Parallel Construction of Module Networks

Ankit Srivastava
Georgia Institute of Technology
Atlanta, GA, USA
asrivast@gatech.edu

Sriram P. Chockalingam
Georgia Institute of Technology
Atlanta, GA, USA
srirampc@gatech.edu

Maneesha Aluru
Georgia Institute of Technology
Atlanta, GA, USA
maneesha.aluru@biology.gatech.edu

Srinivas Aluru
Georgia Institute of Technology
Atlanta, GA, USA
aluru@cc.gatech.edu

ABSTRACT

Module networks (MoNets) are a parameter-sharing specialization of Bayesian networks that are used for reasoning about multi-dimensional entities with concerted interactions between groups of variables. Construction of MoNets is compute-intensive, with sequential methods requiring months for learning networks with a few thousand variables. In this paper, we present the first scalable distributed-memory parallel solution for constructing MoNets by parallelizing *Lemon-Tree*, a widely used sequential software. We demonstrate the scalability of our parallel method on a key application of MoNets – the construction of genome-scale gene regulatory networks. Using 4096 cores, our parallel implementation constructs regulatory networks for 5,716 and 18,373 genes of two model organisms in 24 minutes and 4.2 hours, compared to an estimated 49 and 1561 days using *Lemon-Tree* for generating exactly the same networks, respectively. Our method is application-agnostic and broadly applicable to the learning of high-dimensional MoNets for any of its wide array of applications.

CCS CONCEPTS

• **Computing methodologies** → **Bayesian network models**.

KEYWORDS

Bayesian networks, module networks, score-based learning, parallel machine learning, gene networks

ACM Reference Format:
Ankit Srivastava, Sriram P. Chockalingam, Maneesha Aluru, and Srinivas Aluru. 2021. Parallel Construction of Module Networks. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476207

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476207

1 INTRODUCTION

Bayesian networks (BNs) use a directed acyclic graph (DAG) to represent the joint probability distribution of a set of random variables and thereby provide a compact model for reasoning about interactions in multi-dimensional entities. The capability of the BN framework to reason about uncertainty has led to their successful use in many different fields [9, 11, 62]. However, the deployment of BNs in intricate domains with a large number of variables has uncovered two major limitations – (a) it is difficult to interpret complex interactions between groups of variables that may lead to an emergent behavior of the entity from the BN models of such entities [30, 37], and (b) confidence in learned BN models is low when the data set does not have sufficient number of observations [48]. Specialization of BNs that rely on variations of *parameter-sharing* have been proposed to overcome these limitations [44].

Introduced by Segal *et al.*, module networks (MoNets) [51, 52] are among the most commonly used parameter-sharing specializations of BNs. A learned MoNet can identify groups of variables (or *modules*) that operate in a concerted fashion, possibly driven by other groups of variables. The primary advantage of MoNets over other parameter-sharing BN specializations – such as object-oriented BNs [31], probabilistic relational models [30], hierarchical BNs [19], etc. – is that, unlike these variations, MoNets can be learned in an unsupervised manner, i.e., without requiring any prior knowledge of relationships between variables. Due to their unsupervised nature, MoNets have been utilized in a wide range of applications in computational biology, e.g., gene regulatory studies [53], cancer genomics [34, 49, 50, 54], construction of cellular networks [7, 39, 47], and integration of multi-omics data [1, 12, 13]. MoNets and other parameter-sharing specializations of BNs have also found applications in diverse fields, e.g., medical diagnosis [32], stock market analysis [52], traffic modeling [28], active learning using serious games [57], feature selection and feature extraction [23], computational psychology [17, 18], and data mining [36].

Learning a MoNet from data requires learning of a module assignment function that maps each variable to a module, in addition to learning the parent-child relationships between variables in the form of a DAG. Therefore, the MoNet learning problem is at least as hard as the problem of exactly learning BN structure, which is NP-hard [15]. Consequently, approaches to construct MoNets resort to heuristic methods. However, even using these heuristic methods, learning MoNets from data sets with thousands of variables and observations can take months sequentially.

1.1 Related Work

Various *score-based* heuristic methods have been proposed for constructing MoNets from observed data [5, 13, 25, 40, 52]. The score of a MoNet is a Bayesian metric that evaluates the fitness of both the partition of variables into modules and the structure of the underlying network, given observed data. Similar to the *score-based* approaches used for BN structure learning [20], heuristics are used in MoNet learning to traverse the space of all possible MoNets and obtain a network with locally optimal score in the expectation that it approximates the globally optimal network reasonably well. In contrast to BN learning methods, though, the MoNet learning methods also need to learn the conditional probability distributions (CPDs) for the modules as part of their structure learning routine. The most popular software packages for MoNet learning are *GENOMICA* [52] and *Lemon-Tree* [13]. In both these software, the learned CPDs are represented using regression trees [14].

GENOMICA implements the iterative two-step algorithm proposed by Segal *et al.* [51, 52] to construct MoNets. *Lemon-Tree*, on the other hand, implements the approach outlined by Bonnet *et al.* [13], that refined an earlier approach by Michoel *et al.* [40]. This approach separates the learning of module assignments and parents and CPDs into three distinct tasks which are described in detail in Section 2.2. Previous studies that evaluated the two approaches found *Lemon-Tree* to be more effective at constructing robust MoNets from both synthetic as well as real-world data sets [25, 35, 40]. Further, *Lemon-Tree* software has been successfully used in multiple recent works with potential for far-reaching impact. These include studies intending to increase life expectancy by understanding complex diseases such as glioblastoma [13], cholangiocarcinoma [43], breast cancer [34], penile cancer [38], and rheumatoid arthritis [35]. The software has also been used in works aiming to enhance quality of life by improving food production processes through studies on stress-related immune response [10] and feed efficiency [2] of cattle, analysis of early stage development of European sea bass [27], and identification of genes critical for tomato ripening [4] and apple edibility [6].

However, *Lemon-Tree* is computationally expensive, which has limited its use for genome-wide gene regulatory network studies to smaller micro-organisms. For organisms with tens of thousands of genes, MoNet construction is possible only for a subset of genes that are involved in specific pathways of interest [4, 60]. Even for the single-celled *Saccharomyces cerevisiae*, with 5,716 genes, we estimate that sequentially constructing a whole-genome network using *Lemon-Tree* will take 49 days.

To mitigate the run-time issues in constructing MoNets, the approach proposed by Segal *et al.* has been parallelized by multiple groups. Liu *et al.* [33] parallelized the MoNet learning method using a distributed-memory approach. They report a speedup of up to 29.3X using a maximum of 32 cores. Jiang *et al.* [24] developed a shared-memory parallel solution and report a maximum speedup of 3.5X using 4 threads. In addition to limited scaling, both these parallelization strategies are specific to the approach by Segal *et al.*, i.e., *GENOMICA*, and are not applicable for parallelizing *Lemon-Tree*.

1.2 Contributions

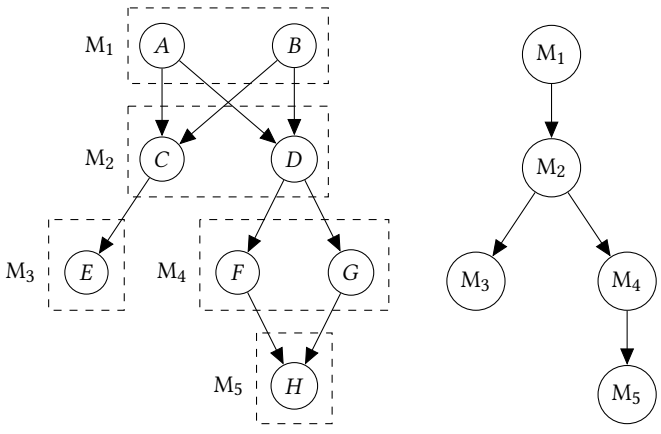
In this paper, we introduce a parallel, distributed-memory based approach for constructing large MoNets efficiently. We limit our focus to parallelizing *Lemon-Tree*, which is more widely used for this purpose. We present distributed-memory parallel algorithms for the tasks used in *Lemon-Tree*, both for learning the modules and the CPD regression trees. To demonstrate that our implementation of the parallel algorithms can scale to constructing networks for tens of thousands of variables from thousands of observations, we construct genome-scale gene networks for *S. cerevisiae* and *Arabidopsis thaliana* with 5,716 and 18,373 genes, respectively. Our parallel implementation can construct a MoNet for *S. cerevisiae* in 25 minutes and for *A. thaliana* in 4.2 hours using 4096 cores as compared to an estimated 13.5 and 433.6 days, respectively, with our optimized *C++* sequential implementation. The corresponding run-time estimates when using *Lemon-Tree* are 48.6 and 1561 days for generating exactly the same network. Our method is application-agnostic and is broadly applicable to the learning of high-dimensional MoNets for any of its wide array of applications.

2 BACKGROUND

2.1 Module networks

For a set of n random variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, the corresponding BN is a compact representation of the joint probability distribution of the variables via a DAG such that the distribution decomposes as $P(\mathcal{X}) = \prod_i P(X_i|\mathsf{Pa}(X_i))$, where $\mathsf{Pa}(X_i)$ is the set of parents of X_i in the DAG. We denote the children of a variable X_i in the DAG by $\mathsf{Ch}(X_i)$.

MoNets are BNs with the variables partitioned into modules, where a module consists of a set of variables that share the same set of parents and the same CPD. We use K to denote the maximum number of modules in the MoNet and represent each module by a module variable (M_1, M_2, etc.) and the set of all the modules as $\mathcal{M} = \{\mathrm{M}_1, \ldots, \mathrm{M}_K\}$. A module assignment function, denoted



(a) Bayesian network **(b) Module network**

Figure 1: An example BN for a set of eight random variables $\{A, B, C, D, E, F, G, H\}$ and the corresponding MoNet.

by $\mathcal{A}$, assigns each variable in $\mathcal{X}$ to one of the modules in $\mathcal{M}$, e.g., $\mathcal{A}(X_i) = \mathrm{M}_j$ implies that the variable X_i is an element of the module M_j. Each module has a set of parent variables, represented by $\mathrm{Pa}(\mathrm{M}_j)$, where $\mathrm{Pa}(\mathrm{M}_j) \subset \mathcal{X}$. Given these definitions, a MoNet is a DAG that has:

- a vertex for every module variable in $\mathcal{M}$, and
- a directed edge $\mathrm{M}_j \rightarrow \mathrm{M}_k$ if and only if there exists a variable $X \in \mathcal{X}$ such that $\mathcal{A}(X) = \mathrm{M}_j$ and $X \in \mathrm{Pa}(\mathrm{M}_k)$.

Figure 1a shows an example BN with a potential assignment of variables to modules shown by dashed rectangles in which $\mathcal{A}(A) = \mathrm{M}_1$, $\mathcal{A}(C) = \mathrm{M}_2$, etc. The MoNet structure corresponding to this assignment of variables to modules is shown in Figure 1b. Note that, the parents of a module should be a parent for all the variables in the module, e.g., $B \in \mathrm{Pa}(\mathrm{M}_2) \iff B \in \mathrm{Pa}(C)$ and $B \in \mathrm{Pa}(D)$ in the figure. However, the variables in a module may have different sets of descendants, e.g., $\mathrm{Ch}(C) \neq \mathrm{Ch}(D)$.

MoNets are learned from multiple (m) observations of the n random variables, represented as an $n \times m$ matrix of either discrete or continuous values. As the focus in this work is parallelization of *Lemon-Tree*, learning of MoNets using the corresponding algorithm is described in greater detail next.

2.2 The *Lemon-Tree* Algorithm

Lemon-Tree implements the MoNet learning method proposed by Bonnet *et al.* [13]. This MoNet learning method consists of three main tasks that are executed in the order they are described below.

2.2.1 GaneSH Co-Clustering. The first task constructs an ensemble of variable clusters using a Gibbs sampler algorithm called *GaneSH*, proposed by Joshi *et al.* [26]. The algorithm performs two-way clustering of variables and observations to get a variable-observation co-clustering. *GaneSH* scores a co-clustering using a decomposable Bayesian scoring function (described in [26]) that can be computed by aggregating the independently computed scores for all the variable and observation clusters. The algorithm explores the space of co-clustering solutions as follows:

(1) *Random Initialization*: The n variables are randomly assigned to a user-provided number of variable clusters, or $n/2$ clusters if no input is provided. In each variable cluster, the m observations are randomly assigned to $\sqrt{m}$ observation clusters.

(2) *Update Steps*: The randomly initialized co-clustering is updated multiple times, as per user input. In each update step of the algorithm, the clustering of variables and observations is updated as follows:

- *Variable Clustering*: For n iterations, the cluster assignment of a randomly selected variable is evaluated while keeping the assignment of all the other variables and observations fixed. The chosen variable is then randomly assigned to one of the existing clusters or moved to its own separate cluster. The probability of each choice for this random reassignment is proportional to the corresponding change in the score. After n reassignment iterations, each variable cluster is considered one at a time and is merged with one of the other clusters or left as is, chosen at random with the probability of each possible action weighted by the corresponding score.
- *Observation Clustering*: The variable cluster assignments are fixed and for each variable cluster, updates to observation clustering proceed similar to the variable clustering iteration. First, the cluster assignment of m randomly selected observations is changed, one at a time, similar to the random reassignment of variables described above. Then, the merging of observation clusters proceeds similar to the merging of variable clusters.

The co-clustering algorithm simulates a Markov chain, i.e., the probability to visit a particular co-clustering corresponds exactly to its posterior probability given the data. In order to get the variable clusters corresponding to high posterior probability, the algorithm is run multiple times with different random initializations and variable clusters are sampled at the end of each run.

Let K be the maximum number of variable clusters and L be the maximum number of observation clusters in any variable cluster. Then, variable and observation clustering phases in each sampling step require $O(nKLm + K^2Lmn)$ and $O(K(mLn + L^2))$ time, respectively, for an asymptotic complexity of $O(K^2Lnm)$ per update step. Therefore, in order to sample variable clusterings from G runs of *GaneSH* with U update steps, the total time required is $O(GUK^2Lnm)$.

2.2.2 Consensus Clustering. In the second task, a single consensus variable clustering solution is constructed from the ensemble of variable clusters sampled in the first task. This is done by creating a symmetric co-occurrence frequency matrix A of size $n \times n$. The entry $A(i, j)$ of the matrix is set to the number of times the variables X_i and X_j occur in the same cluster in the ensemble, as a fraction of the total number of sampled clusters. Note that $A(i, j)$ is set to zero if the co-occurrence weight is below a user-provided threshold. The matrix A is then provided as an input to the spectral clustering algorithm proposed by Michoel and Nachtergaele [41] to obtain the consensus variable clusters. The time complexity of the complete consensus clustering step is $O(Gn^2)$, where G is the number of variable cluster samples from the first task.

2.2.3 Learning the Modules. The consensus variable clusters identified by the second task are defined as the modules ($\mathcal{M}$) of the MoNet and are provided as an input to the third task. In this task, the parent variables and the corresponding CPDs are learned for each module by first learning regression tree structures followed by the assignment of the parent variables and split values, or parent splits, to the nodes of the regression trees. The parent variables are chosen from a list of candidate parent variables for all the modules that can be provided as an input to this step. If no candidate list is provided, then every variable is considered a candidate parent. For each module $\mathrm{M}_i \in \mathcal{M}$, the third task proceeds through the following three main steps:

(1) *Learning Regression Tree Structures*: For the module M_i, an ensemble of regression trees (denoted by $\mathcal{T}(\mathrm{M}_i)$) are learned as follows. First, the leaf nodes of the trees are built by learning multiple different clusterings of observations. This is accomplished by executing the *GaneSH* algorithm (described in Section 2.2.1) while constraining the variable clusters to a single cluster containing the variables assigned to the module M_i, and sampling an ensemble of likely observation clustering solutions for M_i. Then, a binary regression tree structure is constructed by initializing the leaf nodes

with the observation clusters and merging them using Bayesian hierarchical agglomerative clustering [21, 40], until all the nodes are merged into one root node with all the observations.

If R sets of observation clusters are sampled in this step, then *GaneSH* algorithm takes $O(R(mLn+L^2))$ time, where L is the maximum number of observation clusters. Then, the hierarchical clustering for getting each regression tree structure requires $O(Lnm+L^2)$ time. Therefore, this step requires a total of $O(R(Lnm+L^2))$ time that is bounded by $O(RLnm)$, since $L = O(m)$.

(2) *Node Parent Split Assignments*: In this step, for all the regression tree structures learned for M_i, i.e., all the trees in $\mathcal{T}(\mathrm{M}_i)$, the assignment of parent splits to every internal node is accomplished as follows:

(i) *Scoring Candidate Splits*: Given the set of candidate parents $\mathcal{P}$, all the $\langle X_i, \mathrm{D}_{ij}\rangle$ pairs are considered as candidate parent splits for the given internal node, where X_i is a candidate parent and D_{ij} is a value of X_i in D corresponding to the observations at the node. The maximum posterior probability of assigning every such candidate parent split to the node is computed by sampling from a discrete distribution, as described in [25], and the candidate splits with zero posterior probability are discarded. Since all the n variables may be candidate parents in this stage, the number of splits at every node is bounded by $O(nm)$. If S is the maximum number of discrete sampling steps for any split, then computing the posterior probability for a split requires $O(Sm)$ time for a total time of $O(Snm^2)$ for this stage.

(ii) *Assigning Parent Splits*: In this stage, a user supplied number of splits are chosen from all the candidate splits retained in the previous stage, using weighted random sampling with the corresponding posterior probabilities as weights. Additionally, the same number of splits are selected using uniform random sampling. Both these sets of selected splits are assigned to the internal node. This stage performs a linear scan through the list of candidate splits, for weighted sampling, in $O(nm)$ time.

The total number of non-leaf nodes in every binary regression tree is bounded by $O(L)$ as the total number of leaf nodes is bounded by $O(L)$. Therefore, the assignment of splits to all the nodes of the R regression trees of M_i requires a total of $O(RLSnm^2)$ time.

(3) *Learning Module Parents*: For a module M_i, the parents of the module include all the variables corresponding to all the splits assigned to all the nodes of all the regression trees learned for M_i. The score for a parent variable X_i is computed as the average of the posterior probabilities for the splits containing X_i, weighted by the number of observations at the node that the splits are assigned to. Further, the scores of the parents from splits chosen uniformly at random for every node are also computed. The computed scores for both the sets of parents, chosen using weighted sampling as well as uniform random sampling, are used for further downstream analysis, e.g., to assess the significance of the parent variables [13, 25]. If J splits are chosen in the previous step, the parent weights for every module can be learned in $O(JRL)$ time.

The time complexity of the third task for one module is $O(RLnm + RLSnm^2 + JRL)$, where J is bounded by the total number of possible splits $O(nm)$ and $R = O(U)$. Therefore, the run-time of this task for K modules is $O(UKLSnm^2)$. The total time complexity of the three tasks of *Lemon-Tree* is

$$O(GUK^2Lnm + Gn^2 + UKLSnm^2) \quad (1)$$

where G is the number of *GaneSH* runs, U is the number of update steps in each *GaneSH* run, $K = O(n)$ is the maximum number of variable clusters, $L = O(m)$ is the maximum number of observation clusters, and S is the maximum number of sampling steps for computing the split probabilities. Since G, U, K, and L are much smaller than n and m for large data sets, the time taken by the last task dominates the total run-time of *Lemon-Tree* as observed in the experiments reported in Section 5.

Note that, a network learned using the *Lemon-Tree* approach may not satisfy the formal definition of MoNets because of the following two reasons. First, multiple regression trees for every module are learned when $R > 1$. This can be easily addressed by changing the corresponding input parameter to sample only one observation cluster in the third task. Second, the algorithm does not enforce the acyclicity constraint. Therefore, the MoNets learned by the algorithm may need to be post-processed using an existing method to get the DAG for the learned network.

3 OUR PARALLEL ALGORITHM

We design our parallel algorithm for learning MoNets to ensure consistency of results with the sequential *Lemon-Tree* implementation for all data sets. Since the sequential version of *Lemon-Tree* has been proven to be successful in many applications, this ensures ready adoption of our parallel software, while providing the needed scalability.

3.1 Assumptions

We develop the proposed parallel algorithms for execution using p processors assuming the networked distributed memory model. The processors in the model have their own local memory and communicate with the other processors using a communication network. While a processor in the model can only communicate with one other processor at a time, the network is assumed to be capable of supporting communication between multiple distinct pairs of processors concurrently. The communication time of the algorithms designed for this model is estimated by assuming τ time to setup communication and μ time per word to send a message between any two processors. Note that the model is consistent with the widely used *Message Passing Interface* (*MPI*) programming standard. We assume that the complete data set D is available on all the processors.

Random sampling is required in the different tasks of the *Lemon-Tree* algorithm. In our description of the parallel algorithm, we assume the availability of two oracle functions that facilitate uniform and weighted random sampling in parallel. Select-Unif-Rand() accepts as input a distributed list $\mathcal{B}$, and returns an element $b \in \mathcal{B}$ chosen at random with a probability $1/|\mathcal{B}|$. Select-Wtd-Rand() accepts two inputs – a distributed list $\mathcal{B}$ and a corresponding list of real numbers, W, with the weights of all the elements in $\mathcal{B}$. It chooses an element $b \in |\mathcal{B}|$ with the probability $W(b)/\sum_{x\in\mathcal{B}} W(x)$, where $W(x)$ is the weight corresponding to the element x. When sampling using p processors, Select-Unif-Rand() requires $O(1)$ computation time and $O((\tau+\mu)\log p)$ time for communicating the

chosen element to all the processors, while Select-Wtd-Rand() requires $O(|\mathcal{B}|/p+\log p)$ computation and $O((\tau+\mu)\log p)$ communication time, in order to compute the probability of picking each element from W. Notice that the calls to these sampling functions are collective communication calls, i.e., all the processors participate in the sampling calls. We discuss the implementation of distributed random sampling in Section 4.2.

3.2 Parallelizing *Lemon-Tree*

The sequential *Lemon-Tree* algorithm executes three different tasks for the construction of MoNets. In this section, we parallelize *Lemon-Tree* by developing parallel algorithms for the different tasks. We present pseudo-codes for the proposed algorithms from the perspective of an arbitrary processor with rank k ($0 \leq k < p$). The data structures local to the processor are identified by the subscript k. We use standard parallel primitives such as *bcast*, *all-reduce*, *all-gather*, and *scan*, in the design of these algorithms.

3.2.1 GaneSH Co-Clustering. The sequential *GaneSH* task samples an ensemble of variable clusters by performing variable-observation co-clustering as described in Section 2.2.1. We denote a cluster of variables by $\mathcal{V}$ and the cluster of the observations for the variable cluster $V_i \in \mathcal{V}$ by $\mathcal{O}(V_i)$. We also denote the j-th observation in the data set D as D_j.

Algorithm 1: Parallel Update of Variable Clusters

1 **function** Reassign-Var-Cluster():
  Input: Variables $\mathcal{X}$
  Input/Output: Set of variable clusters $\mathcal{V}$
2 **parallel** k = *rank of processor* **do**
3 **for** $i \leftarrow 1$ **to** $|\mathcal{X}|$ **do**
4 $r \leftarrow$ Select-Unif-Rand($\{1, \ldots, |\mathcal{X}|\}$)
5 $V_r \leftarrow$ Cluster assignment of X_r in $\mathcal{V}$
6 $\mathcal{V}_k \leftarrow k^{\text{th}}$ block of $\mathcal{V} \cup \{\text{empty cluster}\}$ partitioned into p blocks
7 **for** $V_j \in \mathcal{V}_k$ **do**
8 $vu\text{-}scores_k(V_j) \leftarrow$ Score for moving X_r to V_j if $V_j \neq V_r$, else for keeping X_r in V_r
9 $V_s \leftarrow$ Select-Wtd-Rand($\mathcal{V}$, $vu\text{-}scores_k$)
10 **if** $V_r \neq V_s$ **then**
11 Move X_r to V_s and update $\mathcal{V}$
12 **function** Merge-Var-Cluster():
  Input/Output: Set of variable clusters $\mathcal{V}$
13 **parallel** k = *rank of processor* **do**
14 **for** $V_i \in \mathcal{V}$ **do**
15 $\mathcal{V}_k \leftarrow k^{\text{th}}$ block of $\mathcal{V}$ partitioned into p blocks
16 **for** $V_j \in \mathcal{V}_k$ **do**
17 $vm\text{-}scores_k(V_k) \leftarrow$ Score for merging V_i with V_j if $V_i \neq V_j$, else for retaining V_i
18 $V_s \leftarrow$ Select-Wtd-Rand($\mathcal{V}$, $vm\text{-}scores_k$)
19 **if** $V_i \neq V_s$ **then**
20 Merge V_i and V_s and update $\mathcal{V}$

We parallelize this task by developing parallel algorithms for the four key functions used by *GaneSH*. The first two functions are used in the variable clustering phase, and therefore modify only the variable clusters $\mathcal{V}$ while keeping $\mathcal{O}$ the same. Algorithm 1 describes the pseudo-code for our parallel algorithm for these functions. For n iterations, Reassign-Var-Cluster() selects a variable X_r and computes the change in score for moving X_r from its current assignment to every other variable cluster. It randomly selects a cluster V_s with probability in proportion to the reassignment scores and reassigns X_r to V_s (lines 3 – 11) . Merge-Var-Cluster() evaluates, for each variable cluster V_i, the score changes for merging it with every other variable cluster. Then, it merges V_i with a randomly chosen cluster with probability proportional to the merge scores (lines 14 – 20). The computation of scores is done in parallel in both the functions. Therefore, using p processors, the variable clustering phase requires a total of $O(K^2Lnm/p + n\log p)$ computation time and $O(n(\tau+\mu)\log p)$ communication time.

The other two functions are used in the observation clustering phase to update the observation clusters $\mathcal{O}$ while keeping $\mathcal{V}$ the same. Our proposed parallel algorithms for these two functions are shown in Algorithm 2. Similar to the functions for updating variable clusters, the pseudo-code for reassigning data instances from one

Algorithm 2: Parallel Update of Observation Clusters

1 **function** Reassign-Obs-Cluster():
  Input: Number of observations m, Data set D
  Input/Output: Set of observation clusters $\mathcal{O}(V_i)$
2 **parallel** k = *rank of processor* **do**
3 **for** $i \leftarrow 1$ **to** m **do**
4 $r \leftarrow$ Select-Unif-Rand($\{1, \ldots, m\}$)
5 $O_r \leftarrow$ Cluster assignment of D_r in $\mathcal{O}(V_i)$
6 $\mathcal{O}_k \leftarrow k^{\text{th}}$ block of $\mathcal{O}(V_i) \cup \{\text{empty cluster}\}$ partitioned into p blocks
7 **for** $O_j \in \mathcal{O}_k$ **do**
8 $ou\text{-}scores_k(O_j) \leftarrow$ Score for moving D_r to O_j if $O_j \neq O_r$, else for keeping D_r in O_r
9 $O_s \leftarrow$ Select-Wtd-Rand($\mathcal{O}(V_i)$, $ou\text{-}scores_k$)
10 **if** $O_r \neq O_s$ **then**
11 Move D_r to O_s and update $\mathcal{O}(V_i)$
12 **function** Merge-Obs-Cluster():
  Input/Output: Set of observation clusters $\mathcal{O}(V_i)$
13 **parallel** k = *rank of processor* **do**
14 **for** $O_i \in \mathcal{O}(V_i)$ **do**
15 $\mathcal{O}_k \leftarrow k^{\text{th}}$ block of $\mathcal{O}(V_i)$ partitioned into p blocks
16 **for** $O_j \in \mathcal{O}_k$ **do**
17 $om\text{-}scores_k(O_j) \leftarrow$ Score for merging O_i with O_j if $O_i \neq O_j$, else for retaining O_i
18 $O_s \leftarrow$ Select-Wtd-Rand($\mathcal{O}(V_i)$, $om\text{-}scores_k$)
19 **if** $O_i \neq O_s$ **then**
20 Merge O_i and O_s and update $\mathcal{O}(V_i)$

Algorithm 3: Parallel *GaneSH* Co-Clustering

1 **function** GaneSH():
  Input: $\mathcal{X}$, m, D, Initial number of variable clusters K_0, Number of update steps U
  Output: $\mathcal{V}$, $\mathcal{O}(V_i)$ $\forall V_i \in \mathcal{V}$
2 **parallel** k = *rank of processor* **do**
3 $\mathcal{V} \leftarrow$ Randomly assign each variable $X_i \in \mathcal{X}$ to K_0 variable clusters
4 **for** $V_i \in \mathcal{V}$ **do**
5 $\mathcal{O}(V_i) \leftarrow$ Randomly assign observations D_j $\forall j \in \{1, \ldots, m\}$ to $\sqrt{m}$ observation clusters
6 **for** $u \leftarrow 1$ **to** U **do** // Update Steps
7 Reassign-Var-Cluster($\mathcal{X}$, $\mathcal{V}$)
8 Merge-Var-Cluster($\mathcal{V}$)
9 **for** $V_i \in \mathcal{V}$ **do**
10 Reassign-Obs-Cluster(m, D, $\mathcal{O}(V_i)$)
11 Merge-Obs-Cluster($\mathcal{O}(V_i)$)

observation cluster to another is shown in Reassign-Obs-Cluster() function and that for merging observation clusters is shown in Merge-Obs-Cluster() function. These functions proceed similar to the two functions for variable clustering described earlier and they require a total computation run-time of $O(KLnm/p+Km\log p)$ and communication run-time of $O(Km(\tau+\mu)\log p)$ when running on p processors.

Algorithm 3 shows our parallel algorithm for the *GaneSH* task. The algorithm starts by randomly initializing a set of variable clusters $\mathcal{V}$ and, for each variable cluster $V_i \in \mathcal{V}$, a set of observation clusters $\mathcal{O}(V_i)$ (lines 3 – 5). Then, the algorithm proceeds to the main loop of the update steps (lines 6 – 11). In each update step, the parallel functions defined in Algorithm 1 update the variable clusters (lines 7 – 8) and those defined in Algorithm 2 update the observation clusters (lines 9 – 11). The number of updates is controlled by the input parameter U. Adding the parallel run-time complexity of the constituent functions and simplifying, one run of GaneSH() takes $O(UK^2Lnm/p + U(n+Km)\log p)$ computation run-time and $O(U(n+Km)(\tau+\mu)\log p)$ communication run-time. Notice that, G runs of *GaneSH* can be executed in parallel on p/G processors each, without any communication, to obtain G samples of $\mathcal{V}$.

3.2.2 Consensus Clustering. The consensus clustering task takes the G samples of $\mathcal{V}$ generated by Algorithm 3 as input and outputs the consensus variable clusters. In our experiments, described in Section 5, executing the consensus clustering task requires less than 0.04% of the total sequential run-time in all the cases. Even for a data set with 5,716 variables and 1,000 observations – the largest data set that we used for learning the networks sequentially – consensus clustering takes less than one second, while the other two tasks take more than two days. Therefore, we do not focus on developing a parallel algorithm for the consensus clustering task. Instead, we execute the sequential version of this task, using Consensus-Clustering() implemented as described in Section 2.2.2, on all p processors in our parallel solution.

3.2.3 Learning the Modules. Given the set of consensus variable clusters that are used as modules ($\mathcal{M}$), the final task of *Lemon-Tree* constructs an ensemble of regression tree structures for each module and then assigns parent splits to the nodes of the regression trees. Algorithm 4 shows the pseudo-code for the construction of an ensemble of regression tree structures for a module $\mathrm{M}_i \in \mathcal{M}$. The first part of the algorithm uses *GaneSH* to sample an ensemble of observation clusters for the variable cluster corresponding to M_i and stores them in $\mathcal{S}(\mathrm{M}_i)$ (lines 3 – 9). Unlike the *GaneSH* run described in the section 3.2.1, the variable clusters are not updated. Therefore, only the parallel *GaneSH* functions for observation clustering, presented in Algorithm 2, are used here. Correspondingly, getting $\mathcal{S}(\mathrm{M}_i)$ in parallel takes $O(U(KLnm/p + Km\log p))$ time for computation and $O(U(Km(\tau+\mu)\log p)))$ for communication. The second part of the algorithm constructs the ensemble of regression tree structures by hierarchical clustering for each observation clustering $\mathcal{Q} \in \mathcal{S}(\mathrm{M}_i)$ (lines 10 – 18). For R observation clusters in $\mathcal{S}(\mathrm{M}_i)$, this part takes $O(RLnm/p + RL\log p)$ time in computation and $O(RL(\tau+\mu)\log p)$ time in communication. Since $R = O(U)$, the time complexity of getting regression tree structures in parallel is dominated by that of the first part.

The next phase of this task is the assignment of parent splits to the non-leaf nodes of the ensemble of trees. This is the most time consuming of all the phases in *Lemon-Tree*, accounting for more than 90% of the sequential run-times in our experiments. It requires

Algorithm 4: Parallel Learning of Tree Structures

1 **function** Learn-Tree-Struct():
  Input: m, D, Module M_i, Number of update steps U, Number of burn-in steps B
  Output: Ensemble of trees for M_i – $\mathcal{T}(\mathrm{M}_i)$
2 **parallel** k = *rank of processor* **do**
3 $\mathcal{O}(\mathrm{M}_i) \leftarrow$ Randomly assign observations D_j $\forall j \in \{1, \ldots, m\}$ to $\sqrt{m}$ observation clusters
4 $\mathcal{S}(\mathrm{M}_i) \leftarrow \emptyset$ // Sampled Observation Clusters
5 **for** $u \leftarrow 1$ **to** U **do** // GaneSH Loop
6 Reassign-Obs-Cluster(m, D, $\mathcal{O}(\mathrm{M}_i)$)
7 Merge-Obs-Cluster($\mathcal{O}(\mathrm{M}_i)$)
8 **if** $u > B$ **then**
9 Add the current $\mathcal{O}(\mathrm{M}_i)$ to $\mathcal{S}(\mathrm{M}_i)$
10 **for** $\mathcal{Q} \in \mathcal{S}(\mathrm{M}_i)$ **do** // Build Tree Ensemble
11 $\mathcal{Q}_k \leftarrow k^{\text{th}}$ block of $\mathcal{Q}$ partitioned into p blocks
12 $subtrees_k \leftarrow$ Trees with a node for all $\mathrm{Q}_i \in \mathcal{Q}_k$
13 **repeat**
14 $tm\text{-}scores_k \leftarrow$ Scores for merging consecutive trees in $subtrees_k$
15 $max\text{-}tms \leftarrow$ *all-reduce* $\max_{0 \le k < p}$ $tm\text{-}scores_k$
16 Merge the trees corresponding to $max\text{-}tms$
17 **until** $\sum_{0 \le k < p} |subtrees_k| = 1$
18 *bcast* the remaining tree in $subtrees_k$ to all the processors and add it to $\mathcal{T}(M_i)$

Algorithm 5: Parallel Assignment of Splits to Tree Nodes

1 **function** LEARN-TREE-SPLITS():
  Input: D, Modules $\mathcal{M}$, Ensemble of trees $\mathcal{T}$, Candidate parents $\mathcal{P}$, Number of splits to choose J
  Output: Weighted splits *wr-splits*,
2 Random splits *ur-splits*
3 **parallel** k = *rank of processor* **do**
4 *cand-splits* ← List of tuples $\langle \mathrm{M}_i, T, N, X_i, \mathrm{D}_j \rangle$ for all $\mathrm{M}_i \in \mathcal{M}$, $T \in \mathcal{T}(\mathrm{M}_i)$, $N \in$ *internal-nodes*(T), $X_i \in \mathcal{P}$, $\mathrm{D}_j \in$ *observations*(N)
5 *cand-splits*$_k$ ← k^{th} block of *cand-splits* partitioned into p blocks
6 **for** $\langle \mathrm{M}_i, T, N, X_i, \mathrm{D}_j \rangle \in$ *cand-splits*$_k$ **do**
7 *cand-probs*$_k[\langle \mathrm{M}_i, T, N, X_i, \mathrm{D}_j \rangle]$ ← Posterior probability of assigning the split $\langle X_i, \mathrm{D}_{ij} \rangle$ to node N of regression tree T for module M_i
8 **for** $\mathrm{M}_i \in \mathcal{M}, T \in \mathcal{T}(\mathrm{M}_i), N \in$ *internal-nodes*(T) **do**
9 *tnode-splits*$_k$ ← Elements of *cand-splits*$_k$ in which the first three elements are $\langle \mathrm{M}_i, T, N \rangle$
10 *tnode-probs*$_k$ ← Computed probabilities for the elements of *tnode-splits*$_k$ from *cand-probs*$_k$
11 **for** $s \leftarrow 1$ **to** J **do**
12 *wr-splits*$[\langle \mathrm{M}_i, T, N, s \rangle]$ ← SELECT-WTD-RAND(*tnode-splits*$_k$, *tnode-probs*$_k$)
13 *ur-splits*$[\langle \mathrm{M}_i, T, N, s \rangle]$ ← SELECT-UNIF-RAND(*tnode-splits*$_k$)

the computation of posterior probabilities for every combination of the following five components: module M_i, tree T in the ensemble $\mathcal{T}(\mathrm{M}_i)$, non-leaf node N in the tree T, variable X_i in the list of candidate parents $\mathcal{P}$, and observation D_j at node N. Algorithm 5 depicts our parallel solution for this phase.

A simple parallelization scheme for this phase may assign all the probability computations for a module, a tree, or a node to one processor in order to reduce communication between the processors. However, such a scheme is sub-optimal because the total number of splits assigned to different processors will vary significantly, thus leading to severe load imbalance. Therefore, to enable a more fine-grained distribution of the computations across processors, we first identify the total work required in this phase using a key data structure – the list of all the candidate splits (line 4). All the tuples corresponding to the candidate splits for a particular node, i.e., tuples with the same first three elements $\langle \mathrm{M}_i, T, N \rangle$, are arranged contiguously in the list. This list is partitioned into p equal chunks and assigned to the different processors for a more balanced computation load (line 5). Then, the posterior probabilities for all the local candidate splits are computed and stored on each processor (lines 6 – 7). Finally, for each node, J candidate splits are selected randomly using the posterior probabilities as weights and another J splits are selected uniformly at random (lines 8 – 13). For ease of presentation, we demonstrate the selection of splits using previously defined oracle functions for random sampling. In the actual implementation, the contiguous arrangement of candidate splits for every node allows us to compute the split weights for random sampling for all the nodes using a single segmented parallel *scan* over the distributed *cand-probs*$_k$. Then, the splits for all the nodes in *cand-splits*$_k$ are selected independently on each processor, followed by an *all-gather* call to collect all the chosen splits for all the nodes on all the processors.

Algorithm 6: Parallel Learning of Modules

1 **function** LEARN-MODULE-CPDS():
  Input: m, D, $\mathcal{M}$, $\mathcal{P}$, U, B, J
2 **parallel** k = *rank of processor* **do**
3 **for** $\mathrm{M}_i \in \mathcal{M}$ **do**
4 $\mathcal{T}(\mathrm{M}_i)$ ← LEARN-TREE-STRUCT(m, D, M_i, U, B)
5 LEARN-TREE-SPLITS(D, $\mathcal{M}$, $\mathcal{T}$, $\mathcal{P}$, J)
6 LEARN-PARENTS($\mathcal{M}$, *wr-splits*, *ur-splits*)

The size of *cand-splits*$_k$, and therefore *cand-probs*$_k$, is bounded by $O(KRLnm/p)$ and computing the posterior probability for a split requires $O(Sm)$ time. Choosing J splits for every node in parallel, using segmented parallel *scan* and *all-gather*, takes $O(JKRLnm/p + \log p)$ computation time and $O(\tau \log p + \mu JKRL)$ communication time. Therefore, this phase takes $O(KRLSnm^2/p + \log p)$ time for computation and $O(\tau \log p + \mu JKRL)$ time for communication.

Our parallel algorithm for the last task is shown in Algorithm 6. In the interest of space, we omit a detailed pseudo-code description for the last phase in the task that computes scores for parents of

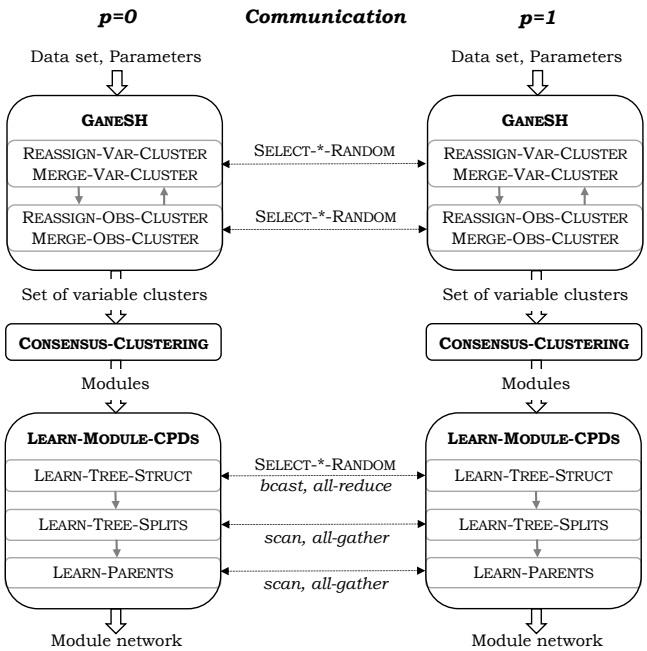


Figure 2: Schematic diagram showing the execution flow of our parallel algorithm for learning module networks with two processors, using the parallel functions developed in Section 3.

each module from the selected node splits. The parallelization of this phase is trivial and is implemented in Learn-Parents() function using a segmented parallel *scan* followed by an *all-gather* call. This phase requires $O(JKRL/p + \log p)$ computation and $O(\tau \log p + \mu JKRL)$ communication time in parallel. Summing up the run-times of the phases and simplifying it in terms of the input parameters, Learn-Module-CPDs() takes $O(UKLSnm^2/p + UL \log p)$ time in computation and $O(UKm(\tau + \mu) \log p)$ time in communication.

A schematic diagram for the execution flow of our parallel algorithm for learning MoNets, when using two processors, is shown in Figure 2. The schematic demonstrates the interactions between the different tasks as well as between the different phases within each task. Further, it shows the communications required by the parallel functions for the different phases during the execution of the algorithm.

4 IMPLEMENTATION

4.1 Sequential Implementation

Lemon-Tree software uses *Java* to implement the approach outlined by Bonnet *et al.* [13]. Even though any software written in *Java* requires compilation, it is referred to as an interpreted language [16]. This is because the byte-code produced by the compilation is interpreted and executed by a platform-independent virtual machine (VM), thus trading performance for portability. Consequently, multiple studies have shown that the performance of *Java* is inferior to that of *C++* for in-memory tasks [16, 22, 56]. We implemented the approach by Bonnet *et al.* using *C++*, adhering to the *C++14* standard, and optimized it for improved sequential run-time performance as shown in Section 5.2.1.

As discussed in Section 1, *Lemon-Tree* is a popular software that has been used in multiple studies for learning MoNets. Therefore, we used *Lemon-Tree* as the baseline for our implementation and ensured that our implementation produces exactly the same output as *Lemon-Tree*, given the same input data set and execution parameters. We had to modify the *Lemon-Tree* implementation to achieve this because of the following reasons. First, the execution of the learning algorithm requires generation of random numbers, which is accomplished in the original *Lemon-Tree* by a *Java* pseudo-random number generator (PRNG) library that is not available for *C++*. Therefore, we modified the *Lemon-Tree* code to use the same PRNG as the one used by our implementation via *Java Native Interface*. Then, we observed that some of the calls to the PRNG were superfluous and we eliminated them in both our implementation as well as *Lemon-Tree*. Finally, we discovered a bug in the implementation of the *GaneSH* algorithm in *Lemon-Tree* that we fixed and submitted to the maintainers of *Lemon-Tree*. We have provided this modified version of *Lemon-Tree* as an artifact and use it for the performance results presented in Section 5.2.1.

4.2 Parallel Implementation

We implemented the parallel algorithms proposed in Section 3 using *MPI* conforming to the *MPI 3.1* standard. For generating random numbers in parallel, we use the *TRNG* library that provides multiple parallelizable PRNGs [8]. We used a multiple recursive generator [29] with 3 feedback terms and a Sophie-Germain prime modulus for the experiments reported in Section 5. Note that our implementation can use any parallel PRNG supported by the library.

In order to implement the distributed random sampling functions described in Section 3.1, Select-Wtd-Rand() and Select-Unif-Rand(), same random number should be generated on all the parallel processors in a call to these functions. We accomplish this by initializing the PRNG with the same seed on all the processors and ensuring that the state of the PRNG is the same on all the processors before the calls to these functions. We also need to match the block distribution of work with the block distribution of the corresponding stream of random numbers between the executing processors, in order to generate the same output when using different numbers of processors. This is achieved in our parallel implementation by block splitting the parallel PRNGs which takes $O(1)$ time [8].

5 EXPERIMENTS AND RESULTS

We performed our experiments on the Phoenix cluster at Georgia Tech [46], where each node has a 2.7 GHz 24-core Intel Xeon Gold 6226 processor and main memory of 192 GB or more. The nodes run `RHEL 7.6` operating system and are connected via HDR100 (100 Gbps) InfiniBand. We compiled the source code, implemented with *C++14* and *MPI*, using `gcc v10.1.0` with `-O3 -march=native` optimization flags and `MVAPICH2 v2.3.3` implementation of MPI. For our experiments reported in this section, we assign 24 MPI processes per node by binding one MPI process to each core.

5.1 Data sets

In order to test the scalability of our implementation, we use gene regulatory networks as the target application area. Since gene regulatory networks have a hierarchical structure and data sets for studying these are typically sparse, MoNets have been successfully applied in numerous gene regulatory studies for various organisms spanning a wide range of complexity – from viruses and bacteria [51, 55, 61] to plants and animals [45, 60]. In this section, we demonstrate the use of our parallel implementation to learn genome-scale gene regulatory networks from two real gene expression data sets with thousands of observations for tens of thousands of genes.

The first gene expression data set that we use is generated from the organism *S. cerevisiae*, colloquially known as Brewer's yeast. Tchourine *et al.* [59] created this data set by aggregating data from multiple RNA-seq studies and it contains 5,716 genes and 2,577 observations for the genes. To further demonstrate the parallel scalability of our implementation on tens of thousands of variables, we used a second data set for the model plant *A. thaliana*. This data set contains 5,102 observations for 18,373 genes and is generated from multiple microarray experiments that studied the *development* process in the plant [3].

For the experiments in this section, we only report the minimum run-time required for learning MoNets from the data sets, i.e., we execute a single *GaneSH* run with one update step and construct only one regression tree structure for each module in the last task. We use all the genes in the data sets as the candidate regulators, i.e., all the variables are treated as candidate parents for all the modules. As noted in Section 2.2, this may lead to cyclic structures in the learned MoNet. The acyclicity constraint can be enforced as

a post-processing step in parallel using the methods developed in the previous works on BN structure learning [42], and is outside the scope of this work. All the runs are repeated three times with different random seeds and the average run-times are reported.

5.2 Sequential Performance

We compiled *Lemon-Tree* with *OpenJDK* v1.8.0_262 and executed it using the corresponding server VM for the run-times reported here.

5.2.1 Comparison with Lemon-Tree. We compared the run-time of the modified *Lemon-Tree* with that of our optimized sequential implementation (both described in Section 4.1) for constructing MoNets. Both *Lemon-Tree* as well as our implementation did not finish learning MoNet for the complete *S. cerevisiae* data set in seven days. Therefore, we created smaller data sets for these experiments using subsamples of $n = \{1000, 2000, 3000\}$ variables and $m = \{125, 250, 500, 750, 1000\}$ observations chosen from the complete data set. The performance of our implementation is compared with that of *Lemon-Tree* in Table 1 on these data sets. Our optimized sequential implementation shows a 3.6–3.8X speedup over *Lemon-Tree* for constructing MoNets from all the data sets. We also verified that our implementation learns the exact same MoNets as the ones learned by *Lemon-Tree* in all the cases.

n	m	Run-time (s)		Speedup
		Lemon-Tree	*Ours*	
1,000	125	416.0	110.3	3.8
	250	1,609.9	428.3	3.8
	500	6,307.9	1,686.2	3.7
	750	13,441.5	3,574.5	3.8
	1,000	25,253.6	6,680.7	3.8
2,000	125	1,407.5	392.8	3.6
	250	5,747.2	1,562.7	3.7
	500	23,258.4	6,202.3	3.7
	750	52,606.2	14,038.7	3.7
	1,000	91,202.7	24,327.0	3.7
3,000	125	2,942.8	792.0	3.7
	250	11,962.1	3,193.4	3.7
	500	50,838.0	13,553.9	3.8
	750	108,545.5	28,942.3	3.8
	1,000	197,493.4	52,709.6	3.8

Table 1: Comparison of the time taken by *Lemon-Tree* and our sequential implementation in constructing MoNets using the first n variables and m observations of the *yeast* data set, measured in seconds, and the corresponding speedup.

5.2.2 Sequential Run-time Estimates for Large Data sets. Both the sequential implementations are not able to construct a MoNet from the complete *S. cerevisiae* data set within a week. Therefore, we estimated the sequential run-time of the two implementations for learning from large data sets based on the growth rate of the sequential run-time of our implementation observed on smaller data sets. To this end, we measured the run-time of our implementation

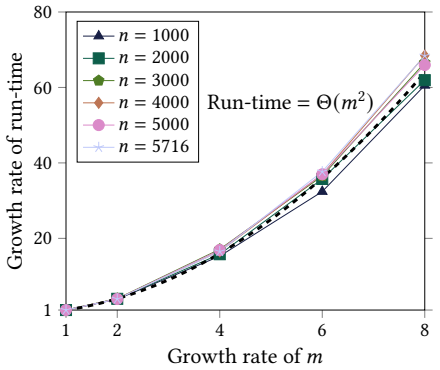


Figure 3: Plots of sequential run-time growth rate as the number of observations grow for data sets with different number of variables.

for constructing MoNets using 30 smaller data sets constructed from the complete *S. cerevisiae* data set by choosing combinations of the first $n = \{1000, 2000, 3000, 4000, 5000, 5716\}$ variables and the first $m = \{125, 250, 500, 750, 1000\}$ observations in the data set.

Figure 3 shows the plots of run-time growth rate as a function of n, while keeping m fixed. For a given n, the rate of increase is computed with respect to the smallest data set, i.e., compared to $m = 125$. The plots for six different values of n show close to quadratic growth rate of run-time for a linear increase in m, indicated by the dashed black line in the figure. We also plot the run-time growth rate as n is increased for five different values of m, in Figure 4, with $n = 1,000$ as the baseline. The quadratic growth rate is again denoted by the dashed black line in the figure. However, we observe that the run-time growth rate with increasing n is slower than quadratic for all the different values of m. We also plot $n^{1.8}$ growth rate in the figure, shown with dashed gray line, that seems to be a lower bound for the growth rate. From the two plots, we estimate the sequential run-time growth rate of our implementation to be $\Theta(m^2)$ for a fixed n and bounded between $O(n^2)$ and $\Omega(n^{1.8})$ for a fixed m. Comparing these empirical estimates with the sequential

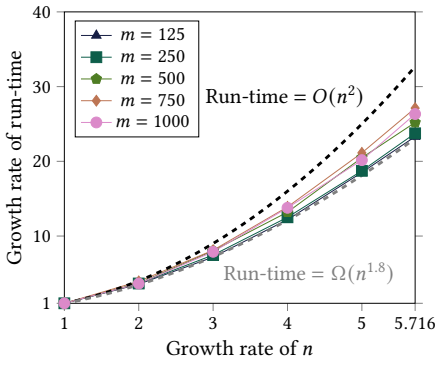


Figure 4: Plots of sequential run-time growth rate as the number of variables grow for data sets with different number of observations.

run-time complexity (Equation 1), we observe that the growth rate with increasing m corresponds well to the complexity. The super-linear growth in run-time with increasing n, on the other hand, can be attributed to a corresponding increase in the number of modules (K) from 28–39 for $n = 1,000$ to 111–170 for $n = 5,716$.

The average sequential run-time of our implementation for learning MoNets from the data set with $n = 5,716$ and $m = 1,000$ is 175,932.7 seconds. Using the growth rate of $\Theta(m^2)$ for a fixed n, we estimate the run-time of our implementation for learning MoNet from the complete *S. cerevisiae* data set as $175,932.7 \times (2,577/1,000)^2$ seconds or 324.5 hours which is about 13.5 days. We were able to verify that this estimate is reasonably accurate using a single sequential run for one random seed that took 325.1 hours. Further, our implementation provides a minimum sequential speedup of 3.6X over *Lemon-Tree*. Therefore, we estimate that *Lemon-Tree* would require a minimum of 48.6 days in order to construct a MoNet for the complete *S. cerevisiae* data set. Similarly, we also estimate the lower bound on the run-time of our sequential implementation for the complete *A. thaliana* data set as $175,932.7 \times (5,102/1,000)^2 \times (18,373/5,716)^{1.8}$ seconds which is 433.6 days or more than 14 months. The corresponding estimated lower bound on the run-time of *Lemon-Tree* is 1561 days which is more than 4 years.

5.3 Parallel Scalability

Our parallel implementation begins the construction of MoNets by reading the given data set in parallel. This is accomplished by block distributing the variables in the data set to the MPI processes – one process per core. Then, every process reads the observations for the variables assigned to it. Finally, the observations for all the variables are communicated to all the processes so that each process has the complete data set. Note that, while this causes duplication of data within the same node, it avoids the use of hybrid shared-memory and distributed-memory programming. This duplication is a non-issue because the problem is compute-bound due to its NP-hard nature, and the data sets are relatively small compared to the available memory size. For example, the size of the larger *A. thaliana* data set is still only 785 MB. During the parallel execution, any intermediate files and the final MoNet structure in XML format are written to the disk by the process with rank 0. In our experiments, we observed that the time for I/O is much smaller than the time required for learning the network, e.g., reading the complete *S. cerevisiae* data set in parallel takes 0.6–6.8 seconds and writing the output takes 1.4–20.8 seconds. We therefore disregard the time required for reading and writing files and only report the time required for learning the network in this section.

We evaluate the scalability of our parallel implementation by conducting strong scaling experiments because our primary motivation is to construct MoNets for specific use cases which are beyond the reach of sequential computing. Understanding the compromise between run-time and computational resources for solving these problems will help biologists choose the optimal trade-off for their specific needs. We use the following metrics for the scalability discussions:

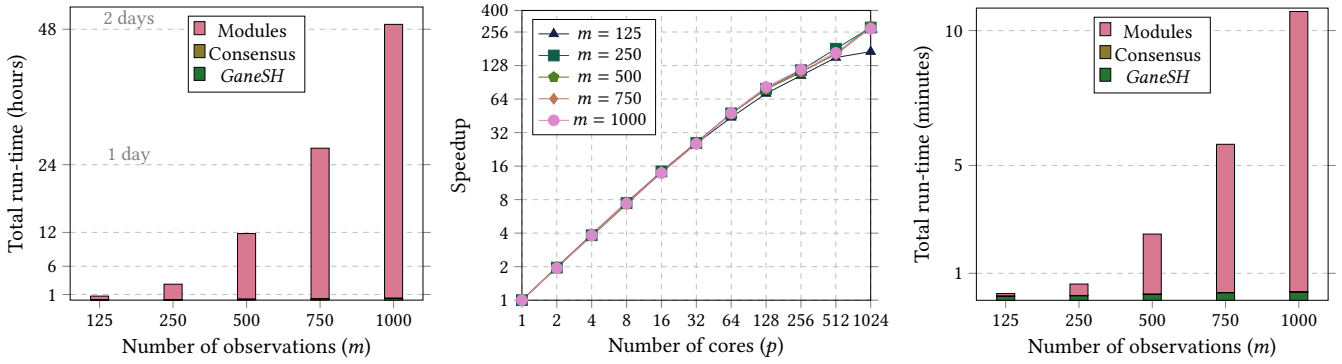
$$\text{Strong Scaling Speedup} = \frac{T_1}{T_p} \text{ and Efficiency } (\%) = \frac{T_1}{p \cdot T_p} \times 100\%$$

where T_1 is the run-time of the best sequential implementation and T_p is the run-time of the parallel implementation when using p cores. We use the run-time of our optimized sequential implementation as T_1 in all the cases, since it has been established as the faster one in the previous section. In cases where running with $p = 1$ is infeasible, we also refer to relative speedup and efficiency between parallel execution using p_1 and $p_2 (\geq p_1)$ cores, defined as:

$$\text{Relative Speedup} = \frac{T_{p_1}}{T_{p_2}} \text{ and Efficiency } (\%) = \frac{p_1 \cdot T_{p_1}}{p_2 \cdot T_{p_2}} \times 100\%$$

where T_{p_1} and T_{p_2} are run-times when using p_1 and p_2 cores.

5.3.1 Strong Scaling for Small Data sets. Since the sequential run-time of our implementation for the complete *S. cerevisiae* data set is estimated to be about two weeks, we conducted strong scaling experiments using smaller data sets from which MoNets can be learned sequentially in a more reasonable time. We created five data sets by selecting a subset of observations ($m = \{125, 250, 500, 750, 1000\}$) for all the variables in the complete data set ($n = 5,716$). The time required for learning MoNets from these data sets using our optimized sequential implementation is shown in Figure 5a with the



(a) Sequential run-times **(b) Strong scaling speedups** **(c) Run-times using 1024 cores**

Figure 5: Plots showing the scalability of our implementation for data sets with different number of observations subsampled from the complete *S. cerevisiae* data set.

time taken by different tasks indicated by different colors. The total sequential run-time for the five data sets varies from 43 minutes for $m = 125$ to more than two days for $m = 1000$. Further, the majority of the sequential run-time is spent in learning the modules. The fraction of the total run-time spent in the task increases from 94.7% for $m = 125$ to 99.4% for $m = 1000$. The consensus clustering task takes less than one second in all the cases.

We learned MoNets from these five data sets in parallel by varying the number of cores (p) from 2 to 1024. Figure 5b shows the strong scaling speedup plots for these data sets. Our parallel implementation scales well for all the data sets when using smaller number of cores. However, for the $m = 125$ data set, the plot diverges from that for the other data sets for larger number of cores. This is explained by the comparatively meager amount of work required for this data set, as is evident from the corresponding total run-time of less than 60 seconds when using 64 cores or more.

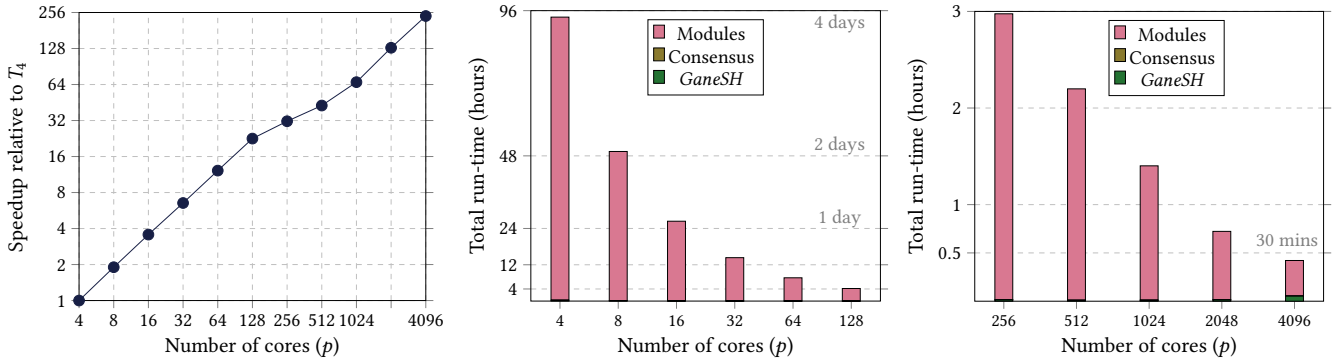
Our implementation achieves close to 48X speedup for the four larger data sets when using 64 cores, corresponding to a 75% efficiency. However, the scaling tapers off as the number of cores is increased because of the load imbalance in the most time consuming phase of the last task – the loop for computing posterior probabilities for all the candidate splits (lines 6 – 7 in Algorithm 5). The posterior probabilities for the splits are computed by discrete sampling for a maximum of S steps. Therefore, the time required for this phase cannot be estimated a priori and varies significantly across splits. As a measure of the load imbalance in this loop across processes, we computed the deviation of the maximum run-time of the loop on any process from the average run-time of the loop across all the processes, normalized by the average run-time. For the largest of the five data sets, the measured load imbalance is less than 0.3 when $p \leq 64$, indicating a reasonably good balance, and then the imbalance steadily increases from 0.5 using $p = 128$ to 2.6 using $p = 1024$. Consequently, the four bigger data sets achieve similar speedups in the range of 273.9–288.3X when $p = 1024$.

The time required for learning MoNets from the five data sets using 1024 cores is shown in Figure 5c. Our parallel implementation reduces the run-time for the two larger data sets from 26.9 and 48.9 hours to 5.8 and 10.7 minutes, respectively, while the learning is completed in less then 60 seconds for the two smaller data sets. Even though Figure 5c shows a higher percentage of run-time in the *GaneSH* task on 1024 cores, when compared to Figure 5a, more than 90% of the run-time is still spent in learning the modules from the three larger data sets.

5.3.2 Scaling for the complete S. cerevisiae data set. We used our parallel implementation to construct MoNets from the complete *S. cerevisiae* data set. In order to limit the time required for the experiments, we used a minimum of 4 cores for these experiments and discuss relative speedup and efficiency with respect to T_4 for this data set. We learned the networks from the data set by repeatedly doubling the number of cores used from 4 to 4096 and plot the relative speedup in Figure 6a.

We show the run-times obtained from the executions using 128 cores and fewer in Figure 6b and those using 128 to 4096 cores in Figure 6c, to accommodate the differences in the scales of the run-times. Our parallel implementation scales well when the number of cores is increased from 4 to 128, reducing the time required for learning the network from close to 4 days using $p = 4$ to about 4 hours using $p = 128$ with a relative speedup of 22.6 and more than 70% relative efficiency. The *GaneSH* task takes less than 0.38% of the total run-time on these cores and is therefore not a visible component of the run-time. The consensus clustering step, even though it is run sequentially, takes less than one second.

Our parallel implementation is able to learn a network from the complete data set in 23.5 minutes using 4096 cores, down from an estimated two weeks sequentially. Due to the comparatively lower work required by the *GaneSH* task – it takes about a minute when using 128 cores or more – and the load imbalance in the computations for candidate parent splits as discussed in 5.3.1, the relative speedup from $p = 4$ to $p = 4096$ is 239.3X corresponding to a relative efficiency of 23.4%. Nevertheless, to construct a MoNet in a computational biology pipeline, a run-time of 23.5 minutes presents a significant saving of computation time as compared to more than 13 days for a sequential run. Further, the difference between a run-time of 24 minutes and the ideal possible run-time of 6 minutes (at 100% relative efficiency) for MoNet learning from



(a) Relative speedup **(b) Run-times using fewer cores** **(c) Run-times using more cores**

Figure 6: Plots showing the run-times of our implementation for the complete *S. cerevisiae* data set using different number of cores and the corresponding relative speedup.

data sets created by wet-lab biological experiments is immaterial, given that conducting these wet-lab experiments can take days.

5.3.3 Scaling for the complete A. thaliana data set. We estimated, in Section 5.2.2, that our optimized sequential implementation will require approximately 14 months for learning a MoNet for the complete *A. thaliana* data set, a significant impediment in practice. Using our scalable parallel method, genome-scale regulatory networks can be learned in a reasonable time from large data sets for multi-cellular organisms with tens of thousands of genes.

Table 2 shows the time required for learning networks for the complete *A. thaliana* data set. Since learning of MoNets from the data set using smaller number of cores will require prohibitively long time, we learned MoNets from the data set by varying the number of cores from 256 to 4096 cores. Our parallel implementation reduces the run-time from almost two days using 256 cores to about 4 hours using 4096 cores. The table also shows relative speedup and efficiency compared to the run-time using 256 cores. While the scaling efficiency relative to 256 cores for the *S. cerevisiae* data set is close to 47% on 4096 cores in Section 5.3.2, the corresponding relative scaling efficiency for the *A. thaliana* data set increases to almost 70%.

Number of Cores (p)	Run-time (s)	Relative to T_{256}	
		Speedup	Efficiency (%)
256	168, 775.6	1.0	100.0
512	91, 349.6	1.8	92.4
1024	54, 099.1	3.1	78.0
2048	28, 529.3	5.9	73.9
4096	15, 097.6	11.2	69.9

Table 2: Parallel run-times for the complete *A. thaliana* data set using large number of cores and the corresponding relative speedup and efficiency.

6 CONCLUSIONS AND FUTURE WORK

We presented the first distributed-memory parallel approach for the construction of MoNets that scales to a large number of cores. Our parallel implementation learns genome-scale gene regulatory networks for two model organisms – *S. cerevisiae* and *A. thaliana*, in 24 minutes and 4.2 hours using 4096 cores, as compared to an estimated 49 and 1561 days, respectively, using the previous state-of-the-art sequential implementation. The proposed method is general and can enable learning of high-dimensional MoNets for analyses of big data in any of its wide array of applications, e.g., single cell genomics [58] where a data set can include hundreds of thousands of observations. We hope that it can also help the adoption of MoNets in novel domains, such as applications that use other parameter-sharing variations of BNs, where the untenable time required for sequentially learning MoNets from large data sets has been a deterrent thus far.

Potential future works can further improve the scalability of our proposed parallel method by implementing a dynamic load balancing scheme for computing the posterior probabilities for all the candidate parent splits. The proposed parallel components can also be extended to develop a parallel solution for *GENOMICA* that scales to thousands of cores.

ACKNOWLEDGEMENT

This research is supported in part by the National Science Foundation under OAC-1828187, OAC-1854828, and CCF-1718479.

REFERENCES

[1] Ashar Ahmad and Holger Fröhlich. 2016. Integrating heterogeneous omics data via statistical inference and learning techniques. *Genomics and Computational Biology* 2, 1 (2016), e32–e32.
[2] Pamela A Alexandre, Lisette JA Kogelman, Miguel HA Santana, Danielle Passarelli, Lidia H Pulz, Paulo Fantinato-Neto, Paulo L Silva, Paulo R Leme, Ricardo F Strefezzi, Luiz L Coutinho, et al. 2015. Liver transcriptomic networks reveal main biological processes associated with feed efficiency in beef cattle. *BMC genomics* 16, 1 (2015), 1–13.
[3] Maneesha Aluru and Sriram Chockalingam. 2021. *A. thaliana Gene Expression Dataset for Development Processes.* https://doi.org/10.5281/zenodo.4672797
[4] Stilianos Arhondakis, Craita E Bita, Andreas Perrakis, Maria E Manioudaki, Afroditi Krokida, Dimitrios Kaloudas, and Panagiotis Kalaitzis. 2016. In silico transcriptional regulatory networks involved in tomato fruit ripening. *Frontiers in plant science* 7 (2016), 1234.
[5] Elham Azizi, Edoardo Airoldi, and James Galagan. 2014. Learning modular structures from network data and node variables. In *International conference on machine learning*. PMLR, 1440–1448.
[6] Yang Bai, Laura Dougherty, Lailiang Cheng, Gan-Yuan Zhong, and Kenong Xu. 2015. Uncovering co-expression gene network modules regulating fruit acidity in diverse apples. *BMC genomics* 16, 1 (2015), 1–16.
[7] Alexis Battle, Eran Segal, and Daphne Koller. 2005. Probabilistic discovery of overlapping cellular processes and their regulation. *Journal of Computational Biology* 12, 7 (2005), 909–927.
[8] Heiko Bauke and Stephan Mertens. 2007. Random numbers for large-scale distributed Monte Carlo simulations. *Physical Review E* 75, 6 (2007), 066701.
[9] Kelli Crews Baumgartner, Silvia Ferrari, and C Gabrielle Salfati. 2005. Bayesian network modeling of offender behavior for criminal profiling. In *Proceedings of the 44th IEEE Conference on Decision and Control*. IEEE, 2702–2709.
[10] Elham Behdani and Mohammad Reza Bakhtiarizadeh. 2017. Construction of an integrated gene regulatory network link to stress-related immune system in cattle. *Genetica* 145, 4 (2017), 441–454.
[11] A Beresniak, E Bertherat, W Perea, G Soga, R Souley, D Dupont, and S Hugonnet. 2012. A Bayesian network approach to the study of historical epidemiological databases: modelling meningitis outbreaks in the Niger. *Bulletin of the World Health Organization* 90 (2012), 412–417a.
[12] Bonnie Berger, Jian Peng, and Mona Singh. 2013. Computational solutions for omics data. *Nature reviews genetics* 14, 5 (2013), 333–346.
[13] Eric Bonnet, Laurence Calzone, and Tom Michoel. 2015. Integrative multi-omics module network inference with Lemon-Tree. *PLoS Comput Biol* 11, 2 (2015), e1003983.
[14] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees.* CRC press.
[15] David Maxwell Chickering, David Heckerman, and Christopher Meek. 2004. Large-sample learning of Bayesian networks is NP-hard. *Journal of Machine Learning Research* 5, Oct (2004), 1287–1330.
[16] Luca Gherardi, Davide Brugali, and Daniele Comotti. 2012. A java vs. c++ performance evaluation: a 3d modeling benchmark. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 161–172.
[17] Thomas L Griffiths. 2004. *Causes, coincidences, and theories.* Ph.D. Dissertation. stanford university.
[18] Thomas L Griffiths and Joshua B Tenenbaum. 2009. Theory-based causal induction. *Psychological review* 116, 4 (2009), 661.
[19] Elias Gyftodimos and Peter A Flach. 2004. Hierarchical Bayesian networks: an approach to classification and learning for structured data. In *Hellenic Conference on Artificial Intelligence*. Springer, 291–300.
[20] David Heckerman. 2008. A tutorial on learning with Bayesian networks. *Innovations in Bayesian networks* (2008), 33–82.
[21] Katherine A Heller and Zoubin Ghahramani. 2005. Bayesian hierarchical clustering. In *Proceedings of the 22nd international conference on Machine learning*. 297–304.
[22] Hugo Heyman and Love Brandefelt. 2020. A Comparison of Performance & Implementation Complexity of Multithreaded Applications in Rust, Java and C++.
[23] Zena M Hira and Duncan F Gillies. 2015. A review of feature selection and feature extraction methods applied on microarray data. *Advances in bioinformatics* 2015

(2015).
[24] Hongshan Jiang, Chunrong Lai, Wenguang Chen, Yurong Chen, Wei Hu, Weimin Zheng, and Yimin Zhang. 2006. Parallelization of module network structure learning and performance tuning on SMP. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 8–pp.
[25] Anagha Joshi, Riet De Smet, Kathleen Marchal, Yves Van de Peer, and Tom Michoel. 2009. Module networks revisited: computational assessment and prioritization of model predictions. *Bioinformatics* 25, 4 (2009), 490–496.
[26] Anagha Joshi, Yves Van de Peer, and Tom Michoel. 2008. Analysis of a Gibbs sampler method for model-based clustering of gene expression data. *Bioinformatics* 24, 2 (2008), 176–183.
[27] Elisavet Kaitetzidou, Jenny Xiang, Efthimia Antonopoulou, Constantinos S Tsigenopoulos, and Elena Sarropoulou. 2015. Dynamics of gene expression patterns during early development of the European seabass (Dicentrarchus labrax). *Physiological Genomics* 47, 5 (2015), 158–169.
[28] Dietmar Kasper, Galia Weidl, Thao Dang, Gabi Breuel, Andreas Tamke, Andreas Wedel, and Wolfgang Rosenstiel. 2012. Object-oriented Bayesian networks for detection of lane change maneuvers. *IEEE Intelligent Transportation Systems Magazine* 4, 3 (2012), 19–31.
[29] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.
[30] Daphne Koller. 1999. Probabilistic relational models. In *International Conference on Inductive Logic Programming*. Springer, 3–13.
[31] Daphne Koller and Avi Pfeffer. 1997. Object-Oriented Bayesian Networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*. 302–313.
[32] Pieter Kraaijeveld, Marek J Druzdzel, Agnieszka Onisko, and Hanna Wasyluk. 2005. Genierate: An interactive generator of diagnostic bayesian network models. In *Proc. 16th Int. Workshop Principles Diagnosis*. Citeseer, 175–180.
[33] Long Liu, Wei Hu, Chunrong Lai, Hong-shan Jiang, Wenguang Chen, Weimin Zheng, and Yimin Zhang. 2005. Parallel module network learning on distributed memory multiprocessors. In *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. IEEE, 129–134.
[34] Xinguo Lu, Xing Li, Ping Liu, Xin Qian, Qiumai Miao, and Shaoliang Peng. 2018. The integrative method based on the module-network for identifying driver genes in cancer subtypes. *Molecules* 23, 2 (2018), 183.
[35] Youtao Lu, Xiaoyuan Zhou, and Christine Nardini. 2017. Dissection of the module network implementation "LemonTree": enhancements towards applications in metagenomics and translation in autoimmune maladies. *Molecular BioSystems* 13, 10 (2017), 2083–2091.
[36] Saisai Ma, Jiuyong Li, Lin Liu, and Thuc Duy Le. 2016. Mining combined causes in large data sets. *Knowledge-Based Systems* 92 (2016), 104–111.
[37] Suzanne M Mahoney and Kathryn B Laskey. 1996. Network Engineering for Complex Belief Networks. In *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*. 389–396.
[38] Fabio Albuquerque Marchi, David Correa Martins, Mateus Camargo Barros-Filho, Hellen Kuasne, Ariane Fidelis Busso Lopes, Helena Brentani, Jose Carlos Souza Trindade Filho, Gustavo Cardoso Guimarães, Eliney F Faria, Cristovam Scapulatempo-Neto, et al. 2017. Multidimensional integrative analysis uncovers driver candidates and biomarkers in penile carcinoma. *Scientific reports* 7, 1 (2017), 1–11.
[39] Florian Markowetz and Rainer Spang. 2007. Inferring cellular networks–a review. *BMC bioinformatics* 8, 6 (2007), 1–17.
[40] Tom Michoel, Steven Maere, Eric Bonnet, Anagha Joshi, Yvan Saeys, Tim Van den Bulcke, Koenraad Van Leemput, Piet Van Remortel, Martin Kuiper, Kathleen Marchal, et al. 2007. Validating module network learning algorithms using simulated data. *BMC bioinformatics* 8, 2 (2007), 1–15.
[41] Tom Michoel and Bruno Nachtergaele. 2012. Alignment and integration of complex networks by hypergraph-based spectral clustering. *Physical Review E* 86, 5 (2012), 056111.
[42] Sanchit Misra, Md Vasimuddin, Kiran Pamnany, Sriram P Chockalingam, Yong Dong, Min Xie, Maneesha R Aluru, and Srinivas Aluru. 2014. Parallel bayesian network structure learning for genome-scale gene networks. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 461–472.
[43] Anuradha Moirangthem, Xue Wang, Irene K Yan, and Tushar Patel. 2018. Network analyses–based identification of circular ribonucleic acid–related pathways in intrahepatic cholangiocarcinoma. *Tumor Biology* 40, 9 (2018), 1010428318795761.
[44] Radu Stefan Niculescu, Tom M Mitchell, R Bharat Rao, Kristin P Bennett, and Emilio Parrado-Hernández. 2006. Bayesian Network Learning with Parameter Constraints. *Journal of machine learning research* 7, 7 (2006).
[45] Noa Novershtern, Zohar Itzhaki, Ohad Manor, Nir Friedman, and Naftali Kaminski. 2008. A functional and regulatory map of asthma. *American journal of respiratory cell and molecular biology* 38, 3 (2008), 324–336.
[46] PACE. 2017. *Partnership for an Advanced Computing Environment (PACE)*. http://www.pace.gatech.edu
[47] Dana Pe'er, Amos Tanay, Aviv Regev, and Tommi Jaakkola. 2006. MinReg: A scalable algorithm for learning parsimonious regulatory networks in yeast and mammals. *Journal of Machine Learning Research* 7, 2 (2006).
[48] Dana Pe'er, Aviv Regev, Gal Elidan, and Nir Friedman. 2001. Inferring subnetworks from perturbed expression profiles. *Bioinformatics* 17, suppl_1 (2001), S215–S224.
[49] Eran Segal, Nir Friedman, Naftali Kaminski, Aviv Regev, and Daphne Koller. 2005. From signatures to models: understanding cancer using microarrays. *Nature genetics* 37, 6 (2005), S38–S45.
[50] Eran Segal, Nir Friedman, Daphne Koller, and Aviv Regev. 2004. A module map showing conditional activity of expression modules in cancer. *Nature genetics* 36, 10 (2004), 1090–1098.
[51] Eran Segal, Dana Pe'er, Aviv Regev, Daphne Koller, and Nir Friedman. 2003. Learning Module Networks. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*. 525–534.
[52] Eran Segal, Dana Pe'er, Aviv Regev, Daphne Koller, Nir Friedman, and Tommi Jaakkola. 2005. Learning module networks. *Journal of Machine Learning Research* 6, 4 (2005).
[53] Eran Segal, Michael Shapira, Aviv Regev, Dana Pe'er, David Botstein, Daphne Koller, and Nir Friedman. 2003. Module networks: identifying regulatory modules and their condition-specific regulators from gene expression data. *Nature genetics* 34, 2 (2003), 166–176.
[54] Eran Segal, Claude B Sirlin, Clara Ooi, Adam S Adler, Jeremy Gollub, Xin Chen, Bryan K Chan, George R Matcuk, Christopher T Barry, Howard Y Chang, et al. 2007. Decoding global gene expression programs in liver cancer by noninvasive imaging. *Nature biotechnology* 25, 6 (2007), 675–680.
[55] Sagi D Shapira, Irit Gat-Viks, Bennett OV Shum, Amelie Dricot, Marciela M de Grace, Liguo Wu, Piyush B Gupta, Tong Hao, Serena J Silver, David E Root, et al. 2009. A physical and regulatory map of host-influenza interactions reveals pathways in H1N1 infection. *Cell* 139, 7 (2009), 1255–1267.
[56] Suraj Sharma. 2019. Performance comparison of Java and C++ when sorting integers and writing/reading files.
[57] Valerie J Shute, Matthew Ventura, Malcolm Bauer, and Diego Zapata-Rivera. 2009. Melding the power of serious games and embedded assessment to monitor and foster learning. *Serious games: Mechanisms and effects* 2 (2009), 295–321.
[58] Oliver Stegle, Sarah A Teichmann, and John C Marioni. 2015. Computational and analytical challenges in single-cell transcriptomics. *Nature Reviews Genetics* 16, 3 (2015), 133–145.
[59] Konstantine Tchourine, Christine Vogel, and Richard Bonneau. 2018. Condition-specific modeling of biophysical parameters advances inference of regulatory networks. *Cell reports* 23, 2 (2018), 376–388.
[60] Vanessa Vermeirssen, Inge De Clercq, Thomas Van Parys, Frank Van Breusegem, and Yves Van de Peer. 2014. Arabidopsis ensemble reverse-engineered gene regulatory network discloses interconnected transcription factors in oxidative stress. *The Plant Cell* 26, 12 (2014), 4656–4679.
[61] Vanessa Vermeirssen, Anagha Joshi, Tom Michoel, Eric Bonnet, Tine Casneuf, and Yves Van de Peer. 2009. Transcription regulatory networks in Caenorhabditis elegans inferred through reverse-engineering of gene expression profiles constitute biological hypotheses for metazoan development. *Molecular BioSystems* 5, 12 (2009), 1817–1830.
[62] Jing Xu and Christian R Shelton. 2010. Intrusion detection using continuous time Bayesian networks. *Journal of Artificial Intelligence Research* 39 (2010), 745–774.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We utilized the Phoenix cluster at Georgia Tech for our experiments, using a maximum of 171 nodes of the cluster for the results reported in the paper. We compiled the source code, implemented with C++14 and MPI, using gcc v10.1.0 with -O3 -march=native optimization flags and MVAPICH2 v2.3.3 implementation of MPI. We measure the run-times by assigning 24 MPI processes per node and binding one MPI process to each core. We ran all the experiments for three different random seeds and reported the average run-times in all cases. A stepwise guide to running the experiments reported in the paper can be found at https://github.com/asrivast28/ParsiMoNe/blob/main/EXPERIMENTS.md

Author-Created or Modified Artifacts:

```
Persistent ID: https://doi.org/10.5281/zenodo.5144438
Artifact name: ParsiMoNe

Persistent ID:
↪ https://github.com/asrivast28/lemon-tree/commit/⌋
↪ 5c5b91a7dc24b8c092f45c19ab7961b7d9b4598a
Artifact name: Lemon-Tree
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Each node on the Phoenix cluster has a 2.7 GHz 24-core Intel Xeon Gold 6226 processor and a minimum of 192 GB of main memory. The nodes are connected via HDR100 (100 Gbps) InfiniBand. The data sets are stored on a GPFS filesystem, which is accessible from all the nodes.

Operating systems and versions: RHEL 7.6 running Linux kernel 3.10.0

Compilers and versions: gcc v10.1.0

Libraries and versions: MVAPICH2 v2.3.3, Boost v1.74.0, TRNG v4.22, Armadillo v9.800.3, SCons v3.1.2

Input datasets and versions: Yeast Microarray Dataset (DOI: 10.5281/zenodo.3355524), A. thaliana Gene Expression Dataset for Development Processes (DOI: 10.5281/zenodo.4672797)

URL to output from scripts that gathers execution environment information.

```
https://github.com/asrivast28/ParsiMoNe/blob/f653f48⌋
↪ a854bf5cd619bf55c6a4741c9072b9c5a/phoenix\_envir⌋
↪ onment.log
```