Scaling of Union of Intersections for Inference of Granger Causal Networks from Observational Data

Mahesh Balasubramanian*, Trevor D. Ruiz†, Brandon Cook‡, Mr Prabhat‡
Sharmodeep Bhattacharyya†, Aviral Shrivastava*, Kristofer E. Bouchard‡§
*Arizona State University, Arizona, USA, †Oregon State University, Oregon, USA
‡Lawrence Berkeley National Laboratory, Calfornia, USA, § Corresponding Author
Email: mbalasubramanian@asu.edu, kebouchard@lbl.gov

***Abstract*—The development of advanced recording and measurement devices in scientific fields is producing high-dimensional time series data. Vector autoregressive (VAR) models are well suited for inferring Granger-causal networks from high dimensional time series data sets, but accurate inference at scale remains a central challenge. We have recently introduced a flexible and scalable statistical machine learning framework, Union of Intersections (UoI), which enables low false-positive and low false-negative feature selection along with low bias and low variance estimation, enhancing interpretation and predictive accuracy. In this paper, we scale the UoI framework for VAR models (algorithm UoI_{VAR}) to infer network connectivity from large time series data sets (TBs). To achieve this, we optimize distributed convex optimization and introduce novel strategies for improved data read and data distribution times. We study the strong and weak scaling of the algorithm on a Xeon-phi based supercomputer (100,000 cores). These advances enable us to estimate the largest VAR model as known (1000 nodes, corresponding to 1M parameters) and apply it to large time series data from neurophysiology (192 neurons) and finance (470 companies).**

I. Introduction

The growth of the Internet and social media applications has paved the way for the development of highly sophisticated machine learning and statistical data analysis tools. Further scientific data collection strategies have grown exponentially over the years by innovation in the field of sensors and advanced data collection methods. Many fields such as genetics, mass spectrometry, and neuroscience [1]–[4] now have the means of collecting big data through various devices and sensors [5]. In particular, advanced recording devices created as part of the BRAIN Initiative enable recording neural activities from hundreds to thousands of neurons for days at a time, generating TeraBytes and in some cases, PetaBytes of time series data [6], [7]. A challenge in such data sets is to infer the causal network that generated the time series data, and thus gain insight into scientific mechanisms of complex phenomena [3], [4], [6], [7]. Similarly, one may wish to understand the causal influences among companies from stock price time series [8].

Vector autoregressive (VAR) models are well suited for inference of Granger causal networks from such high-dimensional, multi-variate observational time series data. Introduced for the analysis of econometric time series, Granger causality is the amount of variance in one time series accounted for by the past of another time series [8]. Thus, from a statistical-machine learning perspective, the challenge of Granger causality is to accurately infer the existence (or not) and weight of directed edges between nodes in the network from noisy time series observations of the nodes. Although VAR models provide a flexible framework and are probabilistically tractable [9], scaling VAR inference to massive data sets is a major challenge due to unfavorable scaling of the problem size with the number of nodes or features in the network.

The Union of Intersections (UoI) framework developed in [10] is a powerful statistical-machine learning framework which has natural algorithmic parallelism. Methods based on UoI improve the selection of features (model selection) and estimation of the contribution of the selected features (model estimation). The main mathematical innovations of UoI are 1) creating a family of potential model supports through an intersection operation for a range of regularization parameters in model selection, and 2) combining the above-computed supports with a union operation so as to increase prediction accuracy on held out data in model estimation. Theoretical and extensive numerical evaluation of a sparse linear regression algorithm based on UoI (UoI_{LASSO}) presented in [10] shows state of the art feature selection (low false positives and low false negatives) and feature estimation (low-bias, low-variance) compared with many regression algorithms (e.g., LASSO, SCAD and Ridge). This is done without formulating a non-convex optimization problem. Similarly, the statistical performance evaluation for the UoI implementation for VAR models, UoI_{VAR}, presented in [11], shows less bias and superior selection accuracy when compared to LASSO and non-convex optimization method such as the minimax convex penalty (MCP). Note that non-convex optimizations (as utilized in e.g., SCAD and MCP) are extremely challenging for implementation in the multi-nodal distributed computing paradigm [12]. In contrast, methods relying on convex optimization (e.g., UoI_{LASSO} and UoI_{VAR}) can utilize the Alternating Direction Methods of Multipliers (ADMM) [13] for solving the constrained convex optimization in a distributed manner in a multi-nodal computing environment. Thus, while our prior works establish the state-of-the-art statistical properties of UoI_{LASSO} and UoI_{VAR}, several challenges remain in the application of these methods to large data sets.

In this paper, we develop a scalable implementation of UoI_{VAR} to infer Granger causal networks from high dimen-

sional time series data sets. UoI_{VAR} builds upon UoI_{LASSO}: thus, in order to better understand the scalability of UoI_{VAR}, we start by studying the scalability of LASSO-ADMM in UoI_{LASSO}, and then proceed to UoI_{VAR}. We reveal the computation, communication, input/output bottlenecks for UoI_{LASSO} and UoI_{VAR}, and develop solutions to mitigate them. To accommodate the randomness required for bootstrap sampling used in UoI methods, we introduce a Random Data Distribution strategy to efficiently manage data read and distribution time from large data sets. We introduce distributed Kronecker product and vectorization strategies for UoI_{VAR}. Above and beyond parallelization of optimization through ADMM, we analyze both algorithms for natural parallelism and evaluate our multi-node implementation. With high-dimensional synthetic data sets (1000 nodes or features) we demonstrate the weak and strong scaling of each algorithm. Due to the unfavorable scaling of the problem size with the number of nodes in the network ($\approx p^3$ for p nodes), it is rare to encounter VAR models with more than 50 nodes or features. In the statistical literature on high-dimensional VAR modeling, numerical experiments considered adequate to represent typical data applications are around 30 nodes, and larger-scale data applications are on the order of a few hundred nodes: [14] used simulated data of 30 nodes and [15] used monthly home-price appreciation (HPA) data set with 352 nodes. Finally, we analyze a real world neurophysiology data set of 192 neurons and a stock market data set (S&P 500 index in 2013-2014) with 470 companies. Our scaling to 1000 nodes (1M parameters) reflects an $\approx$ 3-fold increase in network scale ($\approx$ 9 fold more parameters), while doing so in the context of a superior inference algorithm.

II. Methods

A. Formal Statistical Description

Let us consider n samples of input data $((Y_1, X_1)$, ...,$(Y_n, X_n))$ with univariate response variable Y and p-dimensional predictor variable X. The linear regression model for this input data is generated as:

$$Y = X\beta + \epsilon \tag{1}$$

where $Y = (Y_1, ..., Y_n), X$ is a $n \times p$ design matrix; $\epsilon = (\epsilon_1, ..., \epsilon_n)$ are random noise terms with $\epsilon \sim N(0, \sigma^2 \mathbf{I}_n)$. Let $S = \{i : \beta_i \neq 0\}$ be the non-zero coefficient set of β.

The LASSO regression algorithm with penalization parameter $\lambda > 0$ minimizes the following constrained convex optimization problem with respect to β:

$$\hat{\beta} = argmin_\beta ||Y - X\beta||^2 + \lambda ||\beta||_1 \tag{2}$$

Here, the first term on the right-hand side penalizes the error of the predictions, while the second term penalizes the L_1 norm of the parameter vector β, setting some values of β to zero.

B. Model Selection and Model Estimation

For every bootstrap sample T^k the LASSO estimates (${}^j\hat{\beta}^k$) are computed (here, using the Alternating Direction Method of Multipliers (ADMM)), see equation 6) across different regularization parameter values, λ_j. For each bootstrap sample, the support (S_j^k) are the non-zero values of the estimates calculated by LASSO-ADMM. It is known [10] that the LASSO estimator is prone to false positives for a decrease in penalizing parameter (λ): i.e., it includes more parameters than are in the model. To mitigate this, in UoI_{LASSO} the support associated with a given λ, S_j is taken as the intersection of the supports across bootstrap samples:

$$S_j = \bigcap_{k=1}^{B_1} S_j^k \tag{3}$$

This is done for each value of (λ), creating a family of potential model supports $\mathbf{S} = [S_1, S_2, ..., S_q]$.

A number B_2 of bootstrap samples are used to compute the model estimates. For each potential support from the model selection step (Algorithm 1 line 18), the unbiased Ordinary Least Squares (OLS) estimator is used to estimate the associated model from each of the B_2 bootstrap samples. The algorithm then computes a *Union* of supports by averaging the OLS estimates that optimize predictions, which reduces variance and performs a union operation on the supports optimizing predictions. The variable set post-union (averaging) can be represented as (approximately):

$$S_{UoI} = \bigcup_{l=1}^{B_2} S_{jl} = \bigcup_{l=1}^{B_2} \bigcap_{k=1}^{B_1} S_{jl}^k \tag{4}$$

C. Distributed Constrained Convex Optimization by Alternating Direction Method of Multiplier

The core calculations in both UoI_{LASSO} and UoI_{VAR} involve solving a constrained convex optimization problem (L_1 regularized linear regression). Here, we use the Alternating Direction Method Multiplier (ADMM) [13] to minimize the loss function (Equation 2). LASSO-ADMM solves the dual problem in form of equation 5:

$$\begin{aligned} \text{minimize } & f(x) + g(z) \\ \text{subject to } & x - z = 0 \\ \text{where, } & f(x) = (1/2)||Y - X\beta||_2^2; \\ & g(z) = \lambda ||\beta||_1 \end{aligned} \tag{5}$$

where x $\in \mathbb{R}^n$, z $\in \mathbb{R}^m$, and f and g are convex. The LASSO-ADMM algorithm consists of an x minimization, z minimization followed by a dual variable update. The separation of minimization over x and z allows for the separate decomposition of f and g. Here, x and z can be updated in sequential or alternating computations which gives the name alternating direction. In the distributed ADMM algorithm, each compute core is responsible for computation of its own objective (x) and constraint (z) variables and its quadratic term

Algorithm 1 UoI_{LASSO} $(InputData(X, y) \in \mathbb{R}^{n\times(p+1)}, \lambda \in \mathbb{R}^q,\ B_1,\ B_2)$

Model Selection
for $k = 1$ to B_1 **do**
  Generate bootstrap sample $T^k = (X_T^k, Y_T^k)$
  for $\lambda_j \in \lambda$ **do**
    Compute LASSO estimate ${}^j\hat{\beta}^k$ from T^k
    Compute support $S_j^k = \{i\}$ s.t ${}^j\hat{\beta}_i^k \neq 0$
  end for
end for
for $j = 1$ to q **do**
  Compute Bootstrap-LASSO support for $\lambda_j : S_j = \bigcap_{k=1}^{B_1} S_j^k$ (as in equation 3)
end for
Model Estimation
for $k = 1$ to B_2 **do**
  Generate bootstrap samples for training and evaluation:
  training $T^k = (X_T^k, Y_T^k)$
  evaluation $E^k = (X_E^k, Y_E^k)$
  for $j = 1$ to q **do**
    Compute OLS estimate $\hat{\beta}^k_{S_j}$ from T^k
    Compute loss on $E^k : L(\hat{\beta}^k_{S_j}, E^k)$
  end for
  Compute best model for each bootstrap sample:
  $\hat{\beta}_S^k = {}_{\hat{\beta}^k_{S_j}}\ L(\hat{\beta}^k_{S_j}, E^k)$
end for
Compute averaged model estimates $\hat{\beta}^* = \frac{1}{B_2}\sum_{k=1}^{B_2}\hat{\beta}_S^k$ (as in equation 4)
Return: $\hat{\beta}^*$

Algorithm 2 UoI_{VAR} $(InputData(X_1, \ldots, X_N)^T \in \mathbb{R}^{N\times p)}, \lambda \in \mathbb{R}^q,\ B_1,\ B_2)$

Model Selection
for $k = 1$ to B_1 **do**
  Generate bootstrap sample $T^k = (X_{T1}^k, \ldots, X_{TN}^k)$
  Construct $(\mathbf{Y}_T^k, \mathbf{X}_T^k)$ (as in equations 7 - 8)
  Construct $Y_T^k = \text{vec}\mathbf{Y}_T^k$ and $X_T^k = (\mathbf{I} \otimes \mathbf{X}_T^k)$
  for $\lambda_j \in \lambda$ **do**
    Compute LASSO estimate ${}^j\hat{\beta}^k$ from (X_T^k, Y_T^k)
    Compute support $S_j^k = \{i\}$ s.t ${}^j\hat{\beta}_j^k \neq 0$
  end for
end for
for $j = 1$ to q **do**
  Compute Bootstrap-LASSO support for $\lambda_j : S_j = \bigcap_{k=1}^{B_1} S_j^k$ (as in equation 3)
end for
Model Estimation
for $k = 1$ to B_2 **do**
  Generate bootstrap samples for training and evaluation:
  training $T^k = (X_{T1}^k, \ldots, X_{TN}^k)$
  evaluation $E^k = (X_{E1}^k, \ldots, X_{EN}^k)$
  Construct $(\mathbf{Y}_T^k, \mathbf{X}_T^k)$ (as in equations 7 - 8)
  Construct $(\mathbf{Y}_E^k, \mathbf{X}_E^k)$ (as in equations 7 - 8)
  Construct $Y_T^k = \text{vec}\mathbf{Y}_T^k$ and $X_T^k = (\mathbf{I} \otimes \mathbf{X}_T^k)$
  Construct $Y_E^k = \text{vec}\mathbf{Y}_E^k$ and $X_E^k = (\mathbf{I} \otimes \mathbf{X}_E^k)$
  for $j = 1$ to q **do**
    Compute OLS estimate $\hat{\beta}^k_{S_j}$ from T^k
    Compute loss on $E^k : L(\hat{\beta}^k_{S_j}, E^k)$
  end for
  Compute best model for each bootstrap sample:
  $\hat{\beta}_S^k = {}_{\hat{\beta}^k_{S_j}}\ L(\hat{\beta}^k_{S_j}, E^k)$
end for
Compute averaged model estimates $\hat{\beta}^* = \frac{1}{B_2}\sum_{k=1}^{B_2}\hat{\beta}_S^k$ (as in equation 4)
Partition $\hat{\beta}^*$ and rearrange into $(\hat{A}_1, \ldots, \hat{A}_d)$ and $\hat{\mu}$
Return: $(\hat{A}_1, \ldots, \hat{A}_d)$ and $\hat{\mu}$

$(f(x))$ is updated so that all the cores converge to a common value of estimates. To ensure a good scalability, the ordinary least squares (OLS) is implemented using LASSO-ADMM algorithm for model estimation by setting regularization parameter λ to 0, thereby making g in equation 5 equal to 0.

D. UoI_{LASSO} Algorithm

A high-level overview of the UoI_{LASSO} algorithm, shown in Algorithm 1, consists of two Map-Solve-Reduce steps (Figure 1). The algorithm takes multiple random bootstrap subsamples of the input data (Map) and distributes it across different computing cores. Next, LASSO and OLS ($Solve$) use the distributed data and solve the convex optimization. The resultant estimates are then combined by intersection and union operations ($Reduce$). The $Reduce$ step in model selection performs a feature compression by intersection operation of supports across bootstraps. The $Reduce$ step in model estimation performs a feature expansion by averaging (union operation) the OLS estimates across different model supports.

E. UoI_{VAR} Algorithm

The UoI_{LASSO} implementation can be adapted to sparse estimation of vector autoregressive model parameters from high-dimensional time series data. In this case the input data is a vector time series $\{X_t\}_{t=1}^N$ generated by a vector autoregressive process of order d, $VAR(d)$:

$$X_t = \sum_{j=1}^{d} A_j X_{t-j} + U_t \tag{6}$$

where $X_t \in \mathbb{R}^p$, the process has p-dimensional Gaussian disturbances $U_t \overset{iid}{\sim} \mathbb{N}_p(0, \Sigma)$. The stability of the process is expressed by the constraint $\mathbf{det}(I - \sum_{j=1}^{d} A_j z^j) \neq 0 \quad \forall\, |z| \leq 1$.

Equation 6 provides a model for the data which can be written as a multivariate least squares problem with correlated errors of the form $\mathbf{Y} = \mathbf{XB} + \mathbf{E}$. In particular, the response is the $(N - d) \times p$ matrix

$$\mathbf{Y} = (X_N, X_{N-1}, \ldots, X_{d+1})^T \tag{7}$$

and the regressors are lagged values represented in the $(N - d) \times (dp)$ matrix

$$\mathbf{X} = \begin{pmatrix} X'_{N-1} & X'_{N-2} & \cdots & X'_{N-d} \\ X'_{N-2} & X'_{N-3} & \cdots & X'_{N-(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ X'_{d} & X'_{d-1} & \cdots & X'_{1} \end{pmatrix} \quad (8)$$

and the coefficient matrix is $\mathbf{B}' = (A_1 A_2 \dots A_d)$. One estimation strategy is to vectorize the problem as shown in equation 9 and apply ordinary least squares to estimate the entries of the A_j matrices.

$$\text{vec}\mathbf{Y} = (\mathbf{I} \otimes \mathbf{X})\, \text{vec}\mathbf{B} + \text{vec}\mathbf{E} \quad (9)$$

Equation 9 then has the same form as equation 1. Noting this correspondence, estimation with sparsity in high-dimensional time series can be accomplished by first rearranging the multivariate least squares problem and then solving the LASSO problem (equation 2) for the resulting rearrangement.

The UoI implementation, shown as Algorithm 2, is consequently similar to UoI_{LASSO}, but with a bootstrap method appropriate for capturing temporal dependence in the input data (here, using a block bootstrap) and large matrix operations required to obtain a problem of the form shown in equation 2. Aside from these modifications, the Algorithm 2 is the same as UoI_{LASSO} Algorithm 1.

III. Scaling UoI_{LASSO} and UoI_{VAR}

UoI_{LASSO} and UoI_{VAR} exhibit a high degree of algorithmic parallelism. In each of the model selection and model estimation steps, the bootstrap subsamples (B_1 and B_2) can be parallelized, referred to as P_B parallelization. Additionally, parallelization over regularization parameters (λ_j) can be used (referred to as P_λ parallelization). An important point to consider is that the model selection and model estimation must occur in sequential order and cannot be parallelized.

A. Challenges in achieving parallelism

To achieve accuracy in selection and estimation, UoI-based methods utilize the notion of stability to perturbations, in this case multiple random resampling of the data. UoI_{LASSO}, in particular, requires random sub-samples generated from the data set in selection and estimation Map steps (Figure 1). In our initial experiments we have seen that repeated access to the data file in the file system takes a lot of data access time, and dividing the data set into chunks for faster access reduces selection and estimation accuracy [10]. Due to the smaller sized data set in UoI_{VAR} (recall that VAR problem scales $\approx p^3$), the centralized distribution strategy was adopted to distribute the data to compute cores. We found that the main challenge for UoI_{VAR} 'Map' is computing the Kronecker product and vectorization steps in a distributed method. As far as we are aware of, there has been no prior methods to distribute the Kronecker product computation and vectorization for VAR models.

B. Randomized Data Distribution Design using HDF5

1) UoI_{LASSO}: We introduce randomized data distribution strategy for UoI_{LASSO} to improve the data read time from the file system and reduce the data distribution time to the compute cores. The synthetic data set matrices used in this evaluation have the "Samples" in rows and "Features" in columns. The data set size is the problem size for UoI_{LASSO}. We use `HDF5` application program interface for data input/output. `HDF5` offers parallel reading of the input file, albeit in contiguous chunks. The library does not provide a random reading of input data without reading the file multiple times in a loop. To parallelize this operation, we introduce a novel randomized data distribution technique. First, the data is read in parallel from the input file into the computing cores in contiguous blocks. As shown in Figure 1, T0 or *Tier0* is the source HDF5 file. The contiguous reading by all the processes is done in T1, *Tier1*, using HDF5 hyperslabs [16]. *Tier0* and *Tier1* data distribution use an underlying HDF5-parallel library for parallel accesses and hyperslab creation. By creating hyperslabs, the application can read the data file and load them into the memory space created on each compute core. After loading the data from the input file, we employ MPI one-Sided communication to randomly distribute the subsamples (T2, *Tier2*). The input data is distributed via row-wise block-striping to distribute the samples. If N is the number of samples, p is the feature size, and B is the number of cores, each core receives $\frac{N}{B}$ rows and p columns. Each core then solves the constrained convex optimization problem using LASSO-ADMM (equation 5) and is responsible for computing its own objective (x) and constraint (z), and the quadratic term is updated to converge to a common value of estimates.

2) UoI_{VAR}: Since UoI_{VAR} is a time series model, the input data for this algorithm exhibit temporal dependence. To maintain this dependence, a block bootstrap approach was adopted by randomly selecting time series blocks for every bootstrap subsample. The Algorithm 2 lines 5 and 21-22, requires a column stacking vectorization step to construct Y_T^k and an identity Kronecker product step to construct X_T^k. In the serial version of the algorithm a simple vectorization and Kronecker product functions can be invoked, but in a distributed-memory parallel paradigm, this is not possible. Unlike UoI_{LASSO}, the synthetic data sets for UoI_{VAR} are relatively small (in order of MegaBytes) and the problem is created in the *Kronecker product* and *vectorization* (lines 5) of Algorithm 2. The actual problem size increases in the order $\approx p^3$, where p is the number of features. Due to the small size of the data, the T1 parallel reading layer cannot be deployed. To overcome this issue, we have developed a distributed Kronecker product and vectorization strategy using `MPI` one-sided communication with the windows created by the *n_reader* processes: a small number of processes (usually equal to the number of samples based on the availability of resources) read the data file in parallel and creates windows for MPI-One sided communication for distributed Kronecker product and vectorization.

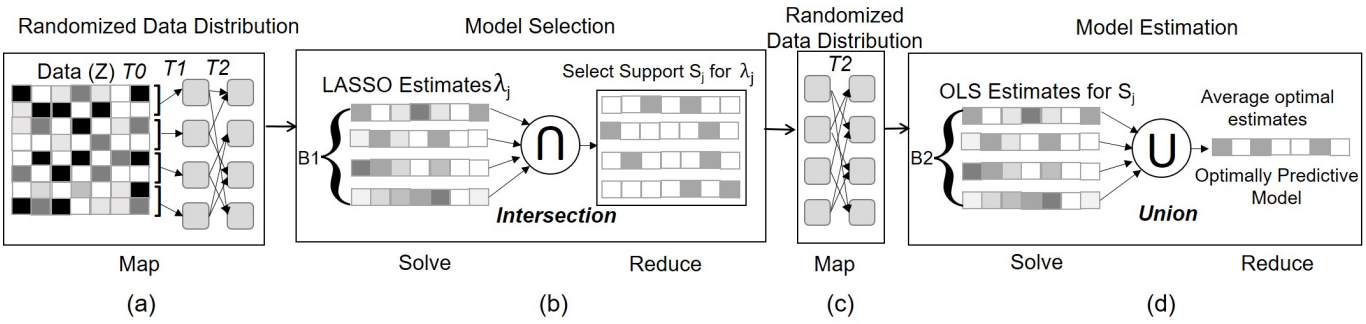

Fig. 1: (a) A Three-Tier (T0, T1 and T2) distribution strategy for randomized distribution of data set across the number of sample from the HDF5 data file to the cores of KNL. (b) Model Selection – LASSO ADMM is used to 'Solve' and **Intersection** operation is used as 'Reduce' to select family of support S_j. (c) Data randomization for cross validation where Tier2 random distribution is employed to randomly reshuffle the data. (d) Model Estimation – OLS is used to 'Solve' and **Union** operation is used to 'Reduce' to get an optimally predictive model.

The Kronecker product (Algorithm 2 line 5, 19, 20) $X_T^k = (\mathbf{I} \otimes \mathbf{X}_T^k)$, is an identity block diagonal matrix of $\mathbf{X}_T^k$ from equation 8. Similarly, vectorization of Y (Algorithm 2 line 5, 19, 20), $Y_T^k = \text{vec}\mathbf{Y}_T^k$ is from equation 7. Since $\mathbf{X}_T^k$ and $\mathbf{Y}_T^k$ are computed *a priori*, the cores holding these data structures create the MPI one-sided communication windows for building $(\mathbf{I} \otimes \mathbf{X}_T^k)$ and vec$\mathbf{Y}_T^k$. Since the LHS of equation 7 and 8 are the actual problem sizes (order of GBs and TBs), the communication strategy does not require explicit computation of the equation 7 and 8 on the computing cores. The main challenge is the increased communication time to create such large matrices as there are few cores (10s to 100s) holding the actual matrices to be distributed to hundreds of thousands of cores. This problem is quantitatively explained in the Weak Scaling sub-section of UoI_{VAR}, Section IV. Note that a conventional method, like computing the Knocker product and vectorization in a single core and distributing it to the other computing cores, is not possible due to the increased space to store the data and limited availability of space per node. Like UoI_{LASSO}, post Kronecker product and vectorization step, each core solves the convex optimization problem in equation 5 in a distributed manner.

IV. Results

The single node and multi-node runs for this paper were conducted on Cori Knights Landing (KNL) supercomputer at NERSC. Cori KNL is a Cray XC40 supercomputer consisting of 9,688 nodes of 1.4 GHz Intel Xeon Phi processors with a single socket 68 cores per node. The aggregated memory for a single node in KNL is 16GB MCDRAM and a 96GB DDR. The UoI_{LASSO} and UoI_{VAR} algorithms were implemented in C++ using Eigen3 library [17] for linear algebra computations and Intel-MKL library [18] for BLAS operations for UoI_{LASSO} to utilize the inbuilt Single Instruction Multiple Data (`SIMD`) directives. The `MPI` framework was used for parallelization and communication between the processes supported by OpenMP multithreading with `OMP_NUM_THREADS` as four, which showed better performance. The performance analysis setup for UoI_{LASSO} and UoI_{VAR} is shown in Table I. Related to this, recently, the optimal configuration for executing neural networks (AlexNet) was calculated in [19]. Although the model shown in [19] could potentially be applied in our context by including the structure of the design matrix (e.g., columns X rows, as well as sparse vs. dense) to find a theoretically better configuration, the practical configuration depends on the realities of the hardware.

For all the evaluations in this paper, synthetic data sets ranging from 16GB to 8TB were generated for UoI_{LASSO}, and data sets that generate problem sizes of 16GB to 8TB were generated for UoI_{VAR}. The experiments were carried out in two phases, Single Node performance and optimizations, and Multi-Node scaling. The feature size for UoI_{LASSO} is kept a constant at 20,101 features across data sets to study the effect of communication. For UoI_{VAR}, the data set features range from 356 for a 128GB problem size to 1000 features for 8TB problem size and the number of samples are twice the size of the features. We evaluate the algorithms for single node performance, exploiting algorithmic parallelism and multi-node scaling experiments. Due to limited resource availability of computing resource the multi-node scaling runs were performed with no P_B and P_λ parallelism and dedicating all the cores to distributed LASSO-ADMM computation.

A. Performance and Scaling of UoI_{LASSO}

1) Single Node Performance: The focus of single node performance analysis is to identify the potential bottlenecks in the program and optimize them. Post optimization, the performance improvement is calculated using a performance roofline model for both the program and the architecture (Xeon Phi) on which the program is executed. A $\approx 16GB$ data set with five selection and estimation bootstrap samples ($B_1 = B_2 = 5$) and eight regularization parameters(q) were chosen for single node optimization of the implementation.

Our initial analyses showed that the Matrix multiplication and Matrix-Vector product in LASSO-ADMM function were the bottlenecks. Execution of these operations showed very

Performance Analysis	Data Size (GB)	No. of cores (UoI_{LASSO})	No. of cores (UoI_{VAR})
Single Node	16	68	68
Weak Scaling	128	4,352	2,176
	256	8,704	4,352
	512	17,408	8,704
	1024	34,816	17,408
	2048	69,632	34,816
	4096	139,264	69,632
	8192	278,528	139,264
Strong Scaling	1024	17,408	4,352
		34,816	8,704
		69,632	17,408
		139,264	34,816

TABLE I: Performance Analysis setup for UoI_{LASSO} and UoI_{VAR}.

poor performance with native Eigen3 library on Cori KNL. To alleviate the poor performance we implemented the `BLAS` operations for matrix multiply and matrix-vector product using the Intel-MKL library. Figure 2 shows the runtime for single node run. Almost 90% of the runtime is dominated by computation and less than 10% by communication. All the `MPI` calls like `MPI_Bcast`, `MPI_Allreduce` etc., constitute the communication bar as shown in the Figure 2. More than 99% of the communication time comes from `MPI_Allreduce` call used to communicate the estimates by the distributed LASSO-ADMM function. MPI one-sided calls for distribution of the data is shown as 'Distribution', while parallel-HDF5 data loading and output saving is shown as 'Data I/O'. We analyzed the program in detail with Intel Advisor [20] tool for the performance of various sections of the code. The performance of matrix multiplication with Intel-MKL was 30.83 GFLOPS (Giga-Floating Point operations per second) with an arithmetic intensity (Floating point operations per byte of data moved from memory) of 3.59 FLOPs/Byte and the performance of matrix-vector multiplication was 1.12 GFLOPS with an arithmetic intensity of 0.32 FLOPs/Byte. Both the `BLAS` operations were found to be DRAM memory bound. The performance of the triangular solve function used by LASSO-ADMM function for matrix decomposition was 0.011 GFLOPS with an arithmetic intensity of 0.075 FLOPs/Byte.

2) Exploiting Algorithmic Parallelism: The innate algorithmic parallelism exhibited by the UoI_{LASSO} was exploited

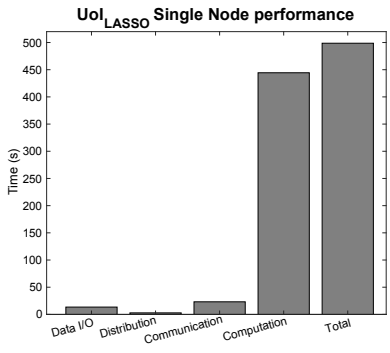

Fig. 2: UoI_{LASSO} runtime number using Intel-MKL linear algebra library with $B_1 = B_2 = 5$ and $q = 8$.

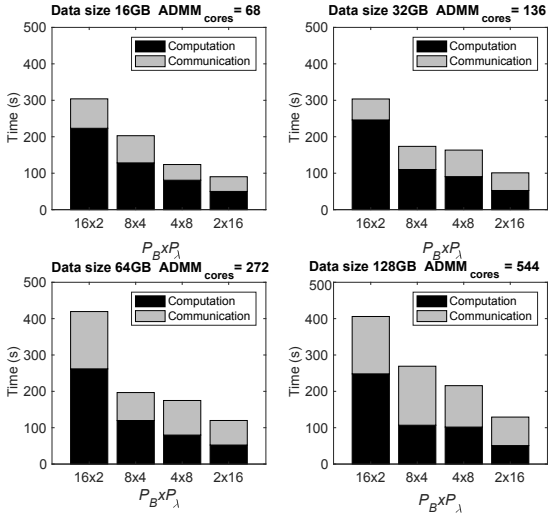

Fig. 3: Exploiting P_B and P_λ parallelism by increasing the data set and ADMM$_{cores}$ by a factor of 2.

by having bootstrap level (P_B), regularization parameter level (P_λ) and ADMM computation level parallelism. These runs were performed on lower end of data set spectrum, 16GB, 32GB, 64GB and 128GB with 2176, 4352, 8704 and 17,408 cores, respectively. The $P_B \times P_\lambda$ configuration used were 16×2, 8×4, 4×8 and 2×16 with $B_1 = B_2 = q = 48$ for all the runs. The data set size and the ADMM$_{cores}$ were doubled maintaining the parallelization configurations. The runtime of the different configurations are shown in Figure 3. Across various configurations the 2×16 has a better runtime. Also across the data set runs we can see a slight increase in the communication time for $ADMM_{cores} = 272$ and $ADMM_{cores} = 544$. This increase in the communication time was accounted by the `MPI_Allreduce` call from LASSO-ADMM implementation to collectively converge at an estimate value.

3) Comparison of Randomized Data Distribution Design with Conventional Distribution strategy: Conventional data distribution strategy involves reading from the data file by a single core using serial HDF5 by creating hyperslabs. The traditional methodology has three issues, namely: 1) it can read only a small chunk of data at a time, 2) it would repeatedly open the data file to read the data completely, and 3) it cannot store the loaded data due to limited space availability (aggregated memory of single KNL node is 96GB, but the data set size is in order of 100s of GBs and TBs). We implemented the conventional data distribution in C++ with serial HDF5 implementation. It should be noted that for UoI algorithms, different random bootstraps of data are required for model selection and model estimation steps (lines 3, and 14 of Algorithm 1). The comparison of data read and distribution time between our Randomized Data Distribution Design (contribution of this paper) and the conventional design is shown in Table II. The number of cores used for runs in Table II is based on Table I. From Table II, it is quantitatively evident that the data read time and distribution time for the

Data Size (GB)	Conventional Method		Randomized Data Distr.	
	Read time (s)	Distr. time (s)	Read time (s)	Distr. time (s)
16	204.71	1.276	11.3191	0.33
128	1200.81	17.596	0.52	5.718
256	2204.52	36.46	1.46	2.62
512	5323.486	74.274	8.043	3.64
1024	11,732.48	158.016	8.781	3.774

TABLE II: Randomized Data Distribution design improves the Data Read and Distribution time compared to Conventional Distribution method. Beyond 1TB data set size the conventional method's data read time crossed beyond 5 hours whereas Randomized Data Distribution read time was below 100 seconds.

conventional method is a bottleneck because of the issues discussed above. From Table II, it should be noted that the read time for the 16GB is higher than the larger data sets because it was not striped into OSTs.

4) Multi-node Scaling: The multi-node scaling analysis is carried out for weak scaling and strong scaling of UoI_{LASSO} implementation. Parallel reading of the input file becomes an issue for multi-node scaling runs as 1000s of cores try to read the data in parallel. In an unoptimized run, the read time takes 10s of minutes which can worsen with an increase in the data size and the number of nodes. For large data sets, the HDF5 input files are stripped into different Object Storage Targets (OSTs), explained in detail; in [21]. The files are stripped for 160 OSTs to achieve a faster reading time, making the data read time of very large data sets to a few seconds.

Weak Scaling: In weak scaling, the problem size associated with each compute core stays constant and additional computing cores are added when the size of the input data set increases. We maintain a factor of 2 for our weak scaling runs, meaning as the data set size is doubled the number of cores were also doubled (refer Table I). Figure 4 shows the weak scaling of UoI_{LASSO}. Since matrix multiplication contributes the most to the computation time, and since the problem size per compute core is almost the same across different configurations, we find that computation exhibits nearly ideal weak scaling with slight increase for 8TB. It is seen that the communication time scales proportional to the increase in the

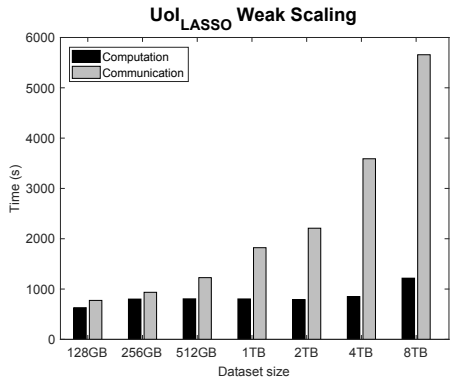

Fig. 4: Weak Scaling plot of UoI_{LASSO}. The problem size per node was kept fixed.

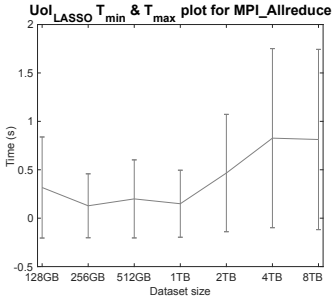

Fig. 5: T_{min} & T_{max} plot for UoI_{LASSO}.

core count. On further analysis of the communication time, we find that the `MPI_Allreduce` calls contributes almost 99% of the communication time. The error modeling of one MPI_Allreduce call for all the data points used for weak scaling as shown in Figure 5. The feature size of all the data sets is kept a constant at 20,101 features, so the array size for MPI_Allreduce communication is uniform across all the cores. The difference in T_{max} and T_{min} for the MPI_Allreduce indicates performance variability of communications. However, despite this we observe good scalability. As future work we are evaluating non-blocking MPI and asynchronous execution models to enable further scaling.

Strong Scaling: In strong scaling, the problem size to be analyzed is kept as 1TB and the number of computing cores is increased from 17,408 to 139,264 (refer Table I). Figure 6 shows the results of the strong scaling run. The computation time shows a decreasing trend across different configurations due to the increase in the number of cores for the same data set size. At 139,264 cores the computation goes below expected computation strong scaling trend, the reason being that the total size of the problem per core becomes small, which Intel-MKL library takes advantage of the AVX512 extensions making the matrix multiplication computed per core faster. The superlinear computation time can also be attributed to the reduced DRAM accesses due to a smaller chunk of data distributed per core. As seen in the weak scaling runs communication time increase with increasing number of

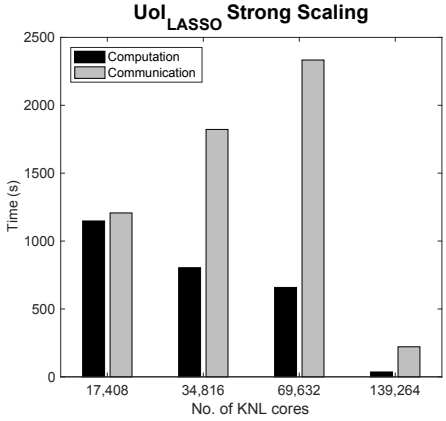

Fig. 6: Strong Scaling plot of UoI_{LASSO}. The problem size was kept fixed at 1TB.

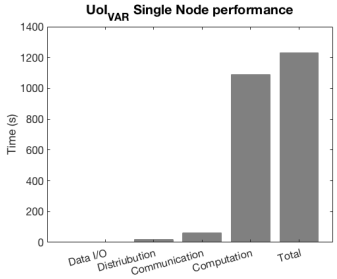

Fig. 7: UoI_{VAR} single node with $B_1 = B_2 = 5$ and $q = 8$.

cores, but beyond 69,632 cores the LASSO-ADMM converges faster making the communication time almost equal to the ideal strong scaling.

B. Performance and Scaling of UoI_{VAR}

1) Single Node Performance: The Algorithm 2 creates a high dimensional matrix by Kronecker product for each bootstrap subsample. The resultant matrix has a block diagonal structure with high sparsity. From Algorithm 2, if the input data is dense the sparsity of the problem can be calculated as $1 - \frac{1}{p}$, where p is the number of features of the input data set. A problem size of ≈16GB with $B_1 = B_2 = 5$ and $q = 8$ and number of lambda parameters $q = 8$ were chosen for single node optimization. For example, if a data set has 95 features, the resultant matrix post Kronecker product has a sparsity of 98.94%. So it is intuitive to exploit this sparsity by utilizing sparse linear algebra libraries. Figure 7 shows the single node run of the UoI_{VAR} implementation with Eigen3 Sparse C++ LASSO-ADMM.

Figure 7 shows the runtime analysis for UoI_{VAR}. Computation contributes 88% of the total runtime. Due to the problem size explosion, communication time for MPI_AllReduce can be seen to increase. The distributed Kronecker product and vectorization MPI calls are included in the distribution time constitutes more than 98% of the distribution time. UoI_{VAR} implementation was also analyzed with the Intel Advisor software for performance metrics. The performance of sparse matrix multiplication was 1.08 GFLOPS with 0.15 arithmetic intensity and the performance of matrix-vector multiplication was 2.08 GFLOPS/sec with 0.33 arithmetic intensity.

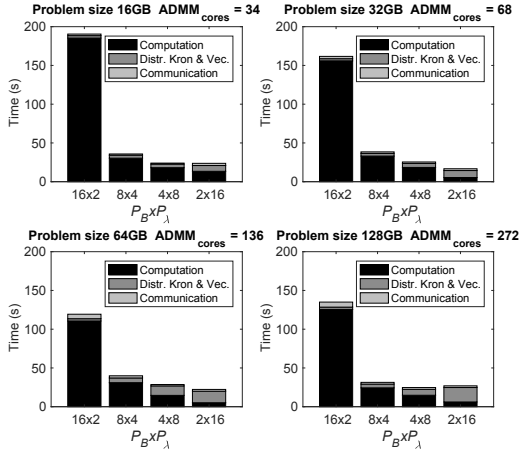

Fig. 8: Exploiting algorithmic parallelism of UoI_{VAR}.

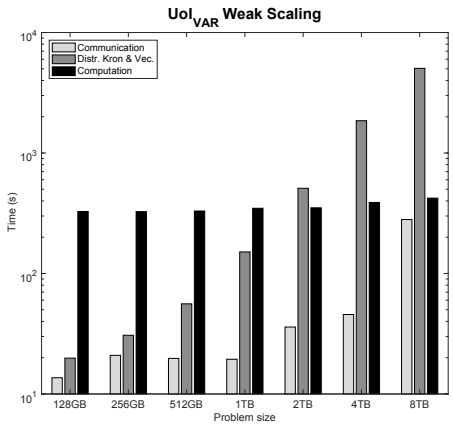

Fig. 9: Weak Scaling plot of UoI_{VAR} in logarithmic scale. The problem size per node was kept fixed

2) Exploiting Algorithmic Parallelism: The runs were carried out for problem set sizes of 16GB, 32GB, 64GB and 128GB. The number of ADMM$_{cores}$ were doubled with doubling the problem size. The runs were performed for $B_1 = B_2 = 32$ and $q = 16$. The computation dominates the execution time, which decreases with increases in parallelism of P_λ as shown in the Figure 8. It can also be noted that as the P_λ parallelism increases the Kronecker product and vectorization time increases. From Algorithm 2 (lines 5, 21 and 22) the distributed Kronecker product and vectorization is done for each bootstrap, and thus by reducing $\mathtt{P}_B$ parallelization increases the distribution time across different problem sets.

3) Multi-node Scaling: The data set size is very small for UoI_{VAR} compared to the problem size that is created during runtime. Unlike UoI_{LASSO} distribution strategy, only a few processes read the actual data set in parallel and the distributed Kronecker product routine builds the problem via MPI one-sided communication.

Weak Scaling: The weak scaling plot for UoI_{VAR} is shown in the Figure 9 for $B_1 = 30, B_2 = 20, q = 20$, with no P_B or P_λ parallelization. The Y-Axis in Figure 9 is given in a log-scale to show logarithmic increase in the distribution time. It can be seen that computation has almost ideal weak scaling, and the communication time also increases with increase in core count as seen in UoI_{LASSO}. The distributed Kronecker product and vectorization is proportional to the increase in the cores and problem size. One of the main reasons for this trend is the cubical increase of the problem size to the features of the input data set. Since only a few cores are responsible to read and distribute the data to thousands of computing cores during analysis, there is a communication bottleneck between the reader cores and the computing cores.

Strong Scaling: The strong scaling plot for UoI_{VAR} is shown in the Figure 10. Across increasing core sizes, computation time has an almost ideal strong scaling. The reason for an ideal computation time and as discussed earlier, Sparse Eigen C++ is used to compute the matrix-vector and matrix

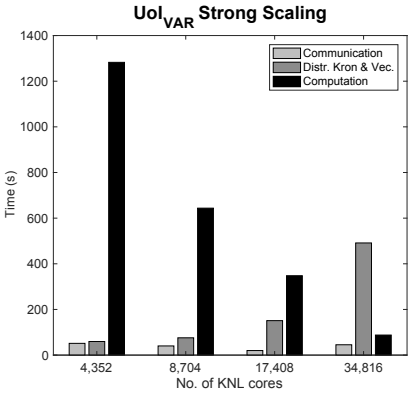

Fig. 10: Strong Scaling plot of UoI_{VAR}. The problem size was kept fixed at 1TB.

multiplication. Even though the communication does not have an ideal scaling it minimally affects the total runtime of the program. The distributed Kronecker product and vectorization scales exponentially to the increase in the number of cores like the weak scaling.

V. Discussion

We found a trade-off between computation and communication in UoI_{LASSO}, shown in Figure 4. When the data size per core increases the computation time increases because the computation bottlenecks are `BLAS gemm` and `gemv` operations. On the other hand for large data sets, the runtime of the code is determined by communication via MPI_Allreduce call, whereas the computation has a near ideal scaling. Almost 98% of the communication time seen in weak and strong scaling is from model selection module of the algorithm. This is due to the fact that the size of the problem solved in the model estimation module is greatly reduced relative to the model selection module. To reduce the communication runtime, P_B and P_λ parallelism can be adopted as shown in Figure 3 based on availability of resources.

In contrast to UoI_{LASSO}, we found a trade-off between computation and distribution in UoI_{VAR}, as shown in Figure 9. For smaller problem sizes computation dominates the program runtime and for larger problem sizes (especially for problem sizes 2TB and above) distribution dominates the total program runtime. The reason for this being the problem size explosion, where for a small input data size the distributed Kronecker product and vectorization creates a large matrix. One of the ways to avoid the problem is by utilizing P_B parallelism. Another way to alleviate this issue is by using communication avoiding algorithms and using local computation modules to create the matrix and then have a one-time communication to create the large matrix. With our UoI_{VAR} scaling analyses, we have implemented computations for the largest-scale VAR estimation problem (1,000 nodes, which corresponds to 1,000,000 parameters) we are aware of.

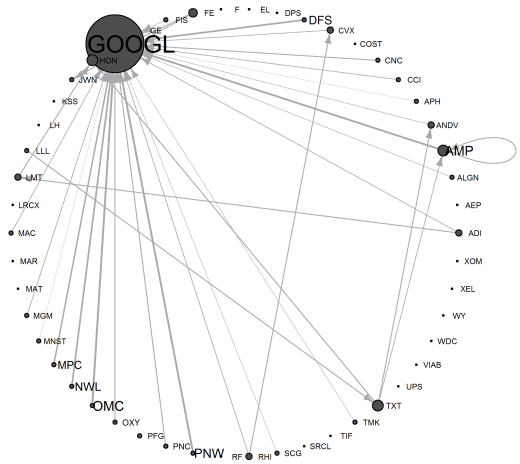

Fig. 11: Parameter estimates of $VAR(1)$ model for first differences of weekly closes of 50 randomly chosen companies on the S& P 500 Index during 2013 and 2014.

VI. Application of UoI_{VAR} to Real data sets

We use a financial time series data set to illustrate a data analysis using UoI_{VAR} and to illustrate computing runtime for UoI_{VAR} in a real application: the data are daily closes on the S& P Index for the years 2013 - 2018. Two different subsets are used to illustrate (i) Granger causality analysis using UoI_{VAR} and (ii) computing runtime with real data: a smaller subset was chosen for the Granger causality analysis to allow easier interpretation of the results; and a larger subset for the runtime analysis was chosen to represent compute times representative of larger-scale applications.

For the Granger causality analysis, we randomly chose 50 companies on the index in the years 2013 and 2014, aggregated the data to weekly closes, and took first differences to obtain a plausibly stationary vector time series. A first-order VAR model was then fit to the first differences using the UoI_{VAR} algorithm with hyperparameters $B_1 = 40, B_2 = 5$, selected to create a strong pressure toward sparse parameter estimates. The matrix of parameter estimates is represented in Figure 11 as a directed graph with nodes for each vector component (company), plotted with node sizes proportional to node degree and labeled according to company ticker, and with directed edges from node j to node i shown when the estimate of a_{ij} is nonzero, with line thickness proportional to estimate magnitude. The result is quite sparse, with fewer than 40 edges, and suggests a complex structure of dependence of Google on a variety of other companies spanning several industry sectors. Thus, the UoI_{VAR} algorithm produces a highly interpretable output.

For the runtime analysis, we retained all 470 companies that were on the index from January 2013 through December 2016, and performed the same aggregation and differencing as in the example analysis for 195 samples. The problem size for this data set is ≈80GB, and scaling it on 2,176 cores yielded a computation time of 376.87s, and a total communication time

of 4.74s. The Kronecker product and vectorization time was found to be 16.409s. In Figure 11 the nodes in graph are vector components and edges are nonzero parameter estimates. Our method identified very few edges thereby showing the effective dependence of Google's share price on other companies.

In addition to the financial data set, a single session non-human primate reaching task data set [22] was analyzed using UoI_{VAR} to illustrate computing runtime for a neuroscience application. Monkey reaching behavioral tasks were recorded in [22] with two monkey subjects. Some of the recorded data sets consist of spikes for both the motor cortex (M1) and, the somatosensory cortex (S1) recordings for 192 electrodes as features. The recorded spikes had 51111 samples recorded for one session. In the VAR model, the data set created a problem size $\approx$ 1.3TB. The problem was executed on 81,600 cores on Cori KNL. The computation and communication times were found to be 96.9s and 1598.72s, respectively. The distribution time recorded was 3034.4s.

VII. Conclusion

In this paper, we developed scalable implementations of UoI_{LASSO} and UoI_{VAR}. To achieve this, we created a randomized data distribution strategy for HDF5 to aid parallel bootstrap subsampling for UoI_{LASSO} and distributed Kronecker product and vectorization for UoI_{VAR}. The single-node performance evaluation and multi-node scaling evaluatios used a wide range of sizes of synthetic data sets. Our weak and strong scaling analyses show that UoI_{LASSO} is communication bound and UoI_{VAR} is distribution bound for large data sets and problem sizes, respectively. Finally, we have presented the data analysis of S&P Index and neuroscience real data set with UoI_{VAR}. Focusing on the S&P data, we illustrated the analysis with (i) a smaller subset for better visualization of the results and (ii) a larger subset closer to 80GB for runtime representation. The VAR-model has a sparse selection for the S&P data set, selecting fewer than 40 edges out of 2500 possible.

Acknowledgement

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This work was partially supported by funding from the National Science Foundation grants CCF 1723476 - the NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA). K.E.B. was funded by Lawrence Berkeley National Laboratory-internal LDRD "Neuro/Nano-Technology for BRAIN" led by Peter Denes. This research was sponsored by the U.S. Army Research Laboratory and Defense Advanced Research Projects Agency under Cooperative Agreement Number W911NF- 15-2-0056. The views, opinions, and/or findings contained in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] Laura Astolfi, Febo Cincotti, Donatella Mattia, M Grazia Marciani, Luiz A Baccala, Fabrizio de Vico Fallani, Serenella Salinari, Mauro Ursino, Melissa Zavaglia, Lei Ding, et al. Comparison of different cortical connectivity estimators for high-resolution eeg recordings. *Human brain mapping*, 28(2):143–157, 2007.

[2] Kendrick N Kay, Thomas Naselaris, Ryan J Prenger, and Jack L Gallant. Identifying natural images from human brain activity. *Nature*, 452(7185):352, 2008.

[3] Jonathan W Pillow, Jonathon Shlens, Liam Paninski, Alexander Sher, Alan M Litke, EJ Chichilnisky, and Eero P Simoncelli. Spatio-temporal correlations and visual signalling in a complete neuronal population. *Nature*, 454(7207):995, 2008.

[4] Kristofer E Bouchard, James B Aimone, Miyoung Chun, Thomas Dean, Michael Denker, Markus Diesmann, David D Donofrio, Loren M Frank, Narayanan Kasthuri, Chirstof Koch, et al. High-performance computing in neuroscience for data-driven discovery, integration, and dissemination. *Neuron*, 92(3):628–631, 2016.

[5] National Research Council et al. *Frontiers in massive data analysis*. National Academies Press, 2013.

[6] Kristofer E. Bouchard et al. International neuroscience initiatives through the lens of high-performance computing. *Computer*, 51(4):50–59, April 2018.

[7] Bijan Pesaran, Martin Vinck, Gaute T Einevoll, Anton Sirota, Pascal Fries, Markus Siegel, Wilson Truccolo, Charles E Schroeder, and Ramesh Srinivasan. Investigating large-scale brain dynamics using field potential recordings: analysis and interpretation. *Nat Neurosci*, 21(7):903–919, 2018.

[8] Clive WJ Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica: Journal of the Econometric Society*, pages 424–438, 1969.

[9] Helmut Lütkepohl. *New introduction to multiple time series analysis*. Springer Science & Business Media, 2005.

[10] Kristofer E. Bouchard et al. Union of Intersections (UoI) for Interpretable Data Driven Discovery and Prediction. In *Advances in Neural Information Processing Systems*, pages 1078–1086, 2017.

[11] Trevor Ruiz, Mahesh Balasubramanian, Kristofer E Bouchard, and Sharmodeep Bhattacharyya. Sparse, low-bias, and scalable estimation of high dimensional vector autoregressive models via union of intersections. *arXiv preprint arXiv:1908.11464*, 2019.

[12] Pinghua Gong and Jieping Ye. Honor: Hybrid optimization for non-convex regularized problems. In *Advances in Neural Information Processing Systems*, pages 415–423, 2015.

[13] Stephen Boyd et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[14] Sumanta Basu and George Michailidis. Regularized estimation in sparse high-dimensional time series models. *Ann. Statist.*, 43(4):1535–1567, 08 2015.

[15] Jianqing Fan, Jinchi Lv, and Lei Qi. Sparse high-dimensional models in economics. *Annu. Rev. Econ.*, 3(1):291–317, 2011.

[16] Mike Folk et al. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[17] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[18] Endong Wang et al. Intel Math Kernel Library. In *High-Performance Computing on the Intel Xeon Phi™—*, pages 167–188. Springer, 2014.

[19] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86. ACM, 2018.

[20] K O'Leary, I Gazizov, A Shinsel, R Belenov, Z Matveev, and D Petunin. Intel Advisor Roofline Analysis: A New Way to Visualize Performance Optimization Trade-offs. *Intel Software: The Parallel Universe*, 27:58–73, 2017.

[21] Mark Howison et al. Tuning HDF5 for lustre file systems. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2010.

[22] Joseph E. O'Doherty et al. Nonhuman Primate Reaching with Multichannel Sensorimotor Cortex Electrophysiology. https://doi.org/10.5281/zenodo.583331.