Conference (NIPS)*, 2011.

[17] T. Zhang. Adaptive forward-backward greedy algorithm for sparse learning with linear models. *Neural Information Processing Systems (NIPS)*, 21, 2008.