PARALLEL ALGORITHMS AND GENERALIZED FRAMEWORKS FOR LEARNING LARGE-SCALE BAYESIAN NETWORKS

A Dissertation
Presented to
The Academic Faculty

By

Ankit Srivastava

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering
College of Computing

Georgia Institute of Technology

August 2021

PARALLEL ALGORITHMS AND GENERALIZED FRAMEWORKS FOR LEARNING LARGE-SCALE BAYESIAN NETWORKS

Thesis committee:

Dr. Srinivas Aluru, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Constantine Dovrolis
School of Computer Science
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard W. Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Mark A. Davenport
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date approved: August 5th, 2021

"As far as I'm concerned, if something is so complicated that you can't explain it in 10 seconds, then it's probably not worth knowing anyway."

– Bill Watterson, *Calvin and Hobbes*

For *Papa*, *Mummy*, and *Baba*

ACKNOWLEDGMENTS

To my advisor, Dr. Srinivas Aluru, first and foremost for encouraging me to embark upon the journey that culminated in this dissertation. Over the years, he gave me freedom to explore my research interests and provided constant guidance and support. I feel extremely grateful for all the opportunities that he entrusted me with.

To my committee members – Dr. Ümit Çatalyürek, Dr. Mark Davenport, Dr. Constantine Dovrolis, and Dr. Richard Vuduc – for their valuable time and suggestions that made this dissertation possible. Also to Dr. Sriram P. Chockalingam whose contribution to this dissertation, in the form of his mentorship and collaboration on the works presented here, is invaluable.

To all the incredible labmates – Indranil Roy, Patrick Flick, Tony Pan, Nagakishore Jammula, Cansu Tetik, Harsh Shrivastava, Shruti Shivakumar, Neda Tavakoli, and all the others – who made me look forward to get to work in the mornings, and to many other fellow graduate students that became friends and made both work and life more enjoyable. Especially to my second family – Mohit Agarwal, Stephanie Grimm, Anuj Gupta, and Ashis Pati – who made Atlanta feel like home.

To all my earlier teachers – a special mention to Arun Mishra at Fatima High School, Mau and Dr. Saumyen Guha at IIT Kanpur – that molded my passion for science and research. To the people at ANSYS, Inc. – Hariharan Krishnan and Dr. Ravi Reddy Manumachu, who trusted me to work on high-performance computing problems before I had any formal training in the subject.

Most of all, to my parents who have stood by my side through the highs and the lows, cheering me on at every juncture. They encouraged me to follow my dreams, and made innumerable sacrifices to ensure that I had the means to do so. For all of this and more, I will forever be indebted.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

A. thaliana *Arabidopsis thaliana*

GS *Grow-Shrink*

IAMB *Incremental Association MB*

Inter-IAMB *Interleaved IAMB*

MMPC *Max-Min Parents and Children*

PC-stable the order-independent variant of the algorithm by *Peter and Clark*

PC the algorithm by *Peter and Clark*

S. aureus *Staphylococcus aureus*

S. cerevisiae *Saccharomyces cerevisiae*

SI-HITON *Semi-Interleaved HITON Parents and Children*

BN Bayesian network

CI conditional independence

CPD conditional probability distribution

CPDAG completed partially directed acyclic graph

DAG directed acyclic graph

DL deep learning

MB Markov blanket

ML machine learning

MoNet module network

MPI Message Passing Interface

PRNG pseudo-random number generator

SUMMARY

Bayesian networks (BNs) are an important subclass of probabilistic graphical models that employ directed acyclic graphs to compactly represent exponential-sized joint probability distributions over a set of random variables. Since BNs enable probabilistic reasoning about interactions between the variables of interest, they have been successfully applied in a wide range of applications in the fields of medical diagnosis, gene networks, cybersecurity, epidemiology, etc. Furthermore, the recent focus on the need for explainability in human-impact decisions made by machine learning (ML) models has led to a push for replacing the prevalent black-box models with inherently interpretable models like BNs for making high-stakes decisions in hitherto unexplored areas.

Learning the exact structure of BNs from observational data is an NP-hard problem and therefore a wide range of heuristic algorithms have been developed for this purpose. However, even the heuristic algorithms are compute-intensive. The existing software packages for BN structure learning with implementations of multiple algorithms are either completely sequential or support limited parallelism and can take days to learn BNs with even a few thousand variables. Previous parallelization efforts have focused on one or two algorithms for specific applications and have not resulted in broadly applicable parallel software. This has prevented BNs from becoming a viable alternative to other ML models.

In this dissertation, we develop efficient parallel versions of a variety of BN learning algorithms from two categories: six different *constraint-based* methods and a *score-based* method for constructing a specialization of BNs known as module networks. We also propose optimizations for the implementations of these parallel algorithms to achieve maximum performance in practice. Our proposed algorithms are scalable to thousands of cores and outperform the previous state-of-the-art by a large margin. We have made the implementations available as open-source software packages that can be used by ML and application-domain researchers for expeditious learning of large-scale BNs.

CHAPTER 1
INTRODUCTION AND MOTIVATION

Machine learning (ML) models are a constant presence in our life. From the time we get up until the time we fall asleep, we are actively interacting with search engines, email clients, social media platforms, news aggregators, e-commerce websites, digital streaming services, etc. Even while sleeping, we are passively interacting with fitness trackers that assess the quality of our sleep. All of these applications – and many more that we use on a daily basis – rely on ML models to implement different aspects of their functionality.

Since the advent of deep learning (DL) during the early part of the last decade, DL models have gradually replaced the other ML models in these applications [1]. DL models are a subset of ML models that first learn multiple layers of representation from a given data set and then use the learned representations to accomplish related tasks. This learning methodology seems to enable DL models to perform well in a wide variety of fields. Although, it also prevents their internal working from being interpretable, even by domain experts. This has led to their moniker of black-box models [2].

The lack of interpretability of DL models may not be of immediate concern in the applications discussed above. However, it is of undeniable importance in areas with potential for a significant impact on human lives. Indeed, the use of black-box models in high human-impact areas has already been shown to be problematic in multiple scenarios, e.g., criminal justice [3], medical diagnosis [4], pollution monitoring [5], etc. While there already exist laws that require a "right to explanation" in certain human-impact decisions – the Equal Credit Opportunity Act in the US [6] and the General Data Protection Regulation in Europe [7] being the most prominent examples – there is a general push towards using interpretable models for making high-stakes decisions [2, 8].

Bayesian networks (BNs) are one such type of interpretable ML model [9]. They are a

subclass of probabilistic graphical models that employ directed acyclic graphs (DAGs) to compactly represent exponential-sized joint probability distributions over a set of random variables [10]. Since BNs enable probabilistic reasoning about direct and indirect interactions between the variables of interest, they have already been successfully employed for making high-stakes decisions in the fields of medical diagnosis [11], legal reasoning [12], forensic science [13], and epidemiology [14]. Further, BNs have been used to remove biases from black-box models [15, 16]. They have also been used in a wide variety of other fields such as for construction of gene networks [17, 18], fMRI analysis [19], cybersecurity [20], etc. and have the potential for application in hitherto unexplored areas.

1.1 Challenges Facing the Adoption of Bayesian Networks

The use of BNs in real-world applications is not without its challenges. To compete with DL models, it should be possible to learn BNs from large data sets. However, given a data set sampled from a joint probability distribution, exact learning of the corresponding BN structure is NP-hard [21]. Correspondingly, as discussed in subsection 2.4.1, both sequential as well as parallel algorithms proposed for the purpose can only learn optimal structure for very small BNs. Therefore, as detailed in subsection 2.1.2, a wide range of heuristic algorithms have been developed for learning BN structure. However, the heuristic algorithms are also compute-intensive and need efficient parallel solutions to learn large-scale networks.

Although the learning of DL models is also compute-intensive, high-performance libraries like *PyTorch* [22] and *TensorFlow* [23] can efficiently utilize substantial computation resources to enable fast learning of these models from large data sets. On the other hand, while there exist open-source libraries for learning BNs with support for multiple structure-learning algorithms, e.g., *bnlearn* [24], *Tetrad* [25], and *pcalg* [26], their implementations are either completely sequential (e.g., *pcalg*) or support only limited intra-node level parallelism (e.g., *Tetrad*). Therefore, they can take days to learn BNs with even a few

thousand variables. Recently, *bnlearn* added support for parallelizing structure-learning algorithms using multiple nodes [27]. However, as we show in subsection 3.4.2, the parallelized algorithms in *bnlearn* do not scale beyond a single node. Previous parallelization approaches which demonstrated scalability to multiple nodes for learning large-scale networks focused on one or two algorithms for specific applications and have not resulted in broadly applicable parallel software. We posit that the lack of a high-performance library for learning BNs from large data sets is one of the key reasons that has prevented them from becoming a viable alternative to other ML models.

1.2 Dissertation Goals and Overview

The goal of this dissertation is to develop efficient parallel versions of a variety of BN learning algorithms and implement them as part of open-source libraries that can be used by ML and application-domain researchers for expeditious construction of large-scale BNs. Towards this end, we parallelize multiple learning algorithms from two different categories, *constraint-based* and *score-based*, that are described in more detail in subsection 2.1.2.

The rest of this dissertation is organized as follows. In chapter 2, we provide the required background for the work presented in the subsequent chapters and review the related works. In chapter 3, we present a general framework for parallelizing *constraint-based* algorithms. Then, we use it to propose efficient parallel algorithms for five different *constraint-based* algorithms. In chapter 4, we extend the framework to parallelize another important *constraint-based* algorithm. In chapter 5, we develop a parallel method for constructing a specialization of BNs using *score-based* methods, known as module networks (MoNets). Finally, in chapter 6, we conclude the work presented in this dissertation.

A majority of the original work presented in this dissertation is part of the following peer-reviewed publications:

- A. Srivastava, S. Chockalingam, and S. Aluru, "A Parallel Framework for Constraint-based Bayesian Network Learning via Markov Blanket Discovery," in *2020 SC20:*

International Conference for High Performance Computing, Networking, Storage and Analysis (SC), IEEE Computer Society, 2020, pp. 74–88

- A. Srivastava, S. Chockalingam, M. Aluru, and S. Aluru, "Parallel Construction of Module Networks," in *2021 SC21: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, 2021

In accordance with the aforementioned goals, we have made the optimized implementations of the parallel learning algorithms developed in this dissertation available as part of the following two open-source software packages:

- A. Srivastava, *ramBLe - A Parallel Framework for Bayesian Learning*, https://github.com/asrivast28/ramBLe, 2020

- A. Srivastava, *ParsiMoNe - Parallel Construction of Module Networks*, https://github.com/asrivast28/ParsiMoNe, 2021

CHAPTER 2

BACKGROUND AND RELATED WORK

We develop parallel algorithms for learning BNs and MoNets in this dissertation. Therefore, we first provide necessary context on some of the key topics that are relevant to the discussion in the rest of this dissertation. We begin this chapter by presenting required background on sequentially learning BNs in section 2.1. Next, we discuss the need for specializing BNs as MoNets as well sequential methods for constructing them in section 2.2. Then, we provide a brief introduction to germane parallel computing concepts in section 2.3. Finally, we discuss the relevant prior works in the field in section 2.4.

2.1 Bayesian Networks

2.1.1 Notations and Definitions

We use upper-case alphabets (e.g., X, X_i, Y) to represent random variables and calligraphic upper-case alphabets (e.g., $\mathcal{X}, \mathcal{PA}, \mathcal{CH}$) to represent sets of random variables. The values that a random variable can take are represented using lower-case letters (e.g., a, b, c). We represent conditional independence (CI) between two random variables X, Y given a third variable Z as $I(X, Y|Z)$ and conditional dependence as $\neg I(X, Y|Z)$. The strength of association between X and Y given Z is represented using $\mathit{Assoc}(X, Y|Z)$. We also use the $I(\cdot, \cdot|\cdot)$, $\neg I(\cdot, \cdot|\cdot)$, and $\mathit{Assoc}(\cdot, \cdot|\cdot)$ notations for sets of variables.

Let $\mathcal{X}$ be a set of n random variables $\{X_1, \ldots, X_n\}$. Let $\mathbb{G} = (\mathcal{X}, \mathbb{E})$ be a DAG with random variables in $\mathcal{X}$ as vertices and $\mathbb{E}$ as the set of edges between the vertices in $\mathbb{G}$. X_j is said to be a *parent* of X_i in $\mathbb{G}$ if the edge $X_j \rightarrow X_i$ exists in $\mathbb{E}$, and X_i is referred to as a *child* of X_j. The set of all the parents of X_i is denoted by $\mathcal{PA}(X_i)$, and the set of all the children of X_i is denoted as $\mathcal{CH}(X_i)$. The set of both the parents as well as the children of

X_i is represented by $\mathcal{PC}(X_i)$. X_k is said to be a *descendant* of X_i if a directed path exists from X_i to X_k ($X_i \rightarrow \ldots \rightarrow X_k$), and a *nondescendant* if no such path exists. The set of all the nondescendants of X_i is represented by $\mathcal{ND}(X_i)$.

Let Θ represent a joint probability distribution of the variables in $\mathcal{X}$. $(\mathbb{G}, \Theta)$ is said to satisfy the *Markov condition* if every $X \in \mathcal{X}$ is conditionally independent of the set of all its nondescendents given the set of all its parents, i.e., $I(X, \mathcal{ND}(X)|\mathcal{PA}(X))\ \forall X \in \mathcal{X}$. $(\mathbb{G}, \Theta)$ is a BN if it satisfies the Markov condition. Markov condition enables the decomposition of the joint probability distribution Θ in terms of probability distribution of the variables conditioned on their parents as follows:

$$\Theta(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|\mathcal{PA}(X_i))$$

Figure 2.1 shows an example BN for the six variables $\{A, B, C, D, E, F\}$. The directed arrows in the BN represent parent-child relationships as defined above, e.g., $\mathcal{R}(C)$ is $\{A\}$ and C is present in both $\mathcal{R}(E)$ and $\mathcal{R}(F)$ in the BN shown in the figure. The $\mathcal{PC}$ set of a variable in a BN consists of variables that are dependent on it given any conditioning set not containing the two variables, i.e., $X \in \mathcal{PC}(Y)$ if and only if $\neg I(X, Y|\mathcal{S})\ \forall \mathcal{S} \subseteq \mathcal{X} \setminus \{X, Y\}$. For example, in the BN shown in the figure, $\mathcal{PC}(C)$ is $\{A, E, F\}$. Using the Markov condition, the joint probability distribution Θ of the variables decomposes as $P(A)P(B|\{A, E\})P(C|A)P(D)P(E|\{C, D\})P(F|C)$.

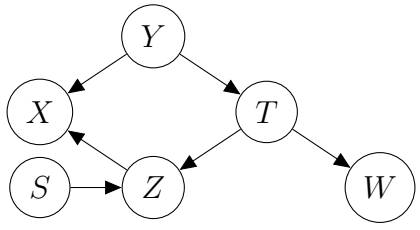


Figure 2.1: An example BN for a set of six random variables $\{A, B, C, D, E, F\}$.

$\mathbb{G}$ *entails* $I(X, Y|Z)$ if the CI holds for all the probability distributions Θ that satisfy the Markov condition with $\mathbb{G}$. Every such CI is identified by *d-separation*, which is defined

as follows. Two distinct nodes X and Y in $\mathbb{G}$ are said to be d-separated by a set of nodes $\mathcal{S} \subseteq \mathcal{X} \setminus \{X, Y\}$ if every *chain*, i.e., path between X and Y ignoring edge directions, has a node Z such that one of the following holds:

- $Z \in \mathcal{S}$ and the edges incident to Z meet head-to-tail ($\ldots \rightarrow Z \rightarrow \ldots$ or $\ldots \leftarrow Z \leftarrow \ldots$). For example, C in the chain $A \rightarrow C \rightarrow F$ in the BN shown in Figure 2.1.

- $Z \in \mathcal{S}$ and the edges incident to Z meet tail-to-tail ($\ldots \leftarrow Z \rightarrow \ldots$). For example, C in the chain $E \leftarrow C \rightarrow F$ in the example BN.

- Neither Z nor any of its descendants are in $\mathcal{S}$ and the edges incident to Z meet head-to-head ($\ldots \rightarrow Z \leftarrow \ldots$). For example, the chain $A \rightarrow B \leftarrow E$ in the example BN is d-separated by $\mathcal{S} = \emptyset$ in the example BN because $\mathcal{S}$ does not contain B or any of its descendants.

$(\mathbb{G}, \Theta)$ is said to satisfy the *faithfulness condition* if $\mathbb{G}$ entails all and only the CI in Θ. All the DAGs that entail the same CI form a *Markov equivalence class* and can be represented using a completed partially directed acyclic graph (CPDAG), i.e., a DAG in which some edges are undirected. We make the faithfulness assumption in the subsequent discussion in this dissertation.

A *Markov blanket (MB)* of $X \in \mathcal{X}$ is a set of variables $\mathcal{B}(X)$ which completely d-separate X from all the other variables in $\mathcal{X}$, i.e., $I(X, Y|\mathcal{B}(X)))\ \forall Y \in \mathcal{X} \setminus (\mathcal{B}(X) \cup \{X\})$. A *Markov boundary* of X, denoted by $\mathcal{MB}(X)$, is any minimal MB of X. If $(\mathbb{G}, \Theta)$ satisfy the faithfulness condition, then the set consisting of parents, children, and parents of children of X make up its unique Markov boundary, i.e.,

$$\mathcal{MB}(X) = \mathcal{PC}(X) \cup \left(\bigcup_{X \in \mathcal{CH}(X)} \mathcal{PA}(X) \right)$$

The terms Markov blanket and Markov boundary are used interchangeably in the literature to denote the set $\mathcal{MB}(X)$ as defined above. In the example BN shown in Figure 2.1,

$\mathcal{MB}(C)$ is $\mathcal{PC}(C) \cup \{D\} = \{A, D, E, F\}$. Note that under the faithfulness assumption, $\mathcal{MB}$ implies a symmetric relation similar to $\mathcal{PC}$, i.e., $X \in \mathcal{MB}(Y) \iff Y \in \mathcal{MB}(X)$.

2.1.2 Sequential Learning of Bayesian Networks

Given a data set of m observations sampled from a joint probability distribution for a set of random variables, represented by D, getting the BN for the variables requires learning both its structure as well as the parameters of the conditional probability distributions (CPDs) of all the variables for the corresponding structure, i.e., both components of the pair $(\mathbb{G}, \Theta)$ need to be learned. However, once the structure has been learned estimating the corresponding CPD parameters is comparatively straightforward. Therefore, most of the research in the area has focused on learning the structure of BNs.

Exactly learning the BN structure has been shown to be an NP-hard problem and therefore a wide range of heuristic methods have been developed for this purpose. These methods are broadly classified into *score-based* and *constraint-based* methods. We briefly discuss the two classes of methods below.

Score-based Methods

Score-based methods aim to find the most likely BN structure given the observed data set. For the purpose, these methods use a scoring function which assigns a score to every possible structure that corresponds to the log of its posterior probability given the data set, i.e., $\textit{Score}(\mathbb{G} : \mathsf{D}) = \log P(\mathbb{G}|\mathsf{D})$. Then, the best-scoring BN structure can be picked using the following equation

$$\mathbb{G} = \arg\max_{\mathbb{H}} \ \textit{Score}(\mathbb{H} : \mathsf{D}) = \arg\max_{\mathbb{H}} \ \log P(\mathbb{H}|\mathsf{D})$$

Using Bayes' law, $P(\mathbb{H}|\mathsf{D})$ can be written as $\dfrac{P(\mathsf{D}|\mathbb{H})P(\mathbb{H})}{P(\mathsf{D})}$, where $P(\mathsf{D})$ is the prior probability of the data set and $P(\mathbb{H})$ is the prior probability of the structure. Since $P(\mathsf{D})$

does not depend on the structure, the optimal BN structure can be found by maximizing $\log P(\mathsf{D}|\mathbb{H}) + \log P(\mathbb{H})$. The prior probability of the structure and can either be provided as input or determined using one of the various approaches proposed for the purpose. We refer the reader to [32] for a survey of these approaches. For simplicity of discussion, we assume uniform $P(\mathbb{H})$ for every $\mathbb{H}$ and ignore the corresponding term in the score. Under this assumption, the score of a structure is given by the log of posterior probability of data set given the structure, i.e., $\mathit{Score}(\mathbb{G} : \mathsf{D}) = \log P(\mathsf{D}|\mathbb{G})$.

The number of possible DAGs for a given set of random variables is super-exponential in the number of variables. Therefore, an exhaustive search over the space of all the DAGs is not tractable for more than a few variables. Some of the scoring functions are *additive*, i.e., the global score can be computed from variable-specific scores as

$$\mathit{Score}(\mathbb{G} : \mathsf{D}) = \sum_{X \in \mathcal{X}} \mathit{score}(X, \mathcal{R}(X) : \mathsf{D})$$

where $\mathit{score}(X, \mathcal{R}(X), \mathsf{D})$ evaluates the choice of the parent set $\mathcal{R}(X)$ for the variable X. Even using additive scoring functions, finding the globally optimal DAG still requires exponential run-time. Therefore, *score-based* methods employ heuristic algorithms which use greedy hill-climbing [33], greedy search over score-equivalent network structures [34], etc. and sampling algorithms which use Gibbs sampling [35], Markov chain Monte Carlo [36], etc. for learning BN structure in practice.

Constraint-based Methods

Constraint-based methods learn the structure of a BN in two steps. First, using repeated applications of CI tests, the undirected edges of the BN are learned. The CI tests are conducted using statistical tests on the observation data, which we discuss in further detail in subsection 2.1.3. The learned structure with only undirected edges is referred to as the BN *skeleton*. Then, the edges of the skeleton are directed using the rules of d-

separation [37] to obtain a CPDAG representing the Markov equivalence class of DAGs that entail the same CIs. Since real-world BNs are sparse, orienting the edges takes minuscule time as compared to the first step. Therefore, we focus on the different approaches for learning the skeleton in this dissertation.

The *constraint-based* approaches for learning BN skeleton can be classified as either *global-search* or *local-to-global*. *Global-search* methods start with a fully connected skeleton, i.e., one with edges between all pairs of variables. Then, they progressively eliminate edges between variables which are found to be independent given a subset of the other variables. The most notable examples of algorithms in this category are the algorithm by *Peter and Clark* (*PC*) [38] and the order-independent variant of the algorithm by *Peter and Clark* (*PC-stable*) [39]. *Local-to-global* methods, on the other hand, first learn the local neighborhood of each variable and then combine these local neighborhoods to obtain the global structure. The approaches in this category can be further classified based on the methodology used for learning the local neighborhoods. *Blanket learning* approaches first reduce the neighborhood search space for every variable by learning their MBs. Subsequently, the subset of the variables in the MBs which are also part of the respective immediate neighborhoods, i.e., parents and children, are learned. *Grow-Shrink* (*GS*) [40] was the first to follow this approach. Multiple other algorithms using this approach have been proposed since then, e.g., *Incremental Association MB* (*IAMB*) [41], *Interleaved IAMB* (*Inter-IAMB*) [41], etc. Conversely, *direct learning* approaches learn the immediate neighbors of every variable without any intermediate steps. These include *Max-Min Parents and Children* (*MMPC*) [41, 42], the *HITON Parents and Children* (*HITON-PC*) algorithm [43], *Semi-Interleaved HITON Parents and Children* (*SI-HITON*) [44], *Get Parents and Children* (*GetPC*) [45], etc. We discuss sequential *global-search constraint-based* methods and *local-to-global constraint-based* methods in more detail in section 4.1 and subsection 3.1.3, respectively.

2.1.3 Testing Conditional Independence and Measuring Association

Given a set of observations for the variables, tests for determining CI can be conducted using different statistical tests for discrete and continuous variables. For discrete nominal (categorical) variables, conducting tests of CI involves calculating a statistic which is asymptotically distributed as Chi-squared (χ^2), with appropriate degrees of freedom. The variables are declared independent if the strength of association between them is not deemed significant as discussed below.

Chi-squared Distribution based Tests

The sum of squares of f samples from a standard normal distribution (i.e., a distribution with a mean of 0 and a variance of 1) are distributed as a χ^2 distribution with f degrees of freedom. The cumulative probability that the sum will be equal to or greater than a particular value can be calculated from the distribution. Therefore, tests based on the χ^2 distributions are used to test the significance of relationship between nominal variables, e.g., *Pearson's chi-squared test*, *G-test*, *Fisher's exact test*, etc. G-test has been preferred for the task of BN learning in the literature [46, 47, 42] therefore we will focus on it in this section.

G-test requires computing the G^2 statistic which is defined as follows:

$$G^2 = 2 \sum_i O_i \ln \left(\frac{O_i}{E_i} \right)$$

where $O_i \geq 0$ is the observed frequency of a configuration, $E_i > 0$ is the expected frequency of the configuration assuming the null hypothesis, and the sum is computed over all the different configurations of the variables. Under the null hypothesis, that the variables are independent, the G^2 statistic is asymptotically distributed as χ^2 with appropriate degrees of freedom.

G-test for Constraint-Based Learning

Let S_i^a be a random variable whose value is the number of observations in the data set where $X_i = a$, S_{ij}^{ab} be a random variable whose value is the number of observations in the data set where $X_i = a$ and $X_j = b$, and S_{ijk}^{abc} be a random variable whose value is the number of observations in the data set where $X_i = a$, $X_j = b$, and $X_k = c$. Under the null hypothesis, i.e., $I(X_i, X_j|X_k)$, the expected number of observations can be calculated as

$$E(S_{ijk}^{abc}|S_{ik}^{ac} = s_{ik}^{ac}, S_{jk}^{bc} = s_{jk}^{bc}) = \frac{s_{ik}^{ac} s_{jk}^{bc}}{s_k^c}$$

and the G^2 statistic can be computed as

$$G^2 = 2 \sum_{c \in X_k} \sum_{a \in X_i} \sum_{b \in X_j} s_{ijk}^{abc} \ln \left(\frac{s_{ijk}^{abc} s_k^c}{s_{ik}^{ac} s_{jk}^{bc}} \right) \tag{2.1}$$

The corresponding degree of freedom f is computed as $(r_i - 1) \times (r_j - 1) \times r_k$, where r_i is the arity or the size of the domain of X_i, etc. In the more general case, when testing $I(X_i, X_j|\mathcal{S})$ where $\mathcal{S} \subset \mathcal{X}$, f is computed as $(r_i - 1) \times (r_j - 1) \times \prod_{X_k \in \mathcal{S}} r_k$. The computation of G^2 statistic requires counting the number of rows that match every configuration of the variables in the conditioning set. Specifically, for a conditioning set $\mathcal{S}$, $O(r^{|\mathcal{S}|})$ counts are needed, where $r = \max_{X_i \in \mathcal{X}} r_i$.

The *p-value* of the G-test is computed as the probability that the G^2 statistic was drawn from the χ^2 distribution with f degrees of freedom. If the *p-value* is lower than a significance threshold, denoted by α (typically 0.01 or 0.05 [47]), the null hypothesis is rejected and $\neg I(X_i, X_j|X_k)$ is determined to be true. Lower *p-value* indicates stronger dependence and therefore the additive inverse of *p-value* is used for quantifying the strength of association between the variables, i.e., *Assoc*$(X_i, X_j|\mathcal{S})$.

Computing the G^2 Statistic

In order to compute the G^2 statistic, the number of observations corresponding to each combination of $X_i = a$, $X_j = b$, and $X_k = c$ in Equation 2.1 need to be counted. In BN structure learning implementations, two types of approaches have been used to compute these counts. The first and more common approach is to compute the counts when they are required. A trivial solution for the purpose counts the frequency corresponding to every configuration as and when it is required. This requires $O(m|\mathcal{S}|r^{|\mathcal{S}|})$ time as the whole data set will have to be traversed for every configuration. A faster solution creates a *contingency table* for the required counts and then traverses the data set once to fill the table. This reduces the time required for the computation to $O(m|\mathcal{S}| + r^{|\mathcal{S}|})$ for an increased space requirement of $O(r^{|\mathcal{S}|})$. Other approaches in this category which use advanced strategies based on bit maps and radix sort have also been developed [48, 49].

The other category of approaches pre-process the data set and create an index data structure, e.g., a hash table or an *ADtree* [50], which can be used to retrieve the counts in almost constant time during learning. For instance, once the *ADtree* has been constructed, computing the counts requires $O(r^{|\mathcal{S}|})$ time which is independent of the data set size. However, these approaches require significant pre-processing time which can not be amortized by the corresponding gains during the learning of sparse networks [48]. Thus, we focus on the approaches in the former category in this dissertation.

2.2 Module Networks

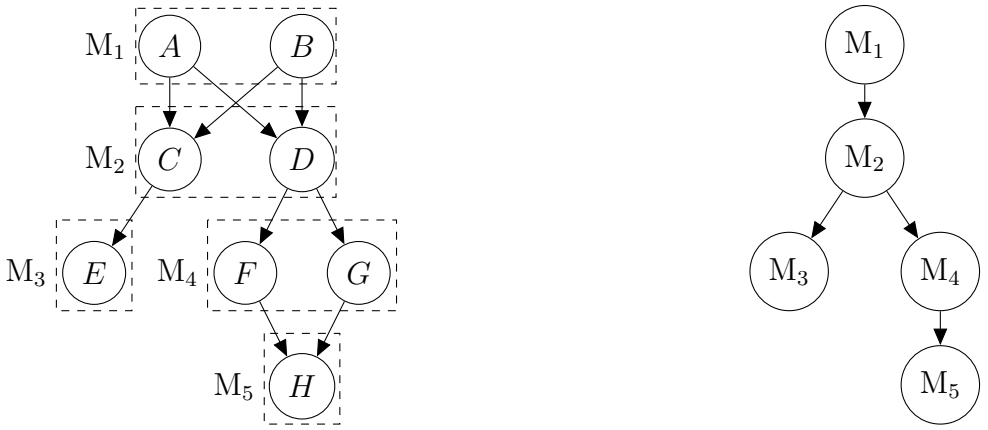
BNs use a DAG to represent the joint probability distribution of a set of random variables and thereby provide a compact model for reasoning about interactions in multi-dimensional entities. The capability of the BN framework to reason about uncertainty has led to their successful use in many different fields [51, 14, 52]. 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 [53, 54], and (b) confidence in learned BN models is low when the data set does not have sufficient number of observations [55]. Specialization of BNs that rely on variations of *parameter-sharing* have been proposed to overcome these limitations [56].

Introduced by Segal et al., MoNets [57, 58] 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 [59], probabilistic relational models [54], hierarchical BNs [60], 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 [61], cancer genomics [62, 63, 64, 65], construction of cellular networks [66, 67, 68], and integration of multi-omics data [69, 70, 71]. MoNets and other parameter-sharing specializations of BN have also found applications in diverse fields, e.g., medical diagnosis [72], stock market analysis [58], traffic modeling [73], active learning using serious games [74], feature selection and feature extraction [75], computational psychology [76, 77], and data mining [78].

2.2.1 Notations and Definitions

MoNets partition the variables into *modules*, where a module is 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 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



(a) Bayesian network (b) Module network

Figure 2.2: An example BN for a set of eight random variables $\{A, B, C, D, E, F, G, H\}$ and the corresponding MoNet.

parent variables, represented by $\mathcal{PA}(\mathrm{M}_j)$, where $\mathcal{PA}(\mathrm{M}_j) \subset \mathcal{X}$. 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 $X \in \mathcal{PA}(\mathrm{M}_k)$.

Figure 2.2a 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 2.2b. Note that, the parents of a module should be a parent for all the variables in the module, e.g., $B \in \mathcal{PA}(\mathrm{M}_2) \iff B \in \mathcal{PA}(C)$ and $B \in \mathcal{PA}(D)$ in the figure. However, the variables in a module may have different sets of descendants, e.g., $\mathcal{CH}(C) \neq \mathcal{CH}(D)$.

2.2.2 Sequential Construction of Module Networks

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 [21]. 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.

Various *score-based* heuristic methods have been proposed for constructing MoNets from observed data [79, 58, 80, 81, 71]. 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 described in subsection 2.1.2, heuristics are used in MoNet learning to traverse the space of all possible MoNets to 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 CPDs for the modules as part of their structure learning routine. The most popular software packages for MoNet learning are *GENOMICA* [58] and *Lemon-Tree* [71]. In both these software, the learned CPDs are represented using regression trees [82].

GENOMICA implements the iterative two-step algorithm proposed by Segal et al. [57, 58] to construct MoNets. During the first step, each variable is considered one at a time and is assigned to the module whose regression tree best predicts its observations as determined by the score, and the parameters of all the module regression trees are re-learned. During the second step, while keeping the module assignments fixed, the parent sets are learned for each module and a regression tree is learned for the parent sets. The two steps are repeated until the score of the network does not improve.

Lemon-Tree, on the other hand, implements the approach outlined by Bonnet et al. [71] that separates the learning of module assignments and parents and CPDs into three distinct steps which are executed consecutively. In the first step, multiple variable clusters are obtained using multiple runs of a Gibbs sampler method that constructs two-way clusters by partitioning data along both the dimensions, i.e., variables and observations [83]. From

these multiple clusters, a consensus variable clustering solution is obtained in the second step using a spectral clustering method proposed in [84]. The consensus variable clusters so obtained are considered as modules of the variables in the third step. Then, regression tree structure is learned for every module followed by the learning of the parent sets for all the modules [71]. We describe the *Lemon-Tree* algorithm in more detail in section 5.1.

2.3 Parallel Computing

There exist a plethora of important problems which are compute-intensive. The scale of such problems that can be tackled using sequential approaches is ultimately limited by the capabilities of a single computation *core*. Parallel computing utilizes multiple cores operating in concert to reduce the time to solution for such problems and enables solutions for larger problem sizes. However, most sequential algorithms can not be directly used for computation using multiple cores. Therefore, novel parallel algorithms need to be designed for the purpose. Multiple models of parallel computation have been developed for designing parallel algorithms, e.g., shared memory, networked distributed memory, task-based, etc.

We use the networked distributed memory model for designing the algorithms presented in this dissertation and explain it in subsection 2.3.1. Then, we define the goals of parallel algorithm design using multiple performance measures in subsection 2.3.2. Finally, we briefly discuss the implementation of parallel algorithms in subsection 2.3.3.

2.3.1 Networked Distributed Memory Model

Each computation core (or *processor*) in this model has its own local memory, which can be accessed by the core in the same time as in sequential computations. The processors are assumed to be connected to each other through a network, referred to as the *interconnection* network, and can communicate with each other by sending and receiving messages over the network. A processor in the model can only communicate with one other processor at a

time. We further assume that the interconnection network can route messages between all the possible pairs of processors concurrently. Consequently, different pairs of processors are assumed to be able to communicate at the same time.

The cost of computation of parallel algorithms designed for this model is estimated similar to the methodology traditionally used for sequential algorithms, i.e., using asymptotic analysis for the purpose and assuming that both unit computation and single local memory access take $O(1)$ time. The communication cost of the algorithms is computed by assuming that transferring a message of m bytes over the network takes $\tau + \mu m$ time, where τ represents the *latency* (in time) and μ represents the inverse *bandwidth* (in time per byte) of the network. Since τ and μ are typically orders of magnitude higher than the unit computation time, the communication time of the parallel algorithms is estimated separately from the computation times. For example, a widely used parallel algorithm for computing prefix sums of n elements on p processors can be estimated to take $O(\frac{n}{p} + \log p)$ time for computation and $O((\tau + \mu) \log p)$ time for communication [85].

2.3.2 Measuring the Performance of Parallel Algorithms

We use $T_{\text{seq}}(n)$ to denote the run-time of the best sequential algorithm for a problem of size n. The run-time of the parallel algorithm for the same problem size on p processors is denoted using $T(n, p)$. The *strong scaling* performance of a parallel algorithm gauges its scalability for a fixed problem size as the parallelism is increased, i.e., fixed n as p is increased. Therefore, *strong scaling speedup* and *strong scaling efficiency* (%) of a parallel algorithm are defined, respectively, as

$$S_{\text{strong}}(n, p) = \frac{T_{\text{seq}}(n)}{T(n, p)} \text{ and } E_{\text{strong}}(n, p) = \frac{T_{\text{seq}}(n)}{p \cdot T(n, p)} \times 100\% \tag{2.2}$$

In cases where sequential execution is infeasible, we also refer to *relative speedup* and *relative efficiency* (%) of the parallel algorithm on p_2 processors relative to p_1 processors

($p_2 \geq p_1$), defined as

$$S_{\text{rel}}(n, p_2) = \frac{T(n, p_1)}{T(n, p_2)} \text{ and } E_{\text{rel}}(n, p_2) = \frac{p_1 \cdot T(n, p_1)}{p_2 \cdot T(n, p_2)} \times 100\% \tag{2.3}$$

The *weak scaling* of a parallel algorithm, on the other hand, is a measure of how it scales when the parallelism is increased while keeping the problem size per processor fixed, i.e., the algorithm is evaluated by choosing n_p on p processors such that $\frac{T_{\text{seq}}(n_p)}{T_{\text{seq}}(n_1)} \approx p$, where n_1 is the problem size used for the sequential experiments. Accordingly, *weak scaling speedup* and *weak scaling efficiency* (%) are defined as

$$S_{\text{weak}}(n, p) = \frac{T_{\text{seq}}(n_1)}{p \cdot T(n_p, p)} \text{ and } E_{\text{weak}}(n, p) = \frac{T_{\text{seq}}(n_1)}{T(n_p, p)} \times 100\% \tag{2.4}$$

Since the parallel algorithms must do at least as much computational work as the corresponding best sequential algorithms, i.e., $T_{\text{seq}}(n) \leq p \cdot T(n, p)$, the following inequalities hold for all parallel algorithms and both types of scaling defined above (except in the case of *super-linear speedup*)

$$S(n, p) \leq p \text{ and } E(n, p) \leq 100\%$$

Sometimes, the run-time of a parallel algorithm is compared with the run-time of the same algorithm on $p = 1$ processor, instead of the best sequential algorithm. The measures of performance so obtained are referred to as *self speedup* and *self efficiency* and are larger than their regular counterparts. For example, strong scaling self speedup defined as below will be greater than strong scaling speedup since $T(n, 1) \geq T_{\text{seq}}(n)$.

$$S_{\text{self}}(n, p) = \frac{T(n, 1)}{T(n, p)} \tag{2.5}$$

The goal of parallel algorithm design in this dissertation is to get the performance measures defined above as close to their respective limits as possible. For the presented parallel algorithms, we estimate these measures analytically using the asymptotic complexity of the algorithms as well as measure them experimentally. Our primary focus is on strong scaling of the presented algorithms. Therefore, we use the terms speedup and efficiency to reference the corresponding strong scaling versions. Further, when the problem size and the type of scaling is clear from the context, we use cleaner notations, e.g., T_{seq}, T_p, S_p, E_p, etc.

2.3.3 Implementing Parallel Algorithms

The parallel algorithms designed for the networked distributed memory model can be implemented using the API described by the *Message Passing Interface (MPI)* standard. The standard defines basic functionality for sending and receiving messages between pairs of processors. Additionally, the standard contains API definitions for common collective operations involving all or subsets of processors, e.g., broadcast, reduce, all-to-all, prefix-scan, etc. We have implemented the algorithms presented in this dissertation using a *C++* interface for MPI [86] and assume knowledge of common MPI operations when explaining the algorithms. We refer the reader to [87] for a primer on the standard.

The design of parallel algorithms assumes perfect balance of load during the execution of the algorithm, i.e., all the processors are assumed to be allocated similar amounts of work and each unit of work is assumed to require the same amount of time. However, these assumptions may be violated in practice because of the following two reasons: i) the work allocated to the processors may change disparately as the algorithm progresses, and ii) each unit of work may end up requiring vastly different times for computations. This may significantly deteriorate the practical scalability of a theoretically efficient algorithm because a few processors may end up doing considerably more work than the average work required by all the processors, thus increasing the total time taken by the parallel algorithm.

Therefore, we quantify the severity of load imbalance by computing the deviation of the maximum load on any processor from the average load on all the processors as

$$\text{Load Imbalance} = \frac{\max_{0 \leq j < p} \text{Load on } j^{\text{th}} \text{ processor}}{\left(\frac{\sum_{0 \leq j < p} \text{Load on } j^{\text{th}} \text{ processor}}{p} \right)} - 1 \tag{2.6}$$

In this dissertation, we use the above equation to balance the work assigned to every processor a priori and also for post hoc analysis of the performance of our parallel algorithms.

2.4 Related Work

2.4.1 Parallel Learning of Bayesian Networks

Previous works on developing parallel methods for BN learning have primarily focused on either *score-based* or *global-search constraint-based* methods.

Score-based Methods

Exact *score-based* algorithms with exponential run-time complexity have been proposed to find the optimal structure for small BNs, i.e., BNs with less than 20 variables [88, 89]. Even parallelization of these exact solutions can only construct networks with a maximum of 37 variables [90, 91, 92]. Compared to exact methods, parallelization of heuristic *score-based* methods has yielded results with much better scalability. Nikolova et al. [93] developed a parallel method that can construct a network with 500 variables in 107 seconds using 1024 cores. Misra et al. [49] developed a similar approach that can construct a $15,216$ variable BN in less than 172 seconds using 1.57 million cores of the *Tianhe-2* supercomputer.

Global-search Constraint-based Methods

Multiple previous works have proposed parallelization of *global-search constraint-based* algorithms. Madsen et al. [94] proposed two different algorithms for parallelizing *PC*. The first algorithm is designed for shared-memory model utilizing *balanced incomplete block* designs [95] to assign statistic computations for CI tests to different threads. However, as noted by the authors, this approach only works for conducting marginal CI tests and an additional heuristic is employed to reduce the amount of work in every subsequent iteration that conducts more CI tests than the sequential algorithm. Further, the conditioning set size is limited to 3 in this algorithm. Using this approach, they report a maximum speedup of almost 7 using 12 threads for constructing a network with $2,371$ variables. Their second algorithm follows a similar approach for parallelizing CI tests with a distributed-memory model and achieves a maximum speedup of about 8 using 10 cores for the same data set.

Since Colombo et al. addressed the issue of order-dependence in *PC* by proposing *PC-stable* [39], it has become the algorithm of choice amongst the *global-search constraint-based* methods. Correspondingly, the parallelization efforts in recent years have also been focused on *PC-stable*. Le et al. proposed the first parallel algorithm for the purpose, called *parallel-PC* [96]. *Parallel-PC* conducts all the CI tests for a given conditioning set size in parallel and then synchronizes the results at the end of every iteration. Their implementation achieves a maximum speedup of 12 using 14 cores for learning a network with $2,810$ variables. Scutari et al. [27] implemented a parallel version of *PC-stable* in *bnlearn* using a similar method as *parallel-PC*. Schmidt et al. [97] attempted to improve the scalability of *parallel-PC* by implementing a dynamic load-balancing scheme using master-worker paradigm. They implemented a shared-memory approach and report a maximum speedup of 39X using 80 threads.

Schmidt et al. [98] proposed the first GPU-based approach for accelerating *PC-stable*. They later improved this method using an out-of-core algorithm [99]. However, both these works have limited real-world applicability because they restrict the maximum condition-

ing set size to 1. Zarebavani et al. [100] proposed a more general methodology using GPUs that works for any conditioning set size and are able to learn networks with up to $4,000$ variables from synthetic data sets. More recently, Hagedorn and Huegle [101] reported being able to learn networks from synthetic data sets with $8,000$ variables using GPUs.

Local-to-global Constraint-based Methods

One of the earliest attempts at parallelizing *local-to-global constraint-based* algorithms was by Aliferis et al. [44] that developed a sequential framework for multiple *local-to-global constraint-based* algorithms and proposed an extension to the framework for parallel and distributed learning [102]. Their method first distributes the variables to available processors and then learns neighborhood for the target variable from only the variables local to the processor. The processor-specific neighborhoods are then combined to get the final neighborhood for the target variable. This strategy suffers from the following two drawbacks: i) the final neighborhood depends on the number of processors and the distribution of variables, and ii) the total work may increase when running in parallel thereby resulting in a slowdown similar to the one observed in the reported results.

Nikolova and Aluru [103] focused on parallelizing two *direct learning* algorithms – *MMPC* and *GetPC* – and reported near perfect scaling for learning neighborhoods of $1,000$ variables on up to 512 cores. However, as the authors observed, their approach does not scale when the number of variables or the number of observations are increased. This is because their approach assigns all the computations for determining the local neighborhood of a variable to the same processor. Due to the differences in the computation requirements across variable neighborhoods, such a static assignment of variables to processors leads to load imbalance.

Among the software for BN structure learning that we surveyed, *bnlearn* is the only one that supports learning BNs on multiple cores using the *constraint-based* algorithms that we focus on. It uses the *parallel* library of the core *R* distribution for parallelizing the structure

learning using a master-worker paradigm on the specified cores [27]. Since *bnlearn* is the only other available parallel implementation of the algorithms under consideration, we evaluate its performance as a baseline for our methods in chapter 3 and chapter 4.

2.4.2 Construction of Module Networks

As discussed in subsection 2.2.2, *GENOMICA* implements the iterative two-step algorithm proposed by Segal et al. [57, 58] to construct MoNets while *Lemon-Tree* implements the approach outlined by Bonnet et al. [71]. 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 [79, 80, 104]. 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 [71], cholangiocarcinoma [105], breast cancer [65], penile cancer [106], and rheumatoid arthritis [104]. 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 [107] and feed efficiency [108] of cattle, analysis of early stage development of European sea bass [109], and identification of genes critical for tomato ripening [110] and apple edibility [111].

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 [112, 110]. 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. [113] parallelized the MoNet learning method using a distributed-memory approach. They report a speedup of up to

29.26X using a maximum of 32 cores. Jiang et al. [114] 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*.

CHAPTER 3

PARALLELIZING LOCAL-TO-GLOBAL CONSTRAINT-BASED ALGORITHMS

In this chapter, we focus on parallelizing *local-to-global constraint-based* algorithms that rely on the discovery of variable neighborhoods as an intermediate step. Towards this end, we present a parallel framework to scale *constraint-based* BN structure learning algorithms to tens of thousands of variables. We demonstrate the applicability of our framework by parallelizing five different algorithms: *GS*, *IAMB*, *Inter-IAMB*, *MMPC*, and *SI-HITON*. Our implementations are able to construct BNs from real data sets with tens of thousands of variables and thousands of observations in less than 38 seconds on 2048 cores, with a speedup of up to $1,745$X and 85.2% efficiency. Furthermore, we demonstrate using simulated data sets that our proposed parallel framework can scale to learning BN structure of even higher dimensionality.

This chapter is organized as follows. First, we develop our proposed parallel framework in section 3.1 and use it to propose efficient parallel versions of the algorithms in section 3.2. Then, we discuss multiple optimizations in the implementations of these algorithms in section 3.3. Finally, we present the results of our experiments in section 3.4 and summarize the work in section 3.5. This chapter extends the following published paper on parallelizing *blanket learning* algorithms:

- A. Srivastava, S. Chockalingam, and S. Aluru, "A Parallel Framework for Constraint-based Bayesian Network Learning via Markov Blanket Discovery," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society, 2020, pp. 74–88

3.1 Proposed Parallel Framework

We propose a parallel framework which enables users to develop and implement an efficient parallel version of any *local-to-global constraint-based* strategy for constructing BNs. First, we state the key assumptions in subsection 3.1.1. Then, we introduce the primary data structures used by our framework in subsection 3.1.2. Subsequently, in subsection 3.1.3, we present the sequential version of the *local-to-global* algorithms and identify their key components. Finally, we develop our parallel framework by proposing parallel algorithms for the identified algorithmic components in subsection 3.1.4.

3.1.1 Assumptions

Similar to the other *constraint-based* algorithms, we assume an ordering of the input variables in $\mathcal{X}$, i.e., $X_1 < X_2 < \ldots < X_n$. We also assume, similar to the previous parallel algorithms, that the input data set D with m observations for n variables is available locally on all the processors. For the run-time complexity computations, we assume that the time required for conducting CI tests and computing $Assoc(\cdot)$ values with conditioning sets of size k is bounded by $O(G_k)$. We use l to represent the maximum number of elements in the candidate neighborhood of any variable, i.e., $l = \max_{X \in \mathcal{X}} |\mathcal{LN}(X)|$ at any stage of algorithm execution. Since, in the worst case, all the other variables may be added to the candidate neighborhood of a variable, l is bounded by $O(n)$. We also assume the networked distributed memory model, described in subsection 2.3.1, to develop the proposed parallel algorithms.

3.1.2 Key Data Structures

The primary data structure that we use in our framework is a list of tuples, referred to as $c\text{-}scores$. Elements of $c\text{-}scores$ are of the form $\langle X, Y, \theta_{XY} \rangle$, where X and Y are variables and θ_{XY} is a numeric value. At any point during the execution of the algorithms, if

$\langle X, Y, \theta_{XY} \rangle$ is an element in $c\text{-}scores$, then the variable Y is a potential candidate for addition to the local neighborhood of the target variable X, i.e., $\mathcal{LN}(X)$. The third element, θ_{XY}, is the score for adding Y to $\mathcal{LN}(X)$ and is used to select the best candidate for every target variable. Apart from the $c\text{-}scores$ list, we also maintain a list, denoted as $variables$, that contains all the variables for which the neighborhood sets are to be computed.

In order to construct the BN skeleton, *local-to-global* algorithms need to identify the local neighborhoods for all the variables. Accordingly, we initialize the $variables$ list with all the variables in $\mathcal{X}$. Since neighborhood discovery generally starts with empty sets, $\mathcal{LN}(T)$ is initialized to $\phi \; \forall T \in variables$. We initialize the $c\text{-}scores$ list with a tuple each for all the possible candidates for all the variables in $\mathcal{X}$ and set all the scores to zero, i.e., the list is initialized with elements from the set $\{\langle X, Y, 0 \rangle | X \in \mathcal{X}, Y \in \mathcal{X} \setminus (\{X\} \cup \mathcal{LN}(X))\}$. At the beginning of the algorithm, there is a tuple in $c\text{-}scores$ corresponding to each of the $n^2 - n$ ordered variable pairs. Furthermore, the tuples in the $c\text{-}scores$ list are initialized in the ascending order of the first variable and then of the second variable. Therefore, all the tuples with the candidate variables corresponding to the same target variable are arranged in a contiguous manner in the list.

When executing the algorithms on p processors, the $c\text{-}scores$ list is initialized in a similar fashion but is block distributed among all the processors. The corresponding list on the processor j is denoted by $c\text{-}scores_j$ and its size is bounded by $\lceil \frac{n^2-n}{p} \rceil$. The list $variables_j$ is initialized with all the variables for which the processor j computes the neighborhoods, i.e., it includes all the elements from the set $\{X | \langle X, Y, \theta_{XY} \rangle \in c\text{-}scores_j\}$. Since the $c\text{-}scores$ list is ordered such that tuples with the same first variable are contiguous, the size of $variables_j$ is bounded by $O(\frac{n}{p})$. In the distributed setting, $\mathcal{LN}(T)$ is initialized on every processor for all $T \in variables_j$. Note that, for two different processors i and j and some variable T, both $variables_i$ and $variables_j$ may contain T. In such cases, both processors i and j compute $\mathcal{LN}(T)$.

3.1.3 Sequential Algorithmic Components

We first identify the key components used by *local-to-global* algorithms for sequential learning of BNs. The execution of the algorithms of interest, in general, can be separated into the following four phases – *Grow*, *Shrink*, *Symmetry Correction*, and *Construct PC from MB*. These phases are utilized for learning the local neighborhood for a variable T ($\mathcal{LN}(T)$) as follows:

- In a *Grow* phase, the candidate neighborhood set for T is grown by adding one variable to the set from among the available candidates.
- During a *Shrink* phase, one or more false positive variables are removed from the candidate neighborhood set.
- *Symmetry Correction* is performed, after identifying candidate neighborhood sets for all the variables in one or more iterations of *Grow* and *Shrink* phases, to obtain symmetrically consistent $\mathcal{LN}$ sets.
- *Construct PC from MB* is used only by *blanket learning* algorithms for learning the skeleton of the BN using the MBs for all the variables. A variable X in $\mathcal{MB}(T)$ is included in $\mathcal{PC}(T)$ if no subset of $\mathcal{MB}(T) \setminus \{X\}$ (or $\mathcal{MB}(X) \setminus \{T\}$) can render X and T conditionally independent.

We now describe in detail the execution of *blanket learning* algorithms and the *direct learning* algorithms in terms of the $c\text{-}scores$ and $variables$ lists, defined and initialized as per subsection 3.1.2.

Blanket Learning

In a *Grow* phase iteration, scores are first updated for all the tuples in the current $c\text{-}scores$ list. For all the *blanket learning* algorithms, we use the associativity of a candidate Y with

Algorithm 1: Sequential *Grow* and *Shrink* phases for *GS*

```
function GROW-SHRINK-GS():
    Input: D, initial MB(·) sets, T
    Output: Updated MB(T) set
    N ← X \ (MB(T) ∪ {T})
    repeat
        Z ← first Y ∈ N s.t. ¬I(T, Y | MB(T), D)
        if such a Z exists then
            MB(T) ← MB(T) ∪ {Z}
            N ← N \ {Z}
    until MB(T) does not change
    for Z ∈ MB(T) do
        if I(T, Z | MB(T) \ {Z}, D) then
            MB(T) ← MB(T) \ {Z}
```

the target T given the current MB of the T as the score of the candidate, i.e.,

$$\theta_{TY} = \mathit{Assoc}(T, Y|\mathcal{MB}(T)) \tag{3.1}$$

Then, using the updated scores, a candidate is selected for every variable. More specifically, the *Grow* phase in *IAMB* and *Inter-IAMB* picks the candidate with the maximum score, i.e., the tuple $\langle T, Z, \theta_{TZ}\rangle$ is picked for T if

$$\langle T, Z, \theta_{TZ}\rangle = \underset{\langle T,Y,\theta_{TY}\rangle \in c\text{-}scores}{\arg\max} \theta_{TY} \tag{3.2}$$

GS, on the other hand, picks for a variable T the first candidate that shows dependency with T. As mentioned in subsubsection 2.1.3, we use the additive inverse of *p-value* of the G^2 test $I(T, X|\mathcal{MB}(T))$ as $\mathit{Assoc}(T, Y|\mathcal{MB}(T))$. Therefore, candidate selection for *GS* can be accomplished by identifying the first tuple with a score greater than the additive inverse of the significance threshold ($-\alpha$), i.e., a tuple $\langle T, Z, \theta_{TZ}\rangle$ is selected for T if

$$\langle T, Z, \theta_{TZ}\rangle \text{ is first entry in } c\text{-}scores \text{ s.t. } \theta_{TY} \geq -\alpha \tag{3.3}$$

In both the cases, if such a tuple is found, then Z is added to $\mathcal{MB}(T)$ and $\langle T, Z, \theta_{TZ} \rangle$ is removed from the $c\text{-}scores$ list.

Algorithm 2: Sequential *Grow* and *Shrink* phases for *IAMB*

```
1  function GROW-SHRINK-IAMB():
       Input: D, initial MB(·) sets, T
       Output: Updated MB(T) set
2      N ← X \ (MB(T) ∪ {T})
3      repeat
4          Z ← arg max_{Y∈N} Assoc(T, Y|MB(T), D)
5          if ¬I(T, Y|MB(T), D) then
6              MB(T) ← MB(T) ∪ {Z}
7              N ← N \ {Z}
8      until MB(T) does not change
9      for Z ∈ MB(T) do
10         if I(T, Z|MB(T) \ {Z}, D) then
11             MB(T) ← MB(T) \ {Z}
```

During the *Shrink* phase, all the $\mathcal{MB}$ sets are examined and a variable X is removed from $\mathcal{MB}(T)$ if $I(T, X|\mathcal{MB}(T) \setminus \{X\})$ holds. *Blanket learning* algorithms differ on how *Grow* and *Shrink* phases are iterated. Both *GS* and *IAMB* execute multiple iterations of *Grow* phase followed by a single *Shrink* phase, whereas *Inter-IAMB* alternates between *Grow* and *Shrink* phases until all the $\mathcal{MB}$ sets stop changing. The sequential *Grow* and *Shrink* phases for *GS*, *IAMB*, and *Inter-IAMB* are shown in algorithm 1, algorithm 2, and algorithm 3.

After the one or more iterations of *Grow* and *Shrink* phases, MB construction proceeds to *Symmetry Correction*, in which $T \in \mathcal{MB}(Y) \iff Y \in \mathcal{MB}(T)$ is verified and when this assertion fails for a pair (T, Y), the offending variables are removed from the respective $\mathcal{MB}$ sets. Finally, the edges of the BN skeleton are learned in form of $\mathcal{PC}$ sets by verifying CI for every subset of $\mathcal{MB}$.

Time Complexity: Each *Grow* phase iteration updates scores for $O(n^2)$ target-candidate variable pairs in $O(n^2 G_l)$ time. Each *Shrink* phase checks the candidate $\mathcal{MB}$ for all the

Algorithm 3: Sequential *Grow* and *Shrink* phases for *Inter-IAMB*

```
function GROW-SHRINK-INTERIAMB():
    Input: D, initial MB(·) sets, T
    Output: Updated MB(T) set
    N ← X \ (MB(T) ∪ {T})
    repeat
        Z ← arg max_{Y∈N} Assoc(T, Y|MB(T), D)
        if ¬I(T, Y|MB(T), D) then
            MB(T) ← MB(T) ∪ {Z}
            N ← N \ {Z}
        for Z ∈ MB(T) do
            if I(T, Z|MB(T) \ {Z}, D) then
                MB(T) ← MB(T) \ {Z}
                N ← N ∪ {Z}
    until MB(T) does not change
```

variables, which requires $n \times O(lG_l)$ time. Therefore, the time taken by *Grow* phase dominates the run-time complexity of every iteration for all three algorithms. Since the algorithms execute *Grow* phase $O(l)$ times, the run-time for getting the blankets for all the variables is $O(n^2 l G_l)$. *Symmetry Correction* can be performed in $O(nl)$ time. Then, getting the skeleton using *Construct PC from MB* requires $O(nl2^l G_l)$ time. Therefore, the sequential run-time complexity of the three *blanket learning* algorithms is

$$O(nl(n + 2^l)G_l) \tag{3.4}$$

Direct Learning

Direct learning algorithms score a candidate Y for the target variable T using the minimum associativity of Y with T given any subset of the current direct neighbors of T. Therefore, for both *MMPC* and *SI-HITON*, θ_{TY} is computed as

$$\theta_{TY} = \min_{\mathcal{S} \subseteq \mathcal{PC}(T)} \textit{Assoc}(T, Y|\mathcal{S}) \tag{3.5}$$

Algorithm 4: Sequential *Grow* and *Shrink* phases for *MMPC*

function GROW-SHRINK-MMPC():
  Input: D, initial $\mathcal{PC}(\cdot)$ sets, T
  Output: Updated $\mathcal{PC}(T)$ set
  $\mathcal{N} \leftarrow \mathcal{X} \setminus (\mathcal{PC}(T) \cup \{T\})$
  repeat
    $\mathit{Sep}(T) \leftarrow \emptyset, \forall T \in \mathcal{X}$
    for $Y \in \mathcal{N}$ **do**
      $\mathit{Sep}(Y) \leftarrow \arg\min_{\mathcal{S} \subseteq \mathcal{PC}(T)} \mathit{Assoc}(T, Y|\mathcal{S}, \mathsf{D})$
    $Z \leftarrow \arg\max_{Y \in \mathcal{N}} \mathit{Assoc}(T, Y|\mathit{Sep}(Y), \mathsf{D})$
    if $\neg I(T, Z|\mathit{Sep}(Z), \mathsf{D})$ **then**
      $\mathcal{PC}(T) \leftarrow \mathcal{PC}(T) \cup \{Z\}$
    $\mathcal{N} \leftarrow \mathcal{N} \setminus \{Z\}$
  until $\mathcal{PC}(T)$ *does not change*
  for $Z \in \mathcal{PC}(T)$ **do**
    if $I(T, Z|\mathcal{S}, \mathsf{D})$ *for some* $\mathcal{S} \subseteq \mathcal{PC}(T) \setminus \{Z\}$ **then**
      $\mathcal{PC}(T) \leftarrow \mathcal{PC}(T) \setminus \{Z\}$

Then, *direct learning* algorithms proceed similar to *blanket learning* algorithms. In each *Grow* iteration, the $c\text{-}scores$ list is updated first and then a candidate is selected for addition to $\mathcal{PC}$ set of every variable. *MMPC* picks the candidate with the maximum score for every variable using Equation 3.2 while *SI-HITON* picks the first variable which is dependent on T given the current $\mathcal{PC}(T)$ using Equation 3.3. Unlike the other *local-to-global* algorithms, though, *SI-HITON* always considers candidates in the order of their marginal associativity with the target. In order to accomplish this, the $c\text{-}scores$ list is sorted after the scores are updated in the first *Grow* phase iteration.

In each *Shrink* phase, false positives are cleared from all the $\mathcal{PC}$ sets by removing every variable X in $\mathcal{PC}(T)$ that is found to be independent of T given any subset of $\mathcal{PC}(T) \setminus \{X\}$. Both *MMPC* and *SI-HITON* execute multiple *Grow* phase iterations followed by a *Shrink* phase execution at the end. Then, the skeleton is obtained in the form of the consistent $\mathcal{PC}$ sets after *Symmetry Correction* is performed. The sequential *Grow* and *Shrink* phases for *MMPC* and *SI-HITON* are shown in algorithm 4 and algorithm 5.

Algorithm 5: Sequential *Grow* and *Shrink* phases for *SI-HITON*

```
1  function GROW-SHRINK-SIHITON():
       Input: D, initial PC(·) sets, T
       Output: Updated PC(T) set
2      N ← X \ (PC(T) ∪ {T})
3      while N ≠ ∅ do
4          Z ← arg max_{Y∈N} Assoc(T, Y|∅, D)
5          if ¬I(T, Z|PC(T), D) then
6              PC(T) ← PC(T) ∪ {Z}
7          N ← N \ {Z}
8      for Z ∈ PC(T) do
9          if I(T, Z|S, D) for some S ⊆ PC(T) \ {Z} then
10             PC(T) ← PC(T) \ {Z}
```

Time Complexity: The score updates in each iteration of *Grow* phase computes *Assoc* for all the $O(n^2)$ pairs using subsets of the candidate $\mathcal{PC}$ sets of size $O(l)$ in $O(n^2 2^l G_l)$ time. Then, *Shrink* phase conducts CI tests for all the elements in each of the n candidate $\mathcal{PC}$ sets and takes $O(nl2^l G_l)$ time. *Symmetry Correction* can be accomplished in $O(nl)$ time. Therefore, for *MMPC*, sequentially getting the skeleton requires $O(l) \times O(n^2 2^l G_l) + O(nl2^l G_l) + O(nl)$ time which is dominated by the total time required by the *Grow* phase:

$$O(n^2 l 2^l G_l) \tag{3.6}$$

As discussed above, *SI-HITON* additionally requires a sort in the first *Grow* phase iteration which takes $O(n^2 \log n^2)$ time. Therefore, the time complexity of *SI-HITON* is

$$O(n^2(l2^l G_l + \log n)) \tag{3.7}$$

3.1.4 Parallel Framework Components

We now discuss the key components of our proposed framework – parallel algorithms for all the four phases described in subsection 3.1.3. We designed these parallel components using common parallel primitives such as *all-reduce*, *scan*, shift permutations, and parallel

sort.

Grow Phase

Our parallel algorithm for *Grow* phase is based on the following two key insights: (i) The neighborhood sets for all the variables are required for constructing the skeleton. Further, for addition to the neighborhood set of a variable, all the other variable are considered a candidate. Therefore, we consider all the variable pairs in parallel, using the distributed $c\text{-}scores$ list. (ii) The time taken in conducting a CI test (or computing $Assoc(\cdot)$) is proportional to the size of the conditioning set. Therefore, we designed this component such that the CI tests (and $Assoc(\cdot)$ computations) with the same conditioning set sizes are conducted in parallel.

The pseudo-code for our parallel *Grow* phase is shown in algorithm 6. The implementation of *Grow* phase for the algorithms of interest differ in two major aspects. First, while the *blanket learning* algorithms use Equation 3.1 for computing the scores and $I(T, Y|\mathcal{MB}(T)\setminus\{Y\})$ for checking if Y is independent of T given the current $\mathcal{MB}(T)$ set, the *direct learning* algorithms use Equation 3.5 and check $I(T, Y|\mathcal{S})\forall\mathcal{S} \subseteq \mathcal{PC}(T) \setminus \{Y\}$ for the purpose, respectively. In order to accommodate these differences, our proposed *Grow* phase algorithmic component accepts two functions as arguments: COMPUTE-SCORE and CHECK-CI. These functions are implemented separately for *blanket learning* and *direct learning* algorithms, as described earlier. Second, as discussed in subsection 3.1.3, *local-to-global* algorithms use different heuristics to select the next variable to be added to the current candidate set. Therefore, our proposed *Grow* algorithmic component requires two more functions as arguments: APPLY-HEURISTIC and REDUCE-HEURISTIC. The function APPLY-HEURISTIC accepts a slice of the $c\text{-}scores$ list corresponding to a variable T such that it contains $\langle T, X, \theta_{TX}\rangle$ for all the candidates X, and returns the candidate most suitable for addition to $\mathcal{LN}(T)$. For example, the APPLY-HEURISTIC selects a candidate as per Equation 3.2 for *IAMB*, *Inter-IAMB*, and *MMPC* and as per Equation 3.3 for *GS* and

Algorithm 6: Parallel *Grow* Phase

```
1  function GROW-PHASE():
       Input: D, c-scores, variables, current 𝓛𝓝(·) sets, scoreOrder
2              COMPUTE-SCORE, CHECK-CI,
3              APPLY-HEURISTIC, REDUCE-HEURISTIC
       Output: Updated 𝓛𝓝(·) sets
4      parallel j = processor's rank do
5          for ⟨T, Y, θ_TY⟩ ∈ c-scores_j do
6              θ_TY ← COMPUTE-SCORE(T, Y, 𝓛𝓝(T), D)
7              Update ⟨T, Y, θ_TY⟩ in c-scores_j
8          if scoreOrder then
9              Parallel sort c-scores, first by T then by θ_TY
10         g-select_j(T) ← nil, ∀T ∈ variables_j
11         for T ∈ variables_j do
12             ts ← ⟨T, X, θ_TX⟩ ∈ c-scores_j, ∀X ∈ 𝒳
13             g-select_j(T) ← APPLY-HEURISTIC(ts)
14         REDUCE-HEURISTIC(c-scores, g-select)
15         for T ∈ variables_j do
16             Z ← g-select_j(T)
17             if ¬CHECK-CI(T, Z, 𝓛𝓝(T), D) then
18                 𝓛𝓝(T) ← 𝓛𝓝(T) ∪ {Z}
19                 Remove ⟨T, Z, θ_TZ⟩ from c-scores_j
```

SI-HITON. The REDUCE-HEURISTIC function accumulates the variable selection results from all the processors to identify for each variable T, the best candidate to be added to its $\mathcal{LN}(T)$.

In algorithm 6, the local $c\text{-}scores$ list is updated with the computed *Assoc* values first (line 5–line 7), which takes $O(\frac{n^2}{p}G_l)$ time for *blanket learning* algorithms and $O(\frac{n^2}{p}2^lG_l)$ time for *direct learning* algorithms. To modify the ordering of the candidate variables for all the target variables, as required by *SI-HITON*, $c\text{-}scores$ list is sorted if $scoreOrder$ is true (line 8–line 9). Parallel sorting can be accomplished by any comparison based sort such as parallel bitonic sort, which takes $O(\frac{n^2}{p}\log\frac{n^2}{p} + \frac{n^2}{p}\log^2 p)$ and $O(\tau\log^2 p + \mu\frac{n^2}{p}\log^2 p)$ time for computation and communication, respectively. The selection heuristic is then applied for each variable (line 11–line 13) followed by the accumulation of results across

processors (line 14). The run-time of these operations depends on the heuristic used by the specific algorithm. For the selection heuristics described in Equation 3.2 and Equation 3.3, two segmented parallel *scan* operations are sufficient for accumulating the results from all the processors because the underlying operators are associative. Note that these parallel *scan* operations exploit the contiguous presence of all the tuples $\langle T, Y, \theta_{TY} \rangle$ corresponding to a target variable T in $c\text{-}scores$. Therefore, selection of candidate variables takes $O(\frac{n^2}{p} + \log p)$ time for computation and $O((\tau + \mu) \log p)$ time for communication. Finally, we add the selected variables to the $\mathcal{LN}$ sets and update $c\text{-}scores$ (line 15–line 19). Since $\mathcal{LN}$ sets updated are local to the processor, the number of $\mathcal{LN}$ sets updated on a processor is bounded by $O(\frac{n}{p})$.

Time Complexity: Each iteration of the *Grow* phase algorithmic component, in general, requires $O(\frac{n^2}{p}G_l) + O(\frac{n^2}{p} + \log p) = O(\frac{n^2}{p}G_l + \log p)$ computation time for the *blanket learning* algorithms and $O(\frac{n^2}{p}2^l G_l) + O(\frac{n^2}{p} + \log p) = O(\frac{n^2}{p}2^l G_l + \log p)$ for the *direct learning* algorithms. The only communication in this component is the collective communication for reducing the heuristic computations across the processors, which takes $O((\tau+\mu) \log p)$ time. However, if $sortOrder$ is true for ***SI-HITON***, then the component requires $O(\frac{n^2}{p}2^l G_l)+O(\frac{n^2}{p}\log\frac{n^2}{p}+\frac{n^2}{p}\log^2 p)+O(\frac{n^2}{p}+\log p) = O(\frac{n^2}{p}(2^l G_l+\log\frac{n^2}{p}+\log^2 p))$ computation time and $O(\tau \log^2 p + \mu\frac{n^2}{p}\log^2 p)$ communication time.

Algorithm 7: Parallel *Shrink* Phase

```
1  function SHRINK-PHASE():
       Input: D, variables, current LN(·) sets, CHECK-CI
       Output: Updated LN(·) sets
2      parallel j = processor's rank do
3          for T ∈ variables_j do
4              for Z ∈ LN(T) do
5                  if CHECK-CI(T, Z, LN(T) \ {Z}, D) then
6                      LN(T) ← LN(T) \ {Z}
```

Shrink Phase

Our proposed parallel component for *Shrink* phase is shown in algorithm 7. Here, the only task is to remove those variables in $\mathcal{LN}$ which are independent given the rest of the $\mathcal{LN}$ in case of *blanket learning* algorithms, or are independent given any subset of the $\mathcal{LN}$ in case of *direct learning* algorithms. This difference is again accommodated by accepting the function CHECK-CI as input. Then, the false positives are removed in a loop over all the $\mathcal{LN}$ sets for the target variables on the processor (line 3–line 6).

Time Complexity: The run-time for the parallel *Shrink* phase is proportional to the size of the $\mathcal{LN}$ sets for all the variables on the processor, which is bounded by $O(\frac{n}{p}) \times O(l) = O(\frac{nl}{p})$. Each call to CHECK-CI requires $O(G_l)$ time for the *blanket learning* algorithms and $O(2^l G_l)$ time for the *direct learning* algorithms leading to a corresponding total computation time of $O(\frac{nl}{p} G_l)$ and $O(\frac{nl}{p} 2^l G_l)$, respectively. This component requires no communications.

Symmetry Correction

The proposed parallel component for checking the symmetry of the $\mathcal{LN}$ sets, shown in algorithm 8, is based on the method developed by [103]. It proceeds by creating $sc\text{-}pairs$, a list of ordered tuples for every member of $\mathcal{LN}$ sets (line 3–line 10) followed by parallel sorting to identify the asymmetric $\mathcal{LN}$ members (line 11–line 12). The $\mathcal{LN}$ sets are then updated to reflect the symmetry correction (line 13–line 16). The time to construct $sc\text{-}pairs$, remove unique tuples, and update $\mathcal{LN}$ sets is bounded by $O(\frac{nl}{p})$. As discussed in subsubsection 3.1.4, this requires $O(\frac{nl}{p} \log \frac{nl}{p} + \frac{nl}{p} \log^2 p)$ computation and $O(\tau \log^2 p + \mu \frac{nl}{p} \log^2 p)$ communication time. Collective communication is also required during the removal of the unique tuples to identify tuple pairs that cross processor boundary. This is accomplished by a pair of shift permutations that take $O(\tau + \mu)$ time.

Algorithm 8: Parallel *Symmetry Correction*

```
1  function SYMMETRY-CORRECTION():
       Input: variables, asymmetric ℒ𝒩(·) sets
       Output: Symmetry corrected ℒ𝒩(·) sets
2      parallel j = processor's rank do
3          sc-pairs_j ← empty list of variable pairs
4          for X ∈ variables_j do
5              if j = 0 or X ∉ variables_{j−1} then
6                  for Y ∈ ℒ𝒩(X) do
7                      if X < Y then
8                          Insert ⟨X, Y⟩ into sc-pairs_j
9                      else
10                         Insert ⟨Y, X⟩ into sc-pairs_j
11         Parallel sort sc-pairs, first by X then by Y
12         Remove all unique ⟨X, Y⟩ from sc-pairs
13         Reset all ℒ𝒩(·) sets to ∅
14         for ⟨X, Y⟩ ∈ sc-pairs_j do
15             ℒ𝒩(X) ← ℒ𝒩(X) ∪ {Y}
16             ℒ𝒩(Y) ← ℒ𝒩(Y) ∪ {X}
```

Time Complexity: The run-time of this component is dominated by the run-time of parallel sort which takes $O(\frac{nl}{p}\log\frac{nl}{p}+\frac{nl}{p}\log^2 p)$ computation time and $O(\tau\log^2 p+\mu\frac{nl}{p}\log^2 p)$ communication time.

Construct PC from MB

Our parallel algorithm to construct the skeleton of the BN from the $\mathcal{MB}$ sets in the *blanket learning* algorithms, is shown in algorithm 9. This component tries to identify a conditioning set for each element Y in $\mathcal{MB}(X)$, that can render X conditionally independent from Y (line 3–line 11). If no such conditioning set can be identified for the pair (X, Y), then Y is added to $\mathcal{PC}(X)$ (line 11). Note that this component requires the $\mathcal{MB}$ sets of both X and Y and therefore the complete $\mathcal{MB}$ sets should be made available on all the processors before CONSTRUCT-PC() is called.

Algorithm 9: Parallel *Construct PC from MB*

```
1  function CONSTRUCT-PC():
       Input: D, variables, complete MB(·) sets
       Output: Distributed PC(·) sets
2      parallel j = processor's rank do
3          for X ∈ variables_j do
4              PC(X) ← ∅
5              for Y ∈ MB(X) do
6                  if |MB(X)| < |MB(Y)| then
7                      B ← MB(X) \ {Y}
8                  else
9                      B ← MB(Y) \ {X}
10             if ¬I(X, Y|S, D)∀S ⊆ B then
11                 PC(X) ← PC(X) ∪ {Y}
12     return PC
```

Time Complexity: This component conducts CI tests for all the subsets of $O(l)$ elements in the $\mathcal{MB}$ set of all the $O(n)$ variables. Therefore, its computation complexity is $O(\frac{nl}{p}2^l G_l)$. It requires no collective communications.

3.2 Our Parallel Algorithms

Using the parallel framework developed in section 3.1, many *local-to-global* algorithms can be implemented. Here, we present efficient parallel versions of three *blanket learning* algorithms – *GS*, *IAMB*, and *Inter-IAMB*, and two *direct learning* algorithms – *MMPC* and *SI-HITON*.

3.2.1 Blanket Learning

The three *blanket learning* algorithms – *GS*, *IAMB*, and *Inter-IAMB* – can be implemented using the parallel skeleton construction algorithm presented in algorithm 10. As discussed in subsection 3.1.3, the only distinction between the three algorithms is how the next variable is selected in the *Grow* phase and this difference can be abstracted using the APPLY-

HEURISTIC and REDUCE-HEURISTIC functions. The computation of candidate scores and checking for conditional independence for all the *blanket learning* algorithms can be accomplished using the functions COMPUTE-SCORE and CHECK-CI implemented as discussed in subsubsection 3.1.4.

Algorithm 10: Parallel Construct Skeleton – *Blanket Learning* Algorithms

```
1  function CONSTRUCT-SKELETON-BLANKET():
       Input: algorithm, D, COMPUTE-SCORE, CHECK-CI,
2             APPLY-HEURISTIC, REDUCE-HEURISTIC
       Output: PC(T) sets for all T ∈ X
3      parallel j = processor's rank do
4          Initialize c-scores_j, variables_j, MB(·) as described in subsection 3.1.2
5          repeat
6              GROW-PHASE(D, c-scores_j, variables_j, MB, false,
                COMPUTE-SCORE, CHECK-CI, APPLY-HEURISTIC,
                REDUCE-HEURISTIC)
7              if algorithm is Inter-IAMB then
8                  SHRINK-PHASE(D, variables_j, MB)
9          until no MB changes on any of the processors
10         if algorithm is GS or IAMB then
11             SHRINK-PHASE(D, variables_j, MB)
12         SYMMETRY-CORRECTION(variables_j, MB)
13         Synchronize MB(·) across all the processors
14         PC ← CONSTRUCT-PC(D, variables_j, MB)
```

Given the four algorithm-specific functions, the proposed parallel versions of these algorithms proceed as follows. The requisite distributed lists and variables are initialized first (line 4), following which these algorithms execute the GROW-PHASE in a loop until convergence (line 5–line 9). While SHRINK-PHASE is called for *Inter-IAMB* after every call to GROW-PHASE (line 8), it is called only once at the end for the other two algorithms (line 11). Then, after SYMMETRY-CORRECTION (line 12), there is a synchronization step for collecting the $\mathcal{MB}(\cdot)$ for all the variables on all the processors (line 13). Finally, CONSTRUCT-PC is called to get the skeleton for the BN, in the form of $\mathcal{PC}$ sets for all the variables (line 14).

Time Complexity: The computational run-time complexity of algorithm 10 is computed by summing up the run-times of the four components, assuming that GROW-PHASE as well as SHRINK-PHASE are called $O(l)$ times, to get the following equation

$$O\left(\frac{nl}{p}(n+2^l)G_l + \frac{nl}{p}\log^2 p + l\log p\right)$$

If $p = O(n)$, then the above equation can be further simplified by noticing that $\log^2 n = O(n)$ as

$$O\left(\frac{nl}{p}(n+2^l)G_l\right) \tag{3.8}$$

Apart from the communication costs incurred by the four components, algorithm 10 also requires collective communications for (i) identifying if any of the $\mathcal{MB}$ sets changed during a *Grow* iteration, and (ii) synchronization of the $\mathcal{MB}$ sets. Using a bit set representation of the $\mathcal{MB}$ sets, both of these operations can be performed using *all-reduce*, which takes $O((\tau + \mu\log n)\log p)$ time. Hence, the communication run-time of this algorithm is

$$O\left(\tau\log p\,(\log p + l) + \mu l\log p\left(\frac{n\log p + p\log n}{p}\right)\right)$$

Again, if $p = O(n)$, the above equation can be further simplified by observing that $p\log n = O(n\log p)$ as

$$O\left(\tau\log p\,(\log p + l) + \mu\frac{nl}{p}\log^2 p\right) \tag{3.9}$$

Parallel Efficiency: Any parallelization strategy needs to have high strong scaling efficiency (computed using Equation 2.2) in order to be scalable, i.e., the difference between $p \times T(n,p)$ and $T_{\text{seq}}(n)$ should be asymptotically negligible. For *blanket learning* algorithms, the asymptotic parallel efficiency can be computed by substituting Equation 3.4 for

$T_{\text{seq}}(n)$ and the sum of Equation 3.8 and Equation 3.9 for $T(n,p)$ as

$$E(n,p) = \frac{nl(n+2^l)G_l}{p \times \left(\left[\frac{nl}{p}(n+2^l)G_l \right] + \left[\tau \log p \left(\log p + l \right) + \mu \frac{nl}{p} \log^2 p \right] \right)} \times 100\%$$
$$= \frac{nl(n+2^l)G_l}{nl(n+2^l)G_l + \tau p \log p \left(\log p + l \right) + \mu nl \log^2 p} \times 100\%$$

Since $\log p + l = O(\log p \times l)$, the denominator in the above equation can be bounded by $nl(n+2^l)G_l + l \log^2 p(\tau p + \mu n)$ which can further be bounded by $nl(n+2^l)G_l + nl \log^2 p(\tau + \mu)$ for $p = O(n)$. Therefore, our proposed parallel versions of the *blanket learning* algorithms are efficient if $(\tau+\mu) \log^2 p = O((n+2^l)G_l)$. Accordingly, we get the following two bounds on the number of processors that can be used by the *blanket learning* algorithms while being efficient

$$p = O(n) \text{ and } p = O\left(2^k\right), \text{ where } k = \sqrt{\frac{(n+2^l)G_l}{(\tau+\mu)}} \tag{3.10}$$

3.2.2 Direct Learning

Our parallel algorithm for constructing skeletons using the two *direct learning* algorithms – *MMPC* and *SI-HITON* – is shown in algorithm 11. The functions COMPUTE-SCORE and CHECK-CI used for computing the scores and checking CI by *direct learning* algorithms, implemented as per subsubsection 3.1.4, are provided as inputs. Further, the different schemes for picking the variable in *Grow* phase by the two algorithms are also provided as input using APPLY-HEURISTIC and REDUCE-HEURISTIC. Given the inputs, algorithm 11 proceeds in a similar manner as algorithm 10 for the *blanket learning* algorithms, except for two differences described below.

The first difference between algorithm 10 and algorithm 11 is that, as discussed in

Algorithm 11: Parallel Construct Skeleton – *Direct Learning* Algorithms

```
1  function CONSTRUCT-SKELETON-DIRECT():
       Input: algorithm, D, COMPUTE-SCORE, CHECK-CI,
2             APPLY-HEURISTIC, REDUCE-HEURISTIC
       Output: PC(T) sets for all T ∈ X
3      parallel j = processor's rank do
4          Initialize c-scores_j, variables_j, PC(·) as described in subsection 3.1.2
5          sortOrder ← false
6          if algorithm is SI-HITON then
7              sortOrder ← true
8          repeat
9              GROW-PHASE(D, c-scores_j, variables_j, PC, sortOrder,
                COMPUTE-SCORE, CHECK-CI, APPLY-HEURISTIC,
                REDUCE-HEURISTIC)
10             sortOrder ← false
11         until no PC changes on any of the processors
12         SHRINK-PHASE(D, variables_j, PC)
13         SYMMETRY-CORRECTION(variables_j, PC)
```

subsubsection 12, *SI-HITON* considers variables in the descending order of their marginal associativity with the target variable. Correspondingly, we designed our GROW-PHASE proposed in algorithm 6 to sort the $c\text{-}scores$ list after score updates, if required. We utilize this by setting $sortOrder$ to true for *SI-HITON* before the first call to GROW-PHASE (line 9) and then set it to false after the call. This ensures that the $c\text{-}scores$ list is sorted only once after it has been updated with the marginal associativity to get the requisite ordering of the candidate variables. The second difference is that *direct learning* algorithms get the correct $\mathcal{PC}$ sets after the call to SYMMETRY-CORRECTION and, therefore, do not call CONSTRUCT-PC.

Time Complexity: During the execution of both *MMPC* and *SI-HITON*, GROW-PHASE is called $O(l)$ times and SHRINK-PHASE is called just once at the end. However, *SI-HITON* also requires a sort step in the first call to GROW-PHASE. Therefore, again assuming that $p = O(n)$ and noticing that $\log^2 n = O(n)$, the computational run-time complexity of

MMPC is

$$O\left(\frac{n^2}{p} l 2^l G_l\right) \tag{3.11}$$

and that of *SI-HITON* is

$$O\left(\frac{n^2}{p}\left(l 2^l G_l + \log\frac{n^2}{p} + \log^2 p\right)\right) \tag{3.12}$$

In addition to the communication required by the parallel components used, algorithm 11 also requires collective communications for identifying if any of the $\mathcal{PC}$ sets changed during a *Grow* iteration. As discussed for algorithm 10, this can be accomplished in $O((\tau + \mu \log n) \log p)$ time. Therefore, the communication run-time of *MMPC* is same as that of *blanket learning* algorithms (Equation 3.9). However, due to the extra sort, the communication run-time of *SI-HITON* increases to

$$O\left(\tau \log p \left(\log p + l\right) + \mu\frac{n^2}{p}\log^2 p\right) \tag{3.13}$$

Parallel Efficiency: The strong scaling efficiency of *MMPC* can be computed by substituting Equation 3.6 for $T_{\text{seq}}(n)$ and the sum of Equation 3.11 and Equation 3.4 for $T(n,p)$:

$$\begin{aligned} E(n,p) &= \frac{n^2 l 2^l G_l}{p \times \left(\left\lceil \frac{n^2}{p} l 2^l G_l \right\rceil + \left\lceil \tau \log p \left(\log p + l\right) + \mu\frac{nl}{p}\log^2 p \right\rceil\right)} \times 100\% \\ &= \frac{n^2 l 2^l G_l}{n^2 l 2^l G_l + \tau p \log p \left(\log p + l\right) + \mu n l \log^2 p} \times 100\% \end{aligned}$$

We notice again, as we did for the parallel efficiency of the *blanket learning* algorithms in subsection 3.2.1, that *MMPC* is efficient if $(\tau + \mu)\log^2 p = O(n 2^l G_l)$. Therefore, the

number of processors that can be used by *MMPC* while being efficient is

$$p = O(n) \text{ and } p = O\left(2^k\right), \text{ where } k = \sqrt{\frac{n2^l G_l}{(\tau + \mu)}} \tag{3.14}$$

Similarly, the strong scaling efficiency of *SI-HITON* can be computed using Equation 3.7 as the numerator and the sum of Equation 3.12 and Equation 3.13 as the denominator

$$\begin{aligned} E(n,p) &= \frac{n^2(l2^l G_l + \log n)}{p \times \left(\left[\frac{n^2}{p}\left(l2^l G_l + \log\frac{n^2}{p} + \log^2 p\right)\right] + \left[\tau \log p\,(\log p + l) + \mu\frac{n^2}{p}\log^2 p\right]\right)} \times 100\% \\ &= \frac{n^2(l2^l G_l + \log n)}{n^2\left(l2^l G_l + \log\frac{n^2}{p}\right) + \tau p \log p\,(\log p + l) + \mu n^2 \log^2 p} \times 100\% \end{aligned}$$

As before, *SI-HITON* is efficient if $(\tau + \mu)\log^2 p = O(l2^l G_l + \log\frac{n^2}{p})$. Since $\log\frac{n^2}{p} = \Omega(\log n)$ for $p = O(n)$, we can get a tighter bound on p by solving $(\tau + \mu)\log^2 p = O(l2^l G_l + \log n)$. Therefore, the number of processors for which *SI-HITON* is efficient is bounded by

$$p = O(n) \text{ and } p = O\left(2^k\right), \text{ where } k = \sqrt{\frac{l2^l G_l + \log n}{(\tau + \mu)}} \tag{3.15}$$

3.3 Implementation

We implemented our framework using *C++* and MPI conforming to the *C++14* and *MPI 3.1* standards, respectively. Our implementations of the algorithms are available part of an open-source software [30].

3.3.1 Sequential Implementation

bnlearn [24] is a popular *R* package that supports a wide range of *score-based* and *constraint-based* algorithms for learning BNs, including the five algorithms that we focus on. The package has been used in multiple recent studies for the construction and analysis of BNs [115, 116, 117, 118]. Even though the top-level logic for most of the algorithms supported by *bnlearn* is implemented using *R*, the computationally intensive tasks such as the computations for conducting the CI tests are implemented in *C*. Hence, in spite of interfacing with an interpreted language, *bnlearn* is able to achieve performance comparable to that of a compiled language.

Our implementations differ from that of *bnlearn* because of the ambiguity in the specification of the *GS* algorithm and the choice of internal data structures. For efficiency purposes, we used different data structures than the ones used by *bnlearn* for some of the underlying tasks. For example, *bnlearn* uses arrays for storing the indices of the variables in a set. But, we use bit sets which enables us to use SIMD instructions for some set operations and also reduce the message sizes during communication. This modification, however, may alter the order in which the variables are considered by the algorithms in some cases. Since CI testing using real data sets is imperfect and any errors in the CI tests may change the behavior of the *constraint-based* algorithms, the BNs learned by such algorithms are known to be sensitive to the ordering of the variables [46, 119, 39]. In order to ensure that our choices for efficiency do not affect the accuracy of the learned network, we validate our implementations against *bnlearn* in subsubsection 3.4.2. Our experiments show that these choices help us achieve considerable speedup over *bnlearn* without significantly impacting the learned network structure.

3.3.2 Statistic Computation Strategies

Prior studies have estimated that more than 90% of the time in *constraint-based* learning is spent in aggregating counts from observation data for the CI tests [48]. Correspondingly,

we observed that computing the G^2 statistic took between 94% and 99% of the total run-time for learning the network sequentially in our experiments in section 3.4. Therefore, both efficiently conducting the CI tests as well as reducing the number of CI tests are essential for good run-time performance of learning algorithms in practice.

Counting Strategies

In order to conduct the CI test $I(X, Y|\mathcal{S})$, using the G^2 statistic (Equation 2.1), the counts of the number of observations $s_{abc}s_{ac}, s_{bc},$ and s_c corresponding to each combination of $X = a$, $Y = b$, and $\mathcal{S} = c$ are required. In BN structure learning implementations, two types of approaches have been used to compute these counts. The most common approach is to compute the counts when they are required. A trivial solution for the purpose counts the frequency corresponding to every configuration by traversing the complete data set as and when it is required and takes $O(m|\mathcal{S}|r^{|\mathcal{S}|})$ time, where r is the maximum arity of any variable in the data set. The time required for the purpose can be reduced to $O(m|\mathcal{S}| + r^{|\mathcal{S}|})$ by scanning the complete data set to fill up contingency tables of size $O(r^{|\mathcal{S}|})$. Advanced strategies that are based on bit maps and radix sort have also been developed for the purpose [48, 49]. An alternate approach is to pre-process the data set and create an index data structure, e.g., a hash table or an *ADtree* [50], which can be used to retrieve the counts in $O(r^{|\mathcal{S}|})$ time during learning. As discussed by Karanet al. [48], the latter category of approaches require significant pre-processing time which can not be amortized by the corresponding gains during the learning of sparse networks. Thus, we focused on the approaches in the former category and implemented the contingency table based approach as well as the two other strategies from the *SABNAtk* library [48]. We observed that the contingency table based approach outperformed the other two approaches for the data sets that we experimented with. Consequently, we report the run-times using the contingency table based approach, also used by *bnlearn*, in section 3.4. Nevertheless, our framework can be easily extended to use other counting strategies.

Algorithm-specific Optimizations

We reduced the number of *Assoc*$(\cdot)$ computations in our implementations of the *direct learning* algorithms and *GS* by utilizing some algorithm-specific observations as described below. We discuss the effect of these optimizations on the sequential run-time of the algorithms in subsubsection 3.4.2.

Reusing Scores in *Direct Learning* Algorithms: We optimized the score computations in *direct learning* algorithms by noticing that they use all subsets of $\mathcal{PC}$ sets for the purpose, as shown in Equation 3.5. Since both *MMPC* and *SI-HITON* only call SHRINK-PHASE after all the GROW-PHASE calls, the elements of $\mathcal{PC}$ set for any target variable always increase by one variable between two subsequent calls to GROW-PHASE. Therefore, these algorithms can reuse the score from the previous iteration to compute the new score as follows. Consider the $\mathcal{PC}$ set of a target variable T with X chosen to be added to $\mathcal{PC}(T)$ in one call to GROW-PHASE. Then, in the next call to GROW-PHASE, to update the score θ_{TY} for a candidate Y for addition to $\mathcal{PC}(T)$, we only need to consider the subsets which contain the last entrant of $\mathcal{PC}(T)$, i.e., θ_{TY} need only be updated if $\theta_{TY} > \min \mathit{Assoc}(T, Y|\mathcal{S})$ where $\mathcal{S} \subseteq \mathcal{PC}(T)$ such that $\mathcal{S}$ contains X. Therefore, this optimization reduces the number of calls to *Assoc* by reducing the number of subsets for which it is called in every update step.

Early Termination of Score Updates in *GS*: In the implementation of GROW-PHASE for *GS*, the update of the $c\text{-}scores$ list for a target variable X can be terminated as soon as the first score θ_{XY} which is greater than or equal to $-\alpha$ is computed. This is because the corresponding candidate Y will be the one picked by the algorithm for addition to $\mathcal{MB}(X)$ in that iteration, as per Equation 3.3. Notice that, this optimization is useful even in a parallel implementation, when the $c\text{-}scores$ list corresponding to a target X may be distributed across multiple processors. However, since the score updates happen concurrently on all

the processors, a processor j will stop the updates for X only after finding the first viable candidate for X in its local list $c\text{-}scores_j$. Therefore, if a suitable candidate for X exists on a processor $i < j$, then extra work is done on the processor j as compared to the sequential execution. We discuss the effect of this optimization on the scaling performance of *GS* in subsection 3.4.4. Even though *SI-HITON* uses the same candidate selection heuristic as *GS*, we do not implement this optimization for *SI-HITON* because it conflicts with the optimization for *direct learning* algorithms discussed above.

3.3.3 Load Balancing

Construction of a BN in parallel starts with a block distribution of the list of candidate tuples, $c\text{-}scores$, to all the processors. In every call to GROW-PHASE, one tuple is selected for every variable and removed from the $c\text{-}scores$ list. Furthermore, if $\mathcal{LN}$ for a variable stops changing, then all the candidate tuples corresponding to that variable are removed from the list as well. After a few iterations, these removals can lead to a disparity between the size of the $c\text{-}scores$ list across the processors. Since the time taken by a processor in an iteration of the GROW-PHASE is proportional to the size of the $c\text{-}scores$ list on that processor, the run-time of an iteration is determined by the the processor with the maximum number of tuples. This load imbalance between processors can, therefore, increase the total time required for learning the $\mathcal{LN}$ sets.

We mitigate the load imbalance problem by a stable block redistribution of the remaining candidate tuples at the end of an iteration. Specifically, we use an `MPI_Alltoallv` call to redistribute the remaining elements of the $c\text{-}scores$ list so that it is block distributed while maintaining the original order of the tuples. However, since redistribution is expensive and adds to the total run-time, we redistribute only if the imbalance is severe. For determining the severity, we compute the imbalance using Equation 2.6 by using the size of the list on a processor as the load on the processor.

In our implementation, redistribution is done if the computed imbalance is greater than

a user-specified threshold. We observed that setting this threshold to 0.2 resulted in optimal performance for every combination of data sets and number of processors in our experiments in section 3.4. Therefore, we use it as the default value for the threshold in our framework. We study the load imbalance and its effect on the total run-time further in subsection 3.4.3.

3.4 Experiments and Results

We performed our experiments on the Phoenix cluster at Georgia Tech [120]. Each node on the cluster has a 2.7 GHz 24-core Intel Xeon Gold 6226 processor and a minimum of 192 GB of main memory. The nodes run `RHEL 7.6` operating system and are connected via HDR100 (100 Gbps) InfiniBand. For the scalability experiments, we used a maximum of 86 nodes on this cluster. We compiled the source code, implemented with *C++14* and MPI, using `gcc v9.2.0` with `-O3 -march=native` optimization flags and `MVAPICH2 v2.3.3` implementation of MPI. We report the run-times measured by assigning 24 MPI processes per node and averaging the run-times over 5 different runs.

In our experiments, we observed that the first calls to the MPI *all-to-all* collectives took significantly longer time than the subsequent calls. Therefore, we warm up both `MPI_Alltoall` and `MPI_Alltoallv` by calling them with one byte on each processor. The time taken by the warm-up is negligible when using 64 processes or fewer. It then increases from 0.2 seconds on 128 processes to 5.7 seconds for 2048 processes and is not included in the reported run-times.

3.4.1 Data sets

To demonstrate performance and scalability of our parallel algorithms, we chose the construction of gene networks. In this application, the genes are modeled as random variables which correspond to the nodes of a BN and the edges of the BN correspond to the biological interactions between the genes. We used three real gene expression data sets of different

sizes, summarized in Table 3.1.

Table 3.1: Benchmark data sets used for experimenting with *local-to-global constraint-based* algorithms.

Name	Organism	Genes (n)	Observations (m)
D1	*S. cerevisiae*	$5,716$	$2,577$
D2	*A. thaliana*	$18,373$	$5,102$
D3	*A. thaliana*	$18,380$	$16,838$

D1 is a data set generated from the organism *Saccharomyces cerevisiae*, a species of yeast involved in baking and brewing. Tchourine et al. [121] created this data set of $2,577$ observations each for $5,716$ genes by combining data from multiple RNA-seq expression studies. The data sets *D2* and *D3* contain expression profiles for *Arabidopsis thaliana*, a model organism in plant biology with more than $23,000$ genes. These data sets are constructed by collecting over $18,000$ microarray images from public databases (ArrayExpress and GEO), and pre-processing them using standard microarray data analysis workflows for quality control and normalization. In order to study process-specific phenomena, it is necessary for plant biologists to consult multiple gene networks generated from many process-specific data sets. *D2* is a subset of *D3*, manually curated by a domain specialist and includes only those microarray experiments that were designed to study the development process in *A. thaliana*. *D2* and *D3* contain $5,102$ and $16,838$ observations for $18,373$ and $18,380$ genes, respectively. We used the method recommended by Friedman et al. [17] for discretizing the data sets.

In order to study the scalability of our implementations on data sets with larger number of variables, we generated three simulated data sets with $n = 30,000$ and $m = 10,000$ using the *pcalg* [26] software as follows. First, we construct three random DAGs with $30,000$ variables of increasing edge density by specifying edge addition probability of 5×10^{-5}, 1×10^{-4}, and 5×10^{-4}. Then, we use the dependency structure specified by the three DAGs to sample $10,000$ observations for all the variables. Finally, we discretize the

data sets as described above. We refer to the three simulated data sets so obtained as *S1*, *S2*, and *S3*, respectively. All the data sets are stored in plain text format on a GPFS storage which is accessible from all the nodes on the cluster. We used a significance threshold (α) of 0.05 for learning BNs in all our experiments.

3.4.2 Comparison with *bnlearn*

We used *bnlearn* v4.6.1 with *R* v4.0.3 for the experiments reported in this section.

Sequential Comparison

We compare the run-time of *bnlearn* with that of our optimized sequential implementation for learning the network using the five algorithms from the benchmark data sets in Table 3.2. The run-times for both *bnlearn* and our method are proportional to the size of

Table 3.2: Comparison of the time taken by *bnlearn* and our sequential implementations in constructing the BNs using the five *local-to-global constraint-based* algorithms for the benchmark data sets, measured in seconds, and the corresponding speedup. The symbol $\times$ indicates that the run did not finish in seven days.

Data set	Algorithm	Run-time (s)		Speedup
		bnlearn	*Ours*	
D1	*GS*	$13,525.9$	310.9	43.5
	IAMB	$1,347.7$	803.8	1.7
	Inter-IAMB	$1,356.6$	808.8	1.7
	MMPC	$7,446.9$	331.0	22.5
	SI-HITON	$5,854.4$	348.4	16.8
D2	*GS*	$546,196.1$	$9,076.3$	60.2
	IAMB	$52,370.7$	$18,999.7$	2.8
	Inter-IAMB	$52,725.6$	$18,976.8$	2.8
	MMPC	$406,884.2$	$6,789.0$	59.9
	SI-HITON	$317,838.0$	$6,923.2$	45.9
D3	*GS*	$\times$	$25,209.5$	N/A
	IAMB	$116,144.7$	$60,280.4$	1.9
	Inter-IAMB	$122,586.9$	$63,306.0$	1.9
	MMPC	$\times$	$32,131.6$	N/A
	SI-HITON	$527,522.2$	$35,341.9$	14.9

the data sets, with *D1* taking the shortest time and *D3* taking the longest. We also observed that the implementation of *bnlearn* for *GS* is almost an order of magnitude slower than that of the other two *blanket learning* algorithms. This is because *bnlearn* implements the variable selection for the algorithm using expensive loops in *R*. Consequently, our implementation of the algorithm is 43.5 – 60.2X faster than *bnlearn* for learning network for the benchmark data sets. Further, *bnlearn* is not able to finish learning the network when using *GS* for *D3* even after running for the cutoff time period of seven days. For both *IAMB* and *Inter-IAMB*, our sequential implementation outperforms *bnlearn* with a speedup of 1.7 – 2.8X for the benchmark data sets. Note that our implementation of the *GS* algorithm is 2 – 3X faster than the other two algorithms because of the optimization discussed in subsubsection 3.3.2. The score-computation optimization for *direct learning* algorithms, also discussed in subsubsection 3.3.2, ensures that our implementations of *MMPC* and *SI-HITON* achieve a speedup of 14.9 – 59.9X over that of *bnlearn*, while *bnlearn* is not able to learn the network for *D3* using *MMPC* in a week.

We validated the networks learned by our implementations against those learned by *bnlearn* for the data set *D1* using the five algorithms. During the validation process, we discovered a bug in the *Construct PC from MB* phase of the *bnlearn* implementation. It was caused by an erroneous assumption in the implementation that if there is only one element in the $\mathcal{MB}$ set of a variable then it must be in the $\mathcal{PC}$ set of that variable. This bug was acknowledged as such by the package's maintainer (personal communication, March 4, 2020). We fixed this bug in *bnlearn* and used the networks learned using this modified version for the purpose of the validation. For all the algorithms, the networks learned using our implementations recall more than 99.84% of the edges present in the networks learned using the corresponding implementations from *bnlearn* with more than 99.92% precision, i.e., our implementations learn more than 99.84% of the edges in the networks learned by *bnlearn* with less than 0.08% additional edges. We verified that these differences arise because of the optimization discussed in subsection 3.3.1.

Parallel Scalability of bnlearn

We use the five algorithms implemented in *bnlearn* for learning the BNs from the benchmark data sets using an increasing number of cores and measure their self-speedup, i.e., speedup compared to the sequential run-time of the *bnlearn* implementation. When using 2 cores, both *IAMB* and *Inter-IAMB* show a speedup of 1.9X and 1.4X for *D1* and *D2*, respectively. *bnlearn* shows further improvement when using 16 cores with an observed speedup of 6.9X and 2.3X. However, the speedup starts flattening when using cores on multiple nodes. For example, when using 64 cores across three nodes, the observed speedup for both *IAMB* and *Inter-IAMB* is 8.4X and 2.4X for *D1* and *D2*, respectively – a marginal improvement over the speedup using 16 cores. Therefore, speedup of the parallel implementations of the two faster algorithms in *bnlearn* show a pattern of diminishing returns. The corresponding self-speedup observed for the three slower algorithms is comparatively better – 34X, 28.5X, and 35X for *GS*, *MMPC*, and *SI-HITON*, respectively, for learning the network from *D2* using 64 cores. However, the run-times of these *bnlearn* algorithms when using 64 cores is still slower than our sequential implementations. Therefore, we do not explore the parallel performance of *bnlearn* further here. The scalability of our implementations, presented in subsection 3.4.4, outperforms *bnlearn* by a significant margin.

3.4.3 Effect of Load Balancing

In order to understand the extent of load imbalance during the parallel execution of the five algorithms, we learned BNs from the benchmark data sets using the algorithms, without the application of load balancing strategies discussed in subsection 3.3.3, and recorded the imbalance (as per Equation 2.6) at the end of each iteration. We observe that the imbalance during execution on less than 16 cores is less than 5.0 for all the algorithms. However, the imbalance increases for all the algorithms when executed on larger number of cores with more and more processes left without any work as the algorithms progress. *Inter-IAMB*,

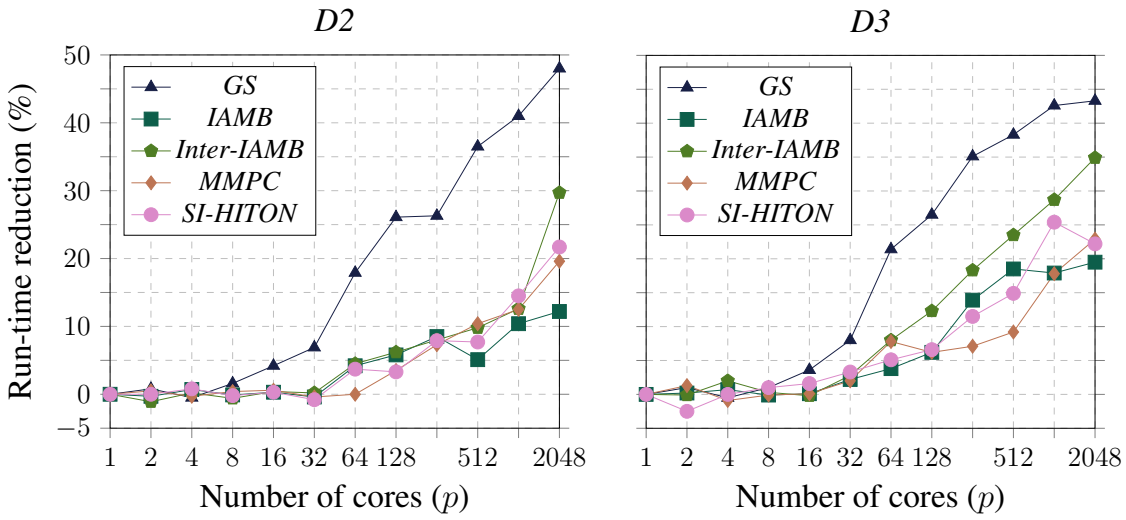


Figure 3.1: Plot of percentage reduction in the run-time, as a result of load balancing, of the five *local-to-global constraint-based* algorithms used for learning BN for *D2* and *D3* on different number of cores.

MMPC, and *SI-HITON* show worse measured imbalance as compared to the other two algorithms for all three data sets. For example, when running on 2048 cores, the worst final imbalance of 681.7 is shown by *Inter-IAMB* for *D2* and by *SI-HITON* for *D3*.

The percentage reduction in the run-time for learning the network with the application of the redistribution strategy from the two bigger data sets using the five algorithms for different number of cores is shown in Figure 3.1. When running on fewer cores, we observe almost no improvement in run-time with load balancing because the observed imbalance is small. Even when the imbalance is high, the time taken in measuring the imbalance and redistributing may be more than the corresponding gains. In such cases, we observe that the run-time increases marginally when load balancing is enabled, with the highest observed increase of just 2.5% when using *SI-HITON* on 2 cores. However, when running on larger number of cores, all the algorithms show reduction in the run-times with load balancing enabled. *GS*, in particular, shows a 43.4 – 48.0% improvement in the run-time for the three benchmark data sets when using 2048 cores even though the measured imbalance for the algorithm stays below 22.2 in the worst case. This is because the optimization for *GS*, discussed in subsubsection 3.3.2, enables faster candidate selection for many variables.

Therefore, the algorithm benefits more from a better spread of the work load through an evenly distributed $c\text{-}scores$ list. The run-time of the other four algorithms for the benchmark data sets also show significant improvement in the range of 12.2 – 34.9% on 2048 cores.

3.4.4 Parallel Scalability of Our Framework

Our procedure to read an input data set in parallel is as follows. First, the rows of the data set are block distributed to all the MPI processes. Then, the processes concurrently read the discretized data from their assigned rows. Finally, the read data is collected on all the processes to get the complete data set using `MPI_Allgatherv`. Once the BN is constructed, the corresponding network is written in *graphviz* [122] format. In our experiments, we observed that time taken in reading the data sets reduces from 5.1 seconds sequentially to 0.3 seconds on 2048 cores for *D1*, from 32.7 to 1.1 seconds for *D2*, from 106.7 to 3.4 seconds for *D3*, and from 104.1 to 5.0 seconds for the simulated data sets. Writing out the learned network takes less than 0.5 seconds in all the cases. For scalability discussions, we report only the time taken for constructing the BN by the parallel algorithm implementation and not for the I/O.

Strong Scaling for Benchmark Data sets

We conducted strong scaling experiments for all the algorithms using the benchmark data sets by repeatedly doubling the number of cores from 1 to 2048. Table 3.3 shows the average run-times for all the combinations of the algorithms, cores, and data sets. To better understand the performance of our implementations, we compute strong scaling speedup and efficiency using Equation 2.2. The strong scaling speedup of the five algorithms for the benchmark data sets as the number of cores used is increased are plotted in the first row of Figure 3.2 and the corresponding plots of efficiency are shown in the second row. Note that a perfect parallel implementation would achieve linear speedup and 100% efficiency.

Table 3.3: Time taken in learning the BNs for the benchmark data sets using the five *local-to-global constraint-based* algorithms on different number of cores, measured in seconds.

Data set	Algorithm	Run-time on different number of cores (s)											
		1	2	4	8	16	32	64	128	256	512	1024	2048
D1	*GS*	310.9	164.9	85.0	44.8	24.0	13.0	7.0	4.0	2.4	1.7	1.2	1.0
	IAMB	803.8	409.8	207.4	105.0	53.7	27.1	14.0	7.1	3.7	2.1	1.2	0.8
	Inter-IAMB	808.8	413.0	209.8	106.1	54.4	27.5	14.2	7.8	3.9	2.3	1.6	1.2
	MMPC	331.0	167.9	85.1	43.1	22.0	11.3	5.9	3.0	1.7	1.0	0.7	0.6
	SI-HITON	348.4	176.9	89.4	45.3	23.4	12.1	6.2	3.3	1.8	1.2	0.8	0.7
D2	*GS*	9,076.3	4,604.1	2,369.3	1,217.7	651.5	356.1	178.0	96.5	54.3	29.6	16.7	10.2
	IAMB	18,999.7	9,780.4	4,859.2	2,469.2	1,248.2	638.4	312.1	157.3	80.0	42.3	21.0	10.9
	Inter-IAMB	18,976.8	9,781.5	4,891.5	2,485.6	1,254.0	639.4	314.8	159.3	80.9	41.8	22.1	11.9
	MMPC	6,789.0	3,399.3	1,724.9	874.5	447.6	230.4	117.4	58.6	29.4	15.4	8.6	5.1
	SI-HITON	6,923.2	3,510.9	1,750.4	895.8	459.1	236.2	117.5	60.3	30.6	16.6	9.0	5.5
D3	*GS*	25,209.5	12,948.8	6,638.4	3,474.5	1,858.8	1,014.3	499.5	293.0	160.9	91.3	52.0	31.1
	IAMB	60,280.4	30,885.0	15,696.5	7,966.4	4,053.7	2,013.6	1,007.8	512.0	258.1	128.9	68.8	36.1
	Inter-IAMB	63,306.0	32,080.0	16,171.7	8,291.2	4,224.2	2,119.6	1,050.6	538.0	281.0	142.5	69.1	37.6
	MMPC	32,131.6	16,221.7	8,268.2	4,195.3	2,183.2	1,087.8	538.0	274.7	142.7	73.6	36.2	20.3
	SI-HITON	35,341.9	17,881.4	8,900.0	4,539.9	2,354.0	1,180.4	602.8	301.8	153.6	79.5	39.9	25.5

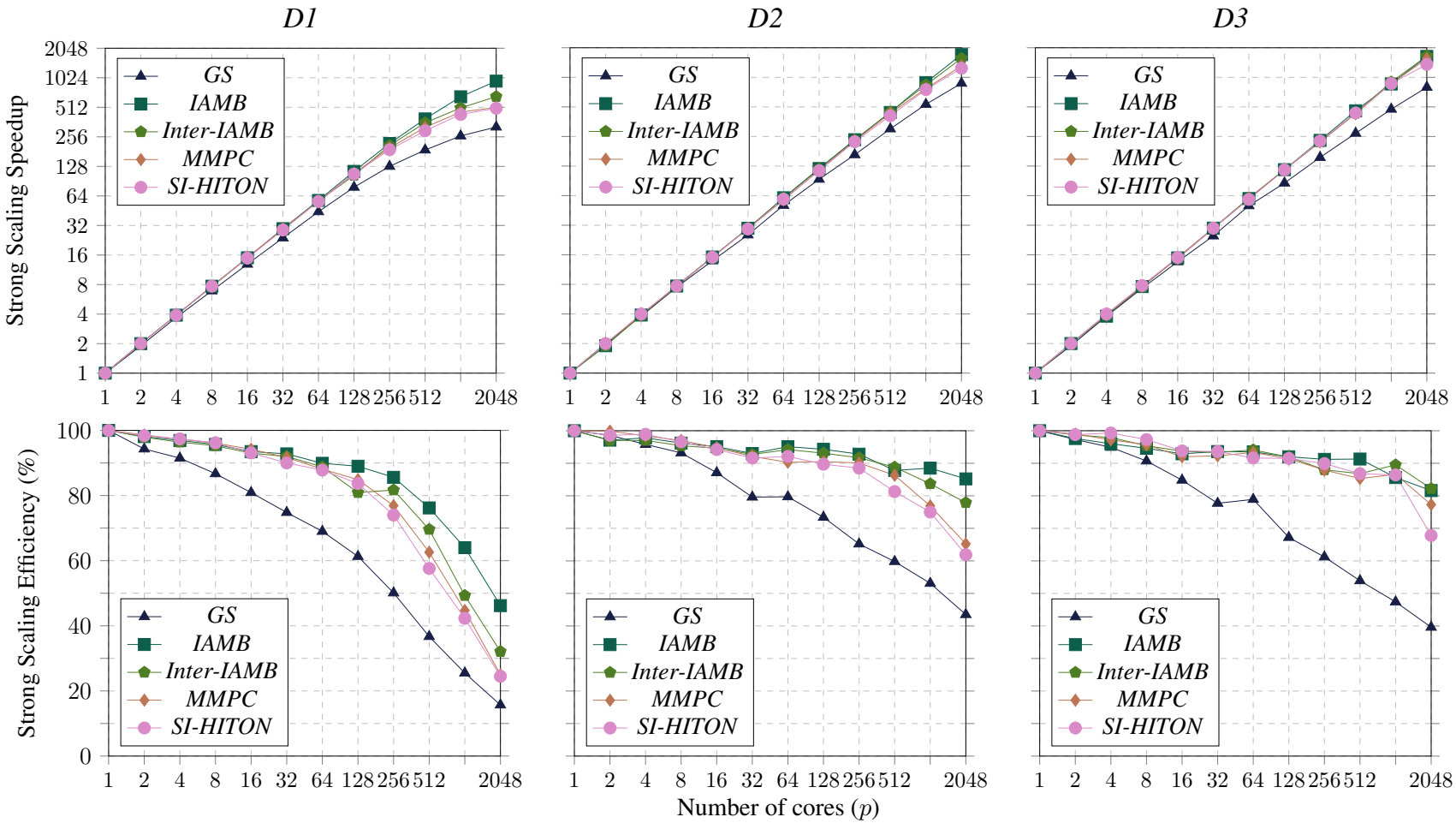


Figure 3.2: Plots of strong scaling speedup and efficiency of the five *local-to-global constraint-based* algorithms in constructing the BNs for the benchmark data sets as a function of the number of cores.

As can be observed from the plots in the figure, our implementations of all the algorithms show near-linear scaling on up to 2048 cores for the two larger data sets (*D2* and *D3*), while the scaling tapers off on 256 cores or more for the smaller data set (*D1*). The poor scaling for *D1* on larger number of cores can be explained by the lower total work required for learning BN from this data set, as demonstrated by the corresponding run-time of less than 3.9 seconds for all the algorithms on 256 cores and above. This loss of efficiency for *D1* also follows from the theoretical bounds on the number of processors that can be used by our parallel algorithms while being efficient as established in Equation 3.10, Equation 3.14, and Equation 3.15. Using $G_l = O(ml + r^l)$ for contingency tables (discussed in subsubsection 3.3.2) in the efficiency bounds, it can be seen that $n < 2^k$ in all the equations for the three data sets and $p = O(n)$. While $p = 2048$ is close to an order of magnitude smaller than n for *D2* and *D3*, it is greater than $n/3$ for *D1* and much closer to the asymptotic bound. Therefore, the poorer scaling of *D1* in our experiments, as compared to that of the other two data sets, corresponds well to our theoretical efficiency analysis of the parallel algorithms.

IAMB, *Inter-IAMB*, *MMPC*, and *SI-HITON* achieve a strong scaling efficiency of more than 75% when run on up to 1024 cores and more than 60% when run on up to 2048 cores for *D2* and *D3*. The efficiency of *GS*, however, is noticeably lower than the other four algorithms with a maximum efficiency of 43.5% on 2048 cores. This is because the optimization discussed in subsubsection 3.3.2 reduces the total work required by *GS*. On larger number of cores, this reduction in total work leads to lower computation load per processor, as compared to the other two *blanket learning* algorithms, and therefore the run-time of *GS* is dominated by communication time. For instance, the fraction of the total run-time spent by all five algorithms in communication while learning BNs from *D2* and *D3* is shown in Figure 3.3. These plots demonstrate a markedly higher communication overhead for *GS*, as compared to the other four algorithms, when running on larger number of cores. However, despite the lower efficiency, the optimization helps *GS* achieve a speedup of up

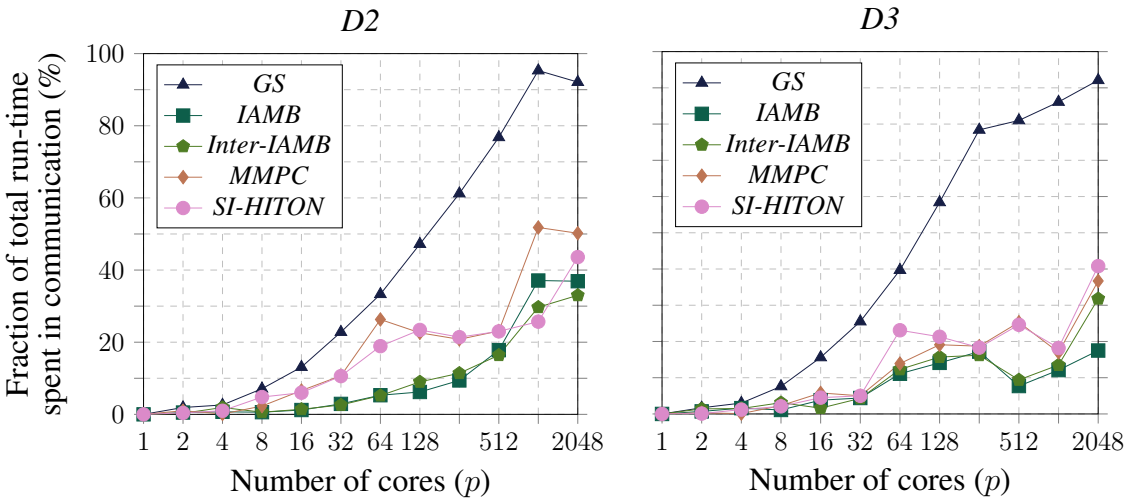


Figure 3.3: Plots of fraction of total run-time spent in communication on different number of cores by the five *local-to-global constraint-based* algorithms used for learning the BN for *D2* and *D3*.

to 1.2X over *IAMB* and *Inter-IAMB* on 2048 cores. Our implementations of the five *local-to-global* algorithms are able to learn BNs from the benchmark data sets in less than 38 seconds on 2048 cores, with a maximum speedup of $1,745$X and a corresponding 85.2% strong scaling efficiency.

Strong Scaling for Simulated Data sets

Our implementations of all the five algorithms scale linearly for learning BNs from the simulated data sets with even larger number of variables. In particular, scalability of *GS* improves significantly compared to the benchmark data sets. Our optimized sequential implementation of *GS* learns the network for *S1*, *S2*, and *S3* in 12.4, 17.0, and 26.5 hours, respectively. Using our parallel implementation on 2048 cores, the corresponding run-times are 36.1, 57.2, and 72.9 seconds. Strong scaling efficiency of *GS* for the simulated data sets is plotted in Figure 3.4. The considerable increase in the efficiency when compared to what is observed for the benchmark data sets (Figure 3.2) is in line with our discussion on the efficiency of the algorithm in the previous section. Since there is more total work required for learning networks from the simulated data sets, the computation load per processor

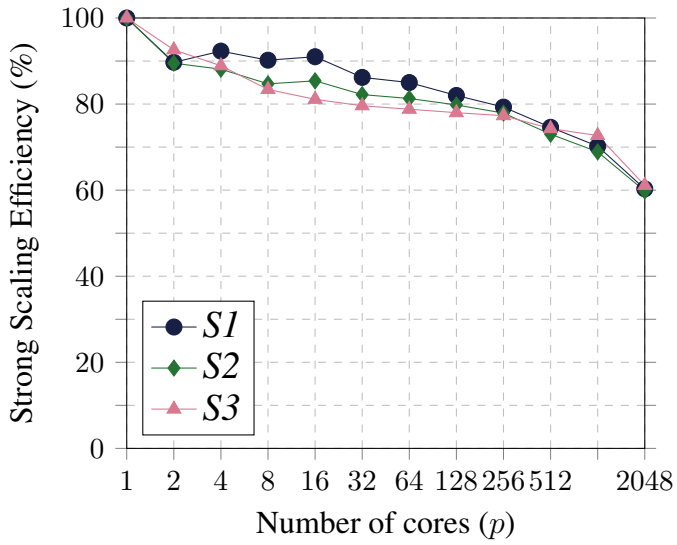


Figure 3.4: Plot of strong scaling efficiency of *GS* algorithm in constructing the BNs for the simulated data sets.

of the algorithm is high even when running on 2048 cores. Correspondingly, the fraction of run-time spent by the algorithm in communication for these data sets on 2048 cores is between 49.2% and 55.5%, that is almost half of that observed for the benchmark data sets.

Table 3.4 shows the sequential run-time of our optimized implementations of the five algorithms for the three simulated data sets. The table also shows the run-times for the simulated data sets on 2048 cores and the corresponding speedup. The parallel performance of our implementations improve as the edge addition probability increases, with all the algorithms achieving more than 56% strong scaling efficiency on 2048 cores for *S3*. Our parallel implementations of the *local-to-global* algorithms are able to reduce the time required for learning BNs from 33 hours for a sequential run to less than 78 seconds using 2048 cores. The maximum strong scaling efficiency achieved for the simulated data sets is 85.9% corresponding to a maximum speedup of $1,760$X.

Weak Scaling

We performed weak scaling experiments to investigate the scalability of our framework when the work per processor is fixed. Since the sequential run-time complexity of all five

Table 3.4: Time taken by our implementations of the five *local-to-global constraint-based* algorithms in constructing the BNs for the simulated data sets, sequentially and in parallel using 2048 cores, and the corresponding speedup.

Data set	Algorithm	Our run-time (s) *Sequential*	Our run-time (s) $p = 2048$	Speedup
S1	*GS*	$44,496.3$	36.0	$1,235.1$
	IAMB	$64,146.3$	56.1	$1,143.8$
	Inter-IAMB	$62,898.8$	71.9	875.1
	MMPC	$12,150.7$	26.0	466.8
	SI-HITON	$12,130.8$	24.2	501.4
S2	*GS*	$61,162.4$	49.9	$1,225.2$
	IAMB	$75,798.3$	57.5	$1,317.6$
	Inter-IAMB	$77,491.7$	47.9	$1,617.8$
	MMPC	$12,171.0$	12.1	$1,003.9$
	SI-HITON	$12,185.8$	12.4	982.9
S3	*GS*	$95,520.8$	76.4	$1,250.7$
	IAMB	$111,014.2$	63.1	$1,760.2$
	Inter-IAMB	$118,868.8$	77.1	$1,541.9$
	MMPC	$55,627.8$	48.1	$1,157.0$
	SI-HITON	$88,395.4$	74.3	$1,189.2$

algorithms is proportional to n^2 (Equation 3.4, Equation 3.6, and Equation 3.7), we conduct the experiments by varying the number of cores used and learning BNs on p cores from data sets with n_p variables such that n_p^2/p remains constant for different values of p. The weak scaling efficiency is then computed using Equation 2.4.

The plots of weak scaling efficiency of the five algorithms are shown in Figure 3.5. We used the data set *D2* and employed a maximum of 1024 cores for these plots in order to prevent the data set size from getting too small on small number of cores. As discussed earlier, we learned BN from the complete data set on 1024 cores and then learned BNs for a subset of n_p variables from the data set when using p (< 1024) cores such that n_p^2/p remains the same. The degradation in scaling efficiency for the different algorithms is in line with the communication intensity of the respective algorithms (Figure 3.3), which suggests that communication overhead is the limiting factor for weak scaling.

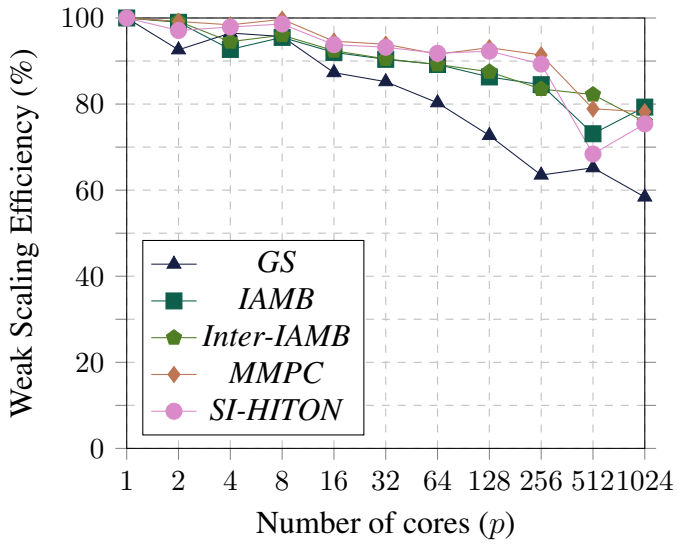


Figure 3.5: Plot of weak scaling efficiency of the five *local-to-global constraint-based* algorithms, measured for *D2*.

3.5 Summary of Contributions

In this chapter, we presented a parallel framework to scale *constraint-based* BN structure learning algorithms to tens of thousands of variables with a focus on *local-to-global* algorithms. We identified common components of these algorithms and developed parallel algorithms for each of these components. Subsequently, we demonstrated the applicability of our framework by using it to develop parallel versions of five different algorithms: *GS*, *IAMB*, *Inter-IAMB*, *MMPC*, and *SI-HITON*. We also introduced different algorithmic techniques that improved run-time performance of these algorithms in practice, both sequentially and in parallel.

We demonstrated the scalability of these algorithms using real data sets to learn genome-scale gene networks for the organisms *S. cerevisiae* and *A. thaliana* – networks with tens of thousands of variables from thousands of observations. The experiments showed that our optimized implementations of the *local-to-global* algorithms achieve significant sequential speedup over the popular *bnlearn* package in learning these networks. Further, our proposed parallel versions of these algorithms are able to learn the networks in less than 38

seconds on 2048 cores, compared to almost 18 hours required by our sequential implementation and more than 7 days required by *bnlearn*. Using simulated data sets, we showed that our algorithms are scalable to learning networks with even larger number of variables and can reduce the time required for the purpose from more than 25 hours sequentially to less than 78 seconds on 2048 cores.

CHAPTER 4

PARALLELIZING GLOBAL-SEARCH CONSTRAINT-BASED ALGORITHMS

We developed a parallel framework for parallelizing multiple *local-to-global constraint-based* algorithms in the previous chapter. In this chapter, we extend the framework to parallelize *global-search constraint-based* algorithms. Specifically, we focus on the most widely used algorithm in the category – *PC-stable*. We develop the new framework components required for parallelizing the algorithm and propose two different parallel versions of the algorithm using the extended framework. Our implementation of the algorithms utilizes a novel load balancing strategy to improve their performance in practice. Similar to the previous chapter, we investigate the scalability of our algorithms for *PC-stable* in constructing gene regulatory networks from real data sets with thousands of variables and thousands of observations. Our algorithms are able to reduce the time required for learning the networks from the biggest data set to 5.9 minutes using 4096 cores, as compared to a sequential run-time of 88.3 hours using our optimized implementation and more than seven days using the previous state-of-the-art approaches.

This chapter is structured as follows. In section 4.1, we describe the sequential *global-search* algorithms that is required for understanding our proposed parallel algorithms for *PC-stable* described in section 4.2. Then, we discuss novel strategies for improving the performance of the corresponding parallel implementations in section 4.3. We present the results of our experiments in section 4.4 and summarize our contributions in section 4.5.

4.1 Sequential Algorithms

We first describe the sequential execution of *global-search* algorithms. Unlike the *local-to-global* algorithms that start with an empty skeleton, the *global-search* algorithms begin with a fully-connected skeleton. Accordingly, the neighborhood set of every variable contains

all the other variables before the start of the algorithms, i.e., $\mathcal{PC}(T)$ is initialized with $\mathcal{X} \setminus \{T\} \forall T \in \mathcal{X}$. Then, the algorithms repeatedly execute *Eliminate* phase, that is similar to *Shrink* phase described in subsection 3.1.3. The only difference between the two is that *Eliminate* phase also requires a number s as input and only uses conditioning sets of size s when conducting the CI tests, i.e., in an *Eliminate* phase, Z is removed from $\mathcal{PC}(T)$ if $I(T, Z|\mathcal{S})$ for some $\mathcal{S} \subseteq \mathcal{PC}(T) \setminus \{Z\}$ such that $|\mathcal{S}| = s$. Both *PC* and *PC-stable* consider conditioning sets of increasing sizes for the *Eliminate* phase, i.e., the tests are repeated for $s = 0, \ldots, \min(l - 1, n - 2)$ where l is the maximum final neighborhood size of any variable. At the end of these iterations, the edges of BN skeleton are learned in the form of $\mathcal{PC}$ sets which are then directed to get the CPDAG for the BN structure.

Algorithm 12: Sequential *Eliminate* phase for *PC*

```
1  function CHECK-REMOVE-EDGE():
       Input: D, 𝒩, s
       Output: remove indicating if the edge should be removed
2      remove ← false
3      for 𝒮 ⊆ 𝒩 such that |𝒮| = s do
4          if I(X, Y|𝒮, D) then
5              remove ← true
6              break
7      return remove
8  function ELIMINATE-PC():
       Input: D, 𝒳, current 𝒫𝒞(·) sets, s
       Output: Updated PC(·) sets
9      for T ∈ 𝒳 do
10         for Z ∈ 𝒫𝒞(T) such that T < Z do
11             if CHECK-REMOVE-EDGE(D, PC(T) \ {Z}, s) or
                   [s > 0 and CHECK-REMOVE-EDGE(D, PC(Z) \ {T}, s)] then
12                 𝒫𝒞(T) ← 𝒫𝒞(T) \ {Z}
13                 𝒫𝒞(Z) ← 𝒫𝒞(Z) \ {T}
```

The sequential *Eliminate* phase for *PC* is shown in algorithm 12. Similar to the other *constraint-based* algorithms, *PC* defines an arbitrary ordering on the variables and considers them in that order (line 10). Since the $\mathcal{PC}$ sets are updated after every removal (line 12

– line 13), any removals from the $\mathcal{PC}$ sets of the variables ordered at the beginning can change the conditioning sets for the variables ordered towards the end. Therefore, any changes in the arbitrary ordering of the variables can have a significant effect on the final learned network [39]. In order to remove this dependency of the learned network on the ordering of variables in $\mathcal{X}$, *PC-stable* modifies the *Eliminate* phase as shown in algorithm 13. The only modification required by *PC-stable* is that a snapshot of the $\mathcal{PC}$ sets is stored at the beginning of the phase (line 2). Then, these $\mathcal{PC}$ sets are used to conduct all the CI tests during the execution of the phase (line 5).

Algorithm 13: Modified Sequential *Eliminate* phase for *PC-stable*

```
1  function ELIMINATE-PCSTABLE():
       Input: D, 𝒳, current 𝒫𝒞(·) sets, s
       Output: Updated PC(·) sets
2      prev-𝒫𝒞 ← 𝒫𝒞
3      for T ∈ 𝒳 do
4          for Z ∈ 𝒫𝒞(T) such that T < Z do
5              if CHECK-REMOVE-EDGE(D, prev-𝒫𝒞(T) \ {Z}, s) or
                 [s > 0 and CHECK-REMOVE-EDGE(D, prev-PC(Z) \ {T}, s)] then
6                  𝒫𝒞(T) ← 𝒫𝒞(T) \ {Z}
7                  𝒫𝒞(Z) ← 𝒫𝒞(Z) \ {T}
```

Time Complexity: Each execution of *Eliminate* phase can call CHECK-REMOVE-EDGE a maximum of two times for $O\binom{n}{2}$ different pairs of variables. Each such call for conditioning sets of size s may conduct a maximum of $O\binom{n}{s}$ CI tests. Assuming that conducting CI tests with a conditioning set of size s requires $O(G_s)$ time, the call to *Eliminate* phase will require $O\left(n^2\binom{n}{s}G_s\right)$. Since real-world networks are usually spares, we also assume that the neighborhood size of every variable is bounded by half the total number of variables, i.e., $l = O(n/2)$. Then, $\binom{n}{s} = O\binom{n}{l} \forall s = \{0, 1, \ldots, l-1\}$. Further, the time required for conducting CI tests can only grow with increasing conditioning set sizes, i.e., $G_s = O(G_l) \forall s \in \{0, 1, \ldots, l-1\}$. As the neighborhood sizes are bounded by l, *PC* and *PC-stable* will call ELIMINATE-PC and ELIMINATE-PCSTABLE phase l times, re-

spectively. Therefore, the sequential run-times of both the algorithms can be bounded by

$$O\left(ln^2\binom{n}{l}G_l\right) \tag{4.1}$$

4.2 Our Parallel Algorithms

We develop our parallel algorithms for *PC-stable* in this section. Towards this end, in subsection 4.2.1, we extend the parallel framework proposed in section 3.1 to enable efficient parallelization of *global-search constraint-based* algorithms. Then, we use the extended framework to propose two parallel algorithms for *PC-stable*. The first algorithm, discussed in subsection 4.2.2, works similar to *parallel-PC*. The second algorithm, proposed in subsection 4.2.3, is an alternate parallelization strategy that is expected to be scalable to a larger number of processors in practice.

4.2.1 Parallel Framework Extensions

Data Structure Modifications

We use the same three key data structures that are described in subsection 3.1.2. However, their usage and sequential initialization is modified for learning using *global-search* algorithms as described below:

- Since *global-search* algorithms try to remove an edge between variables X and Y using the neighbors of both X and Y, the initialization of $c\text{-}scores$ list is modified to have one element each for all the $\binom{n}{2}$ unordered variable pairs, i.e., the list is initialized with $\langle X, Y, 0\rangle \forall X, Y \in \mathcal{X}$ such that $X < Y$ in the ordering of the variables. At any point during the execution of algorithms, if $\langle X, Y, \theta_{XY}\rangle$ is a member of the $c\text{-}scores$ list, then X and Y are in the neighborhood sets of each other and θ_{XY} represents the score for the existence of the corresponding edge in the BN skeleton.

- $variables$ is again initialized with all the variables in $\mathcal{X}$.

- The initialization of $\mathcal{PC}$ sets is modified for *global-search* algorithms since they start with a fully connected network. Therefore, $\mathcal{PC}(T) = \mathcal{X} \setminus \{T\} \forall T \in variables$.

The initialization of the three data structures in parallel is done as described in subsection 3.1.2, with one key difference. Since computing θ_{XY} requires both $\mathcal{PC}(X)$ and $\mathcal{PC}(Y)$, $variables_j$ is initialized with $\{X | \langle X, Y, \theta_{XY} \rangle \in c\text{-}scores_j$ or $\langle Y, X, \theta_{XY} \rangle \in c\text{-}scores_j\}$ and then $\mathcal{PC}(T)$ is initialized for all $T \in variables_j$. Note that, this difference increases the number of $\mathcal{PC}$ sets stored on every processor from $O\left(\frac{n}{p}\right)$ for *local-to-global* algorithms to $O(n)$ for *global-search* algorithms. However, since we use a bit set representation for storing sets (as discussed in section 3.2), this does not have a significant effect on the total space requirements of the algorithms.

Parallel Eliminate Phase

We propose the addition of a new component for *Eliminate* phase to our parallel framework. The proposed component parallelizes the *Eliminate* phase for *PC-stable* (algorithm 13) as shown in algorithm 14 and works as follows. In every call, it first updates the score θ_{TY} for every element of the $c\text{-}scores_j$ list. The score for a pair of variables is computed as the minimum associativity between the two variables given the subsets of size s of the current neighborhood of the first variable (line 4), since the score defined as such can be used to ascertain CI (line 8). If *backward* is set to true, then the neighborhood of the second variable is also checked for computing the score (line 6). Finally, the elements with scores below the given threshold are removed from the $c\text{-}scores$ list and the corresponding $\mathcal{PC}$ sets are updated (line 9 – line 11).

Time Complexity: Each call to this phase requires the scores for $O\left(\frac{n^2}{p}\right)$ elements of $c\text{-}scores$ list on every processor, which takes $O(\binom{n}{s} G_s)$ time in computation for each

Algorithm 14: Parallel *Eliminate* Phase

```
1  function ELIMINATE-PHASE():
       Input: D, c-scores, current PC(·) sets, s, backward
       Output: Updated PC(·) sets
2      parallel j = processor's rank do
3          for ⟨T, Y, θ_TY⟩ ∈ c-scores_j do
4              θ_TY ← min_{S⊆PC(T)\{Y} s.t. |S|=s} Assoc(T, Y|S, D)
5              if backward and θ_TY ≥ −α then
6                  θ_TY ← min_{S⊆PC(Y)\{T} s.t. |S|=s} Assoc(T, Y|S, D)
7          for < T, Y, θ_TY >∈ c-scores_j do
8              if θ_TY < −α then
9                  Remove ⟨T, Y, θ_TY⟩ from c-scores_j
10                 PC(T) ← PC(T) \ {Y}
11                 PC(Y) ← PC(Y) \ {T}
```

element. Therefore, this phase requires $O\left(\frac{n^2}{p}\binom{n}{s}G_s\right)$ computation time and no communications.

4.2.2 Parallel Algorithm for *PC-stable*

Our parallel algorithm for BN skeleton learning using *PC-stable* is shown in algorithm 15. Similar to *parallel-PC*, it distributes the edges through the distributed $c\text{-}scores$ list. Then, using the parallel ELIMINATE-PHASE presented in algorithm 14, each processor conducts all the CI tests for its share of edges independently and updates its local $\mathcal{PC}$ sets corresponding to the eliminated edges. These updates are then synchronized across all the processors at the end of every iteration. Like the sequential algorithm, the execution of the parallel algorithm concludes when the neighborhood of all the variables becomes smaller than the conditioning set size. The $\mathcal{PC}$ sets at the end of the algorithm execution correspond to the skeleton for the BN.

Time Complexity: The computation run-time of this algorithm is dominated by the run-time of ELIMINATE-PHASE, which is called $O(l)$ times. This results in a total computation

Algorithm 15: Parallel Construct Skeleton - *PC-stable*

```
1  function CONSTRUCT-SKELETON-PCSTABLE():
       Input: 𝒳, D
       Output: 𝒫𝒞(T) sets for all T ∈ 𝒳
2      parallel j = processor's rank do
3          Initialize c-scores_j, variables_j, 𝒫𝒞(·) as described in subsection 3.1.2 and
             modified in subsection 4.2.1
4          s ← 0
5          repeat
6              backward ← (s > 0)
7              ELIMINATE-PHASE(D, c-scores_j, 𝒫𝒞, s, backward)
8              Synchronize 𝒫𝒞(·) across all the processors
9              s ← s + 1
10         until |𝒫𝒞(T)| > s for some T ∈ 𝒳
```

run-time of this algorithm of

$$O\left(l\frac{n^2}{p}\binom{n}{l}G_l\right) \tag{4.2}$$

The algorithm incurs the following two communication overheads in every iteration: 1) for synchronizing the $\mathcal{PC}$ sets, and 2) for determining if the next iteration with larger conditioning set size needs to be executed on any of the processors. Again, both these operations can be implemented using *all-reduce* which requires $O((\tau+\mu \log n) \log p)$ time. Therefore, the communication run-time of the algorithm is

$$O(l(\tau + \mu \log n) \log p) \tag{4.3}$$

Parallel Efficiency: The parallel efficiency of the algorithm can be computed using Equation 2.2 by substituting Equation 4.1 for $T_{\text{seq}}(n)$ and the sum of Equation 4.2 and Equa-

tion 4.3 for $T(n,p)$ as

$$E(n,p) = \frac{ln^2\binom{n}{l}G_l}{p \times \left(l\frac{n^2}{p}\binom{n}{l}G_l + l(\tau + \mu \log n)\log p \right)} \times 100\%$$

$$= \frac{n^2\binom{n}{l}G_l}{n^2\binom{n}{l}G_l + (\tau + \mu \log n)p \log p} \times 100\%$$

From the above equations, it can be seen that the denominator will asymptotically be the same as the numerator if $n^2\binom{n}{l}G_l > (\tau + \mu \log n)p \log p$ which implies that $p \log p < \frac{n^2\binom{n}{l}G_l}{\tau + \mu \log n}$. Noting that $p \log p < p^2$, this inequality can be simplified to get stricter bounds on the number of processors that can be used by algorithm 15 while being efficient as

$$p = O\left(n\sqrt{\frac{\binom{n}{l}G_l}{\tau + \mu \log n}} \right) \tag{4.4}$$

4.2.3 Alternate Parallel Algorithm for *PC-stable*

The parallelization of *PC-stable* presented in algorithm 15 assigns all the CI tests for an unordered variable pair to the same processor, i.e., the tests for a variable pair $\langle X, Y \rangle$ using the neighborhood of X as well as using the neighborhood of Y are always conducted by the same processor. Therefore, similar to the sequential algorithm, if X and Y are found to be independent using the neighbors of X, then the CI tests using the neighbors of Y are not conducted. This method of distributing CI tests ensures that the parallel algorithm conducts the same number of CI tests as the sequential algorithm. However, this distribution method is not optimal when the number of unordered variable pairs is comparable to the number of available processors. This is because, as the algorithm progresses and eliminates variable pairs found to be independent, some processors may be left without any work.

Algorithm 16: Alternate Parallel Skeleton Algorithm - *PC-stable*

```
1  function CONSTRUCT-SKELETON-PCSTABLE-ALTERNATE():
       Input: 𝒳, D
       Output: 𝒫𝒞(T) sets for all T ∈ 𝒳
2      parallel j = processor's rank do
3          Initialize c-scores_j, variables_j, 𝒫𝒞(·) as described in subsection 3.1.2 and
             modified in subsection 4.2.1
4          s ← 0
5          repeat
6              ELIMINATE-PHASE(D, c-scores_j, 𝒫𝒞, false)
7              Synchronize 𝒫𝒞(·) across all the processors
8              s ← s + 1
9              if s = 1 then
                   // Duplicate the unordered pairs
10                 for ⟨X, Y, θ_XY⟩ ∈ c-scores_j do
11                     Add ⟨Y, X, θ_XY⟩ to c-scores_j
12                 Redistribute c-scores
13             if s > 1 then
                   // Updates for removals on other processors
14                 for ⟨X, Y, θ_XY⟩ ∈ c-scores_j do
15                     if Y ∉ PC(X) then
16                         Remove ⟨X, Y, θ_XY⟩ from c-scores_j
17         until |𝒫𝒞(T)| > s for some T ∈ 𝒳
```

We propose an alternate parallelization strategy for *PC-stable*, shown in algorithm 16, that creates ordered variable pairs from the initial unordered pairs at the end of the first iteration (line 9 – line 12). This duplication allows CI testing of all the pairs using only the neighborhood of the first variable by modifying the call to ELIMINATE-PHASE (line 6 in algorithm 16 as compared to line 7 in algorithm 15). Since the two tuples corresponding to a pair might end up on different processors with this change, $c\text{-}scores$ list on every processor is updated for removals on the other processors (line 13 – line 16).

Time Complexity and Parallel Efficiency: The computation run-time of algorithm 16 is also dominated by that of ELIMINATE-PHASE and it requires the same communication as algorithm 15. Therefore, the computation and communication run-times of the algorithm

are also given by Equation 4.2 and Equation 4.3, respectively. Correspondingly, it is also efficient when using the number of processors bounded by Equation 4.4.

4.3 Implementation

We implemented the two proposed parallel algorithms for *PC-stable*, using *C++14* and MPI, as part of the same open-source software package that contains the implementations of our parallel *local-to-global constraint-based* algorithms [30]. We also utilized the same counting strategies as described in subsubsection 3.3.2 for these implementations.

4.3.1 Directing the Learned Skeleton

Similar to our implementations of the *local-to-global* algorithms, we implemented both our *PC-stable* algorithms to learn exactly the same BN as that learned by the corresponding implementation in *bnlearn*. In contrast to the *local-to-global* algorithms, though, the *bnlearn* implementation of *PC-stable* requires an additional overhead for remembering the conditioning sets used for removing all the edges as it uses this information to direct the edges of the learned BN skeleton using the rules of d-separation [37].

We store the conditioning sets used for removing a variable pair on the processor that eliminated the pair during the execution of the algorithm. Then, at the end of the skeleton learning procedure, the conditioning sets that can not be used for directing the edges are discarded. Finally, all the remaining conditioning sets are collected on all the processors for directing the edges without any communication. Even including the overhead required for storing and communicating the conditioning sets, directing the edges using our implementation still requires a maximum of 2.2% of the total run-time required for learning BNs in our experiments discussed in section 4.4, sequentially and in parallel.

4.3.2 Load Balancing

The performance of our implementations of the proposed parallel algorithms for *PC-stable* can be significantly impacted because of load imbalance. This is because of a combination of the following two reasons:

1. In every iteration, the call to ELIMINATE-PHASE removes some elements from the $c\text{-}scores$ list. However, the number of elements removed on every processor can vary widely. Since the computation run-time of ELIMINATE-PHASE on a processor depends on the size of $c\text{-}scores$ list on the processor, this may lead to imbalance between the processors.

2. The issue described above is similar to the problem of load imbalance during the execution of *local-to-global* algorithms discussed in subsection 3.3.3 with a key difference. During the execution of *local-to-global* algorithms, the neighborhood of every variable increases by a maximum of one variable in an iteration. On the other hand, in an iteration of *global-search* algorithms, the call to ELIMINATE-PHASE may remove multiple variables from the neighborhood of a variable. This further exacerbates the problem of load imbalance because the maximum number of CI tests with conditioning sets of size s that can be conducted using the neighborhood of a variable T is $\binom{|\mathcal{PC}(T)|}{s}$. Therefore, the disparity between the number of CI tests conducted for different variables increases as the algorithm removes dissimilar number of variables from the neighborhoods and also with increasing s.

To address the issues discussed above, we implemented two different approaches for balancing the load at the end of every iteration as described below. We compare the performance of the two approaches in subsubsection 4.4.2.

Simple Approach

The first approach that we implemented for load balancing works similar to the strategy used for the purpose in the implementation of *local-to-global* algorithms, described in subsection 3.3.3. At the end of every iteration, we block redistribute the $c\text{-}scores$ list to fix the imbalance in the size of the list on every processor. Unlike the approach for *local-to-global* algorithms, though, we redistribute the list at the end of every iteration for *PC-stable* algorithms. Since this approach assumes all the tuples require equal amount of work – an assumption that does not hold in practice because of reasons discussed above – we expect this approach to provide limited gains over the implementation without any load balancing.

Weighted Approach

We also implemented an alternate approach for load balancing that aims to minimize the difference between estimated computation run-times across the processors in every iteration. Towards this end, we assign a weight to every element of the $c\text{-}scores$ list that is proportional to an upper bound on the computation run-time required by ELIMINATE-PHASE for the element. We estimate this weight as the maximum number of CI tests that can be conducted for the element. For conditioning sets of size s, the maximum number of CI tests that can be conducted for a tuple $\langle X, Y, \theta_{XY} \rangle$ by algorithm 16 is $\binom{|\mathcal{PC}(X)|}{s}$, while algorithm 15 may conduct $\binom{|\mathcal{PC}(Y)|}{s}$ additional tests. Notice that, the block distribution of the elements of $c\text{-}scores$ to processors before the first iteration corresponds to the estimate of the run-times during the first iteration because exactly one CI test is conducted for every tuple when $s = 0$.

Once the weights for all the elements of $c\text{-}scores$ have been computed at the end of an iteration, the elements of the list are redistributed to minimize the maximum total weight of the list on any processor. However, finding an optimal assignment of the elements that minimizes the maximum weight is known to be an NP-hard problem for two or more processors [123]. Correspondingly, multiple heuristics have been developed for the purpose

with guarantees on the quality of the solution [124, 125, 126]. We implement a relatively straightforward heuristic that prioritizes minimizing the communication cost of the redistribution by moving the elements to the processors with neighboring ranks to fix the weighted load imbalance.

4.4 Experiments and Results

We conducted the experiments reported in this section on the Phoenix cluster at Georgia Tech, previously described in section 3.4. We also compiled our implementations and run the experiments as discussed in the section. We use a maximum of 171 nodes on the cluster for these experiments.

4.4.1 Data sets

Global-search constraint-based algorithms have been used in multiple previous studies for the construction of gene regulatory networks [127, 128, 129]. Further, we used learning of gene regulatory networks for our experiments in the previous chapter. Therefore, we show the efficacy of our proposed algorithms for *PC-stable* in the same application area.

Previous parallelization efforts for *PC-stable* have evaluated their implementations using a set of six gene expression data sets [96, 98, 97, 100]. These data sets were first compiled by Le et al. to evaluate *parallel-PC* [96]. For our experiments, we use the two largest data sets of these that were originally created for the DREAM5 challenge [130]. The first data set from the challenge consists of gene expression data for *S. aureus* that is a human pathogen and can cause diseases such as pneumonia, endocarditis, osteomyelitis, etc. This data set contains 160 observations each for $2,810$ genes. The second data set of 805 observations for $1,643$ genes is referred to as the DREAM5 *in silico* data set since it was created from a simulated network. Finally, we also use the *S. cerevisiae* gene expression data set from Tchourine et al. [121], previously described in subsection 3.4.1. This data set is bigger than any of the real-world data sets used in the previous parallelization

efforts for *PC-stable*.

Table 4.1: Benchmark data sets used for experimenting with *global-search constraint-based* algorithms.

Name	Organism	Genes (n)	Observations (m)
D4	*S. aureus*	$2,810$	160
D5	N/A (*in silico*)	$1,643$	805
D1	*S. cerevisiae*	$5,716$	$2,577$

Table 4.1 lists the data sets used for our experiments in this section. Since we used *D1* – *D3* to identify the data sets for presenting the results in section 3.4, we use *D4* and *D5* to refer to the new *S. aureus* and *in silico* data sets, respectively. We discretize and store these data sets as described in subsection 3.4.1 and use $\alpha = 0.05$ for learning BNs from these data sets for the results presented in this section.

4.4.2 Parallel Performance

We investigate the performance of our implementations of the two proposed parallel algorithms for *PC-stable* – the first one described in subsection 4.2.2, is referred to as "our primary" algorithm or simply "primary" algorithm, and the second one described in subsection 4.2.3, is referred to as "our alternate" algorithm or just "alternate" algorithm. We first discuss the effect of the two load balancing schemes on the run-time of our implementations. Then, we discuss the previous state-of-the-art approaches and finally present the results of strong scaling experiments conducted for our implementations.

We use the benchmark data sets listed in Table 4.1 for these experiments and learn networks for them in parallel, by repeatedly doubling the number of cores used for the purpose from 1 to 4096. We begin the execution in parallel by reading the data sets and writing the learned networks at the end as described in subsection 3.4.4. In all the cases, reading the data sets requires less than 5.1 seconds and writing the networks takes less than 0.2 seconds. Therefore, similar to the experiments for the *local-to-global constraint-*

based algorithms, we report only the time required for learning the networks for the results presented in this section.

Effect of Load Balancing

We proposed two different approaches for fixing the load imbalance at the end of every iteration of our parallel implementations in subsection 4.3.2 – simple and weighted. We evaluate these two approaches using our primary algorithm. For this purpose, we learn networks for the two bigger data sets (*D5* and *D1*) using the algorithm in the following configurations: without any load balancing, with simple load balancing, and with weighted load balancing. Then, we compute the reduction in the time required by our parallel implementation of the algorithm using the two load balancing approaches, as compared to the run-time without any load balancing, and plot it in Figure 4.1.

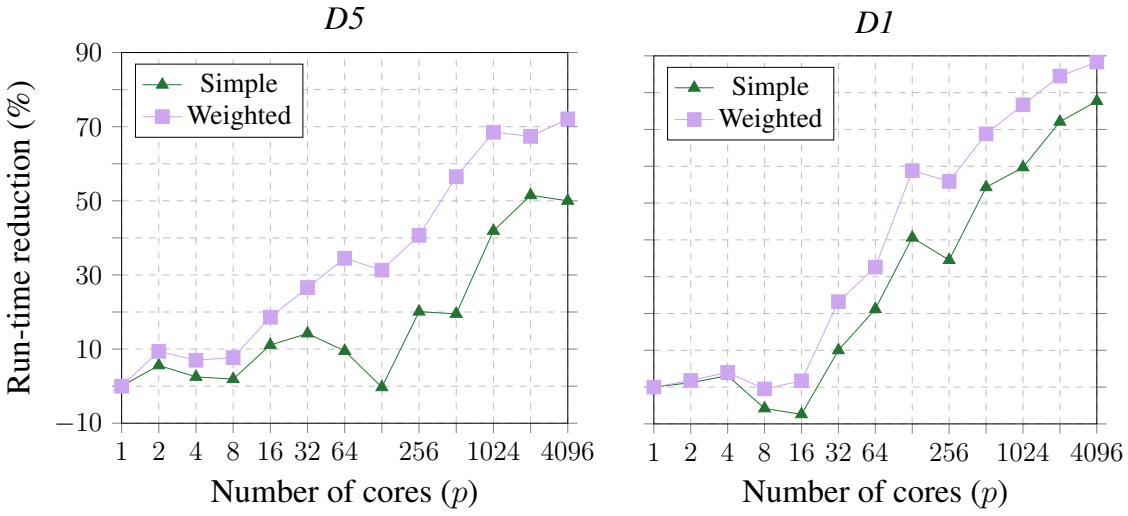


Figure 4.1: Plot of percentage reduction in the run-time of *PC-stable*, as a result of different load balancing schemes, for learning BN from data sets *D5* and *D1* on different number of cores.

The run-times show only marginal improvements using either approach when running on smaller number of cores, with load balancing even deteriorating the performance in some cases. For example, the run-time for *D1* shows a maximum increase of 7.4% on 32 cores when using the simple approach and a maximum increase of 0.5% on 16 cores

when using the weighted approach. This increase in run-time can be attributed to the fact that the time required for fixing the load imbalance may be more than the corresponding gains when the load imbalance is not severe. However, the run-times improve using both the approaches when learning on a larger number of cores. Further, both the approaches show increasingly higher improvement as the number of cores used is increased. In all the cases, the weighted balancing approach outperforms the simple approach with a maximum run-time improvement of 72.1% and 88.3% on 4096 cores for *D5* and *D1*, respectively. Therefore, we use the weighted approach for load balancing in our final implementation of both the proposed algorithms used for the scaling experiments.

Comparison with Previous State-of-the-Art

Le et al. implemented their *parallel-PC* algorithm as part of an *R* package [131]. As discussed in subsubsection 2.4.1, the parallel version of *PC-stable* implemented in *bnlearn* also follows an approach similar to *parallel-PC*. Recently, Hagedorn and Huegle compared the two *R* implementations and found the *bnlearn* implementation of *PC-stable* to be up to 400X faster than the corresponding implementation by Le et al. [101]. Therefore, we evaluated *bnlearn* as a potential baseline for our implementations.

The *bnlearn* implementation of *PC-stable* requires 54.8 hours to sequentially learn the network for *D5*, that is 21.7X slower than our implementation for learning exactly the same network, and does not finish learning the network for *D4* and *D1* in seven days while our sequential implementation is able to learn the networks in 3.2 minutes and 88.3 hours, respectively. Since the *bnlearn* implementation is significantly slower than our optimized implementations, we do not assess its parallel scalability. Instead, we use the optimized implementation of our primary algorithm, without any load balancing, as the baseline for our experiments and refer to it as the "optimized *parallel-PC*" in the discussion of the results next.

Table 4.2: Time taken by the optimized *parallel-PC* and our two parallel algorithms for *PC-stable* in learning the BNs for the benchmark data sets on different number of cores, measured in seconds.

Number of Cores (p)	*D4* Optimized *Parallel-PC*	*D4* Ours Primary	*D4* Ours Alternate	*D5* Optimized *Parallel-PC*	*D5* Ours Primary	*D5* Ours Alternate	*D1* Optimized *Parallel-PC*	*D1* Ours Primary	*D1* Ours Alternate
1	189.1	189.1	189.1	9,097.1	9,097.1	9,097.1	317,879.2	317,879.2	317,879.2
2	95.6	99.3	147.2	5,433.8	4,925.6	5,829.6	166,005.2	162,959.8	185,923.8
4	49.6	51.6	76.5	2,783.8	2,588.1	3,004.0	86,425.1	82,992.5	94,412.1
8	25.2	27.9	39.9	1,471.1	1,358.5	1,550.2	45,204.2	45,438.5	51,683.6
16	15.0	17.0	22.4	881.6	717.8	817.5	24,144.2	23,724.0	27,170.7
32	8.6	8.7	11.6	516.6	379.0	445.2	18,063.2	13,871.7	15,261.5
64	6.3	5.5	7.6	312.6	204.9	236.3	12,159.0	8,197.0	8,310.4
128	3.6	3.6	4.6	171.1	117.5	146.1	10,698.7	4,407.1	4,897.8
256	2.7	2.3	3.0	141.3	83.8	92.0	6,323.7	2,786.7	2,804.1
512	2.3	1.6	2.1	113.6	49.5	54.3	5,478.5	1,709.5	1,683.0
1024	1.4	1.4	1.4	102.5	32.3	30.3	4,169.1	971.8	1,033.3
2048	1.8	1.1	1.3	68.3	22.2	19.9	3,726.5	577.3	637.2
4096	2.9	3.6	4.0	46.2	12.9	18.2	3,027.6	354.7	370.6

Strong Scaling Performance

Table 4.2 shows the time required by the baseline and our two algorithms for learning networks from the benchmark data sets when using different number of cores. The corresponding strong scaling speedup and efficiency, computed using Equation 2.2, is shown in the first and the second row of Figure 4.2, respectively.

Our primary algorithm outperforms optimized *parallel-PC* by a significant margin for learning BNs from the two bigger data sets, i.e., *D5* and *D1*, using any number of cores. Further, our alternate algorithm also outperforms optimized *parallel-PC* for the two bigger data sets, when running on 32 cores or more. However, our primary algorithm is slower than optimized *parallel-PC* for *D4* when using 32 cores or fewer. This is because the total work required for learning the network from the data set is very low, as evidenced by the corresponding sequential run-time of about three minutes and run-time of less than 10 seconds of all the implementations for the data set when using 64 cores or more. Therefore, the gains from a balanced load can not compensate for the overhead required to achieve it.

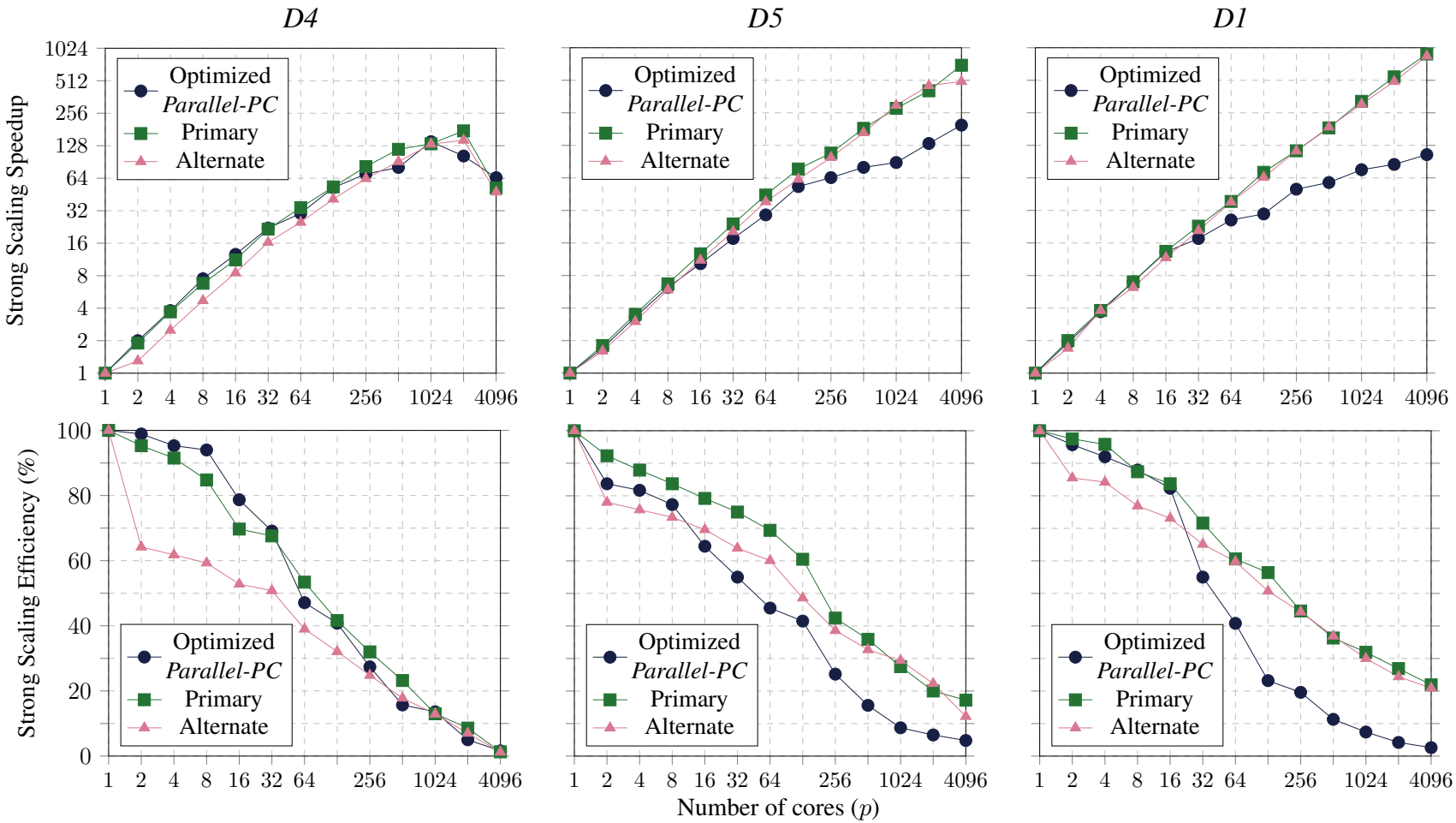


Figure 4.2: Plots of strong scaling speedup and efficiency of the optimized *parallel-PC* and our two parallel algorithms for *PC-stable* in constructing the BNs for the benchmark data sets as a function of the number of cores.

When using fewer cores, our alternate algorithm is significantly slower than our primary algorithm for learning networks from all the three data sets, with even optimized *parallel-PC* outperforming the alternate algorithm in some cases. Le et al. asserted that conducting the CI tests for an unordered variable pair $\langle X, Y \rangle$ using $\mathcal{PC}(X)$ and $\mathcal{PC}(Y)$ on different processors, as is done in our alternate algorithm, is inefficient [96]. This is because if X and Y are found to be independent using subsets of $\mathcal{PC}(X)$, then the tests conducted using subsets of $\mathcal{PC}(Y)$ would be extra work as compared to the sequential implementation. However, the performance of our two algorithms are very similar when running on larger number of cores. Further, our alternate algorithm is faster than our primary algorithm for *D5* on 1024 and 2048 cores. This is in contrast to the observation by Le et al. The results of our experiments show that while conducting CI tests using both the sets of neighborhood sets for an unordered variable pair may be an indisputably good strategy when using fewer cores, splitting up the tests on larger number of cores to provide enough work to every processor may result in a better performance.

Even though our implementations of the two proposed algorithms achieve significantly higher speedup than the baseline for the two bigger data sets, the maximum strong scaling efficiency achieved by the algorithms on 4096 cores is only about 21%. This lower efficiency seems to contradict the theoretical bounds on the number of processors that can be used by the algorithms while being efficient, as specified in Equation 4.4. Even with the weighted load balancing scheme, we observe imbalance between the run-times of the different processors. This discrepancy between the theoretical analysis and the practical performance is due to the fact that the actual number of CI tests that will be conducted for a variable pair can not be determined a priori. Therefore, both the theoretical analysis as well as the weight computations for load balancing assume that the maximum number of CI tests will be conducted for all the variable pairs in the $c\text{-}scores$ list. Since this assumption does not hold in practice, we observe a drop in efficiency of all the implementations for all three data sets in Figure 4.2 as the load is distributed across more processors. Nonetheless,

the weighted balancing of load prevents the steep efficiency loss in our implementations as compared to the one observed for optimized *parallel-PC*.

Our two algorithms are able to reduce the time required for learning the networks from 3.2 minutes sequentially to 4 seconds using 4096 for *D4*, from 2.5 hours to less than 18.2 seconds for *D5*, and from 88.3 hours to less than 6.2 minutes for *D1*. For learning from the biggest data set that we used, our primary and alternate algorithms achieve the maximum speedup of 896.2X and 857.7X on 4096 cores while the corresponding speedup obtained by the baseline implementation is 105X.

4.5 Summary of Contributions

We extended our framework for *constraint-based* algorithms to parallelize *global-search* algorithms in this chapter. We used the extended framework to propose two parallel algorithms for the widely used *PC-stable* algorithm. Our proposed parallel algorithms utilize the same basic idea as the previous approaches but, unlike the prior works, we theoretically analyzed the algorithms and showed that they are efficient on a large number of processors. Further, we optimized our implementations of the algorithms and employed a novel load balancing technique that improved the performance of our algorithms in practice.

The results of our experiments showed that our implementations of the parallel algorithms for *PC-stable* are scalable to thousands of cores. The algorithms are able to learn gene networks from a real-world gene expression data set for *S. cerevisiae* with $2,577$ observations for $5,716$ genes in 5.9 minutes on 4096 cores, as compared to sequentially requiring 88.3 hours using our optimized implementation and more than seven days using the previous state-of-the-art CPU-based implementation. To the best of our knowledge, the *S. cerevisiae* data set used in these experiments is bigger than any of the gene-expression data sets used in the previous works that parallelized *global-search* algorithms.

CHAPTER 5

PARALLELIZING MODULE NETWORK CONSTRUCTION

We developed efficient parallelizations of multiple *constraint-based* algorithms in the previous two chapters. As discussed in subsection 2.4.1, previous works have proposed parallel algorithms for BN structure learning using *score-based* methods that are able to learn networks with tens of thousands of variables [49]. Instead, we choose to focus on parallelizing the construction of MoNets – an important specialization of BNs that are also learned using *score-based* methods.

In this chapter, we present the first scalable distributed memory parallel solution for constructing MoNets. We described the two software packages that are primarily used for learning MoNets, *GENOMICA* and *Lemon-Tree*, in subsection 2.2.2. Since *Lemon-Tree* is more widely used of the two, as discussed in subsection 2.4.2, we parallelize the methodology used by *Lemon-Tree* in this work. Similar to the previous chapters, we demonstrate the scalability of our parallel method for 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 15.2 minutes and 2.8 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.

This chapter is organized as follows. First, we describe the sequential algorithm of *Lemon-Tree* in section 5.1 which is required for understanding its proposed parallelization in section 5.2. Then, we discuss the optimized sequential implementation and the implementation of the parallel algorithm in section 5.3. Finally, in section 5.4, we discuss in detail the experiments we conducted to evaluate the performance of the parallel implementation and summarize our contributions in section 5.5. The content covered in this chapter

improves the results presented in the work that has been accepted to appear in the following peer-reviewed paper:

- A. Srivastava, S. Chockalingam, M. Aluru, and S. Aluru, "Parallel Construction of Module Networks," in *2021 SC21: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, 2021

5.1 Sequential *Lemon-Tree* Algorithm

Lemon-Tree implements the MoNet learning method proposed by Bonnet et al. [71]. This MoNet learning method consists of three main tasks that are executed in the order they are described below.

5.1.1 GaneSH Co-Clustering

The first task constructs an ensemble of variable clusters using a Gibbs sampler algorithm called *GaneSH*, proposed by Joshi et al. [83]. 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 [83]) that can be computed by aggregating the values from 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 $O(GUK^2Lnm)$.

5.1.2 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 [84] 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.

5.1.3 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 subsection 5.1.1) 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 [79, 132], 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 $\langle X_i, \mathsf{D}_{ij} \rangle$ 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 [80], 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 [80, 71]. 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

$$O(GUK^2Lnm + Gn^2 + UKLSnm^2) \tag{5.1}$$

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 5.4.

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.

5.2 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.

5.2.1 Assumptions

We develop the proposed parallel algorithms for execution using p processors assuming the networked distributed memory model described in subsection 2.3.1. We also 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 subsection 5.3.2.

5.2.2 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.

GaneSH Co-Clustering

The sequential *GaneSH* task samples an ensemble of variable clusters by performing variable-observation co-clustering as described in subsection 5.1.1. 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.

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. The pseudo-code for our parallel algorithm for these functions is described in algorithm 17. 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

Algorithm 17: 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 $\mathcal{V}$

a cluster V_s with probability in proportion to the reassignment scores and reassigns X_r to V_s (line 3 – line 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 (line 14 – line 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 obser-

Algorithm 18: Parallel Update of Observation Clusters

function REASSIGN-OBS-CLUSTER():
  Input: Number of observations m, Data set D
  Input/Output: Set of observation clusters $\mathcal{O}(V_i)$
  parallel $k =$ *rank of processor* **do**
    for $i \leftarrow 1$ **to** m **do**
      $r \leftarrow$ SELECT-UNIF-RAND($\{1, \ldots, m\}$)
      $O_r \leftarrow$ Cluster assignment of D_r in $\mathcal{O}(V_i)$
      $\mathcal{O}_k \leftarrow k^{\text{th}}$ block of $\mathcal{O}(V_i) \cup \{\text{empty cluster}\}$ partitioned into p blocks
      for $O_j \in \mathcal{O}_k$ **do**
        $ou\text{-}scores_k(O_j) \leftarrow$ Score for moving D_r to O_j if $O_j \neq O_r$, else for keeping D_r in O_r
      $O_s \leftarrow$ SELECT-WTD-RAND($\mathcal{O}(V_i)$, $ou\text{-}scores_k$)
      if $O_r \neq O_s$ **then**
        Move D_r to O_s and update $\mathcal{O}(V_i)$

function MERGE-OBS-CLUSTER():
  Input/Output: Set of observation clusters $\mathcal{O}(V_i)$
  parallel $k =$ *rank of processor* **do**
    for $O_i \in \mathcal{O}(V_i)$ **do**
      $\mathcal{O}_k \leftarrow k^{\text{th}}$ block of $\mathcal{O}(V_i)$ partitioned into p blocks
      for $O_j \in \mathcal{O}_k$ **do**
        $om\text{-}scores_k(O_j) \leftarrow$ Score for merging O_i with O_j if $O_i \neq O_j$, else for retaining O_i
      $O_s \leftarrow$ SELECT-WTD-RAND($\mathcal{O}(V_i)$, $om\text{-}scores_k$)
      if $O_i \neq O_s$ **then**
        Merge O_i and O_s and update $\mathcal{O}(V_i)$

vation clusters $\mathcal{O}$ while keeping $\mathcal{V}$ the same. Our proposed parallel algorithms for these two functions are shown in algorithm 18. Similar to the functions for updating variable clusters, the pseudo-code for reassigning data instances from one 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 19: Parallel *GaneSH* Co-Clustering

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

Our parallel algorithm for the *GaneSH* task is shown in algorithm 19. 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 $\mathcal{O}(V_i)$ (line 3 – line 5). Then, the algorithm proceeds to the main loop of the update steps (line 6 – line 11). In each update step, the parallel functions defined in algorithm 17 update the variable clusters (line 7 – line 8) and those defined in algorithm 18 update the observation clusters (line 9 – line 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 $\mathcal{V}$.

Consensus Clustering

The consensus clustering task takes the G samples of $\mathcal{V}$ generated by algorithm 19 as input and outputs the consensus variable clusters. In our experiments, described in section 5.4,

Algorithm 20: Parallel Learning of Tree Structures

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 – $\mathcal{T}(\mathrm{M}_i)$
  parallel $k = \textit{rank of processor}$ **do**
    $\mathcal{O}(\mathrm{M}_i) \leftarrow$ Randomly assign observations $\mathsf{D}_j \; \forall j \in \{1, \ldots, m\}$ to $\sqrt{m}$ observation clusters
    $\mathcal{S}(\mathrm{M}_i) \leftarrow \emptyset$ `// Sampled Observation Clusters`
    for $u \leftarrow 1$ **to** U **do** `// GaneSH Loop`
      REASSIGN-OBS-CLUSTER(m, D, $\mathcal{O}(\mathrm{M}_i)$)
      MERGE-OBS-CLUSTER($\mathcal{O}(\mathrm{M}_i)$)
      if $u > B$ **then**
        Add the current $\mathcal{O}(\mathrm{M}_i)$ to $\mathcal{S}(\mathrm{M}_i)$
    for $\mathcal{Q} \in \mathcal{S}(\mathrm{M}_i)$ **do** `// Build Tree Ensemble`
      $\mathcal{Q}_k \leftarrow k^{\text{th}}$ block of $\mathcal{Q}$ partitioned into p blocks
      $subtrees_k \leftarrow$ Trees with a node for all $\mathrm{Q}_i \in \mathcal{Q}_k$
      repeat
        $tm\text{-}scores_k \leftarrow$ Scores for merging consecutive trees in $subtrees_k$
        $max\text{-}tms \leftarrow$ ***all-reduce*** $\max_{0 \leq k < p} tm\text{-}scores_k$
        Merge the trees corresponding to $max\text{-}tms$
      until $\sum_{0 \leq k < p} |subtrees_k| = 1$
      bcast the remaining tree in $subtrees_k$ to all the processors and add it to $\mathcal{T}(M_i)$

executing the consensus clustering task requires less than 0.04% of the total sequential runtime 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 subsection 5.1.2, on all p processors in our parallel solution.

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. The pseudo-code for the construction of an ensemble of regression tree structures for a module $\mathrm{M}_i \in \mathcal{M}$ is shown in algorithm 20. 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 $\mathcal{S}(\mathrm{M}_i)$ (line 3 – line 9). Unlike the *GaneSH* run described in the section subsubsection 5.2.2, the variable clusters are not updated. Therefore, only the parallel *GaneSH* functions for observation clustering, presented in algorithm 18, 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 $\mathcal{Q} \in \mathcal{S}(\mathrm{M}_i)$ (line 10 – line 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*, contributing to more than 90% of the sequential run-times in our experiments. It requires 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. Our parallel solution for this phase is depicted in algorithm 21.

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

Algorithm 21: 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 $wr\text{-}splits$,
2 Random splits $ur\text{-}splits$
3 **parallel** $k =$ *rank of processor* **do**
4 $cand\text{-}splits \leftarrow$ List of tuples $\langle \mathrm{M}_i, T, N, X_i, \mathsf{D}_j \rangle$ for all $\mathrm{M}_i \in \mathcal{M}$, $T \in \mathcal{T}(\mathrm{M}_i)$, $N \in$ *internal-nodes*(T), $X_i \in \mathcal{P}$, $\mathsf{D}_j \in$ *observations*(N)
5 $cand\text{-}splits_k \leftarrow k^{\text{th}}$ chunk of $cand\text{-}splits$ distributed into p chunks as per subsubsection 11
6 **for** $\langle \mathrm{M}_i, T, N, X_i, \mathsf{D}_j \rangle \in cand\text{-}splits_k$ **do**
7 $cand\text{-}probs_k[\langle \mathrm{M}_i, T, N, X_i, \mathsf{D}_j \rangle] \leftarrow$ Posterior probability of assigning the split $\langle X_i, \mathsf{D}_{ij} \rangle$ 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 $tnode\text{-}splits_k \leftarrow$ Elements of $cand\text{-}splits_k$ in which the first three elements are $\langle \mathrm{M}_i, T, N \rangle$
10 $tnode\text{-}probs_k \leftarrow$ Computed probabilities for the elements of $tnode\text{-}splits_k$ from $cand\text{-}probs_k$
11 **for** $s \leftarrow 1$ **to** J **do**
12 $wr\text{-}splits[\langle \mathrm{M}_i, T, N, s \rangle] \leftarrow$ SELECT-WTD-RAND($tnode\text{-}splits_k$, $tnode\text{-}probs_k$)
13 $ur\text{-}splits[\langle \mathrm{M}_i, T, N, s \rangle] \leftarrow$ SELECT-UNIF-RAND($tnode\text{-}splits_k$)

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 and assigned to the different processors as discussed below (line 5).

Distributing Candidate Splits: The list of candidate splits can trivially be partitioned into p blocks of equal size in order to distribute the computation load. We also propose an alternate strategy for the purpose. As discussed in subsection 5.1.3, the computation

Algorithm 22: Parallel Learning of Modules

```
1 function LEARN-MODULE-CPDS():
      Input: m, D, M, P, U, B, J
2     parallel k = rank of processor do
3         for M_i ∈ M do
4             T(M_i) ← LEARN-TREE-STRUCT(m, D, M_i, U, B)
5         LEARN-TREE-SPLITS(D, M, T, P, J)
6         LEARN-PARENTS(M, wr-splits, ur-splits)
```

time required by a candidate split is $O(Sm)$. Therefore, the time required is proportional to both the number of sampling steps (S) and the number of observations (m). Since S can not be determined a priori, we weight each candidate split by m and then distribute the splits to minimize the maximum weight on each processor. Notice that, this is similar to the weighted approach for load balancing in *global-search constraint-based* algorithms. Therefore, we use a heuristic similar to the on described in subsection 4.3.2 to distribute the list with weights while ensuring that the splits for a node in the list are contiguous. We compare the performance of the two distribution approaches – unweighted and weighted – in subsubsection 5.4.3.

Once the splits are partitioned, the posterior probabilities for all the local candidate splits are computed and stored on each processor (line 6 – line 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 (line 8 – line 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 $cand\text{-}probs_k$. Then, the splits for all the nodes in $cand\text{-}splits_k$ 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.

The size of $cand\text{-}splits_k$, and therefore $cand\text{-}probs_k$, is bounded by $O(KRLnm/p)$ and

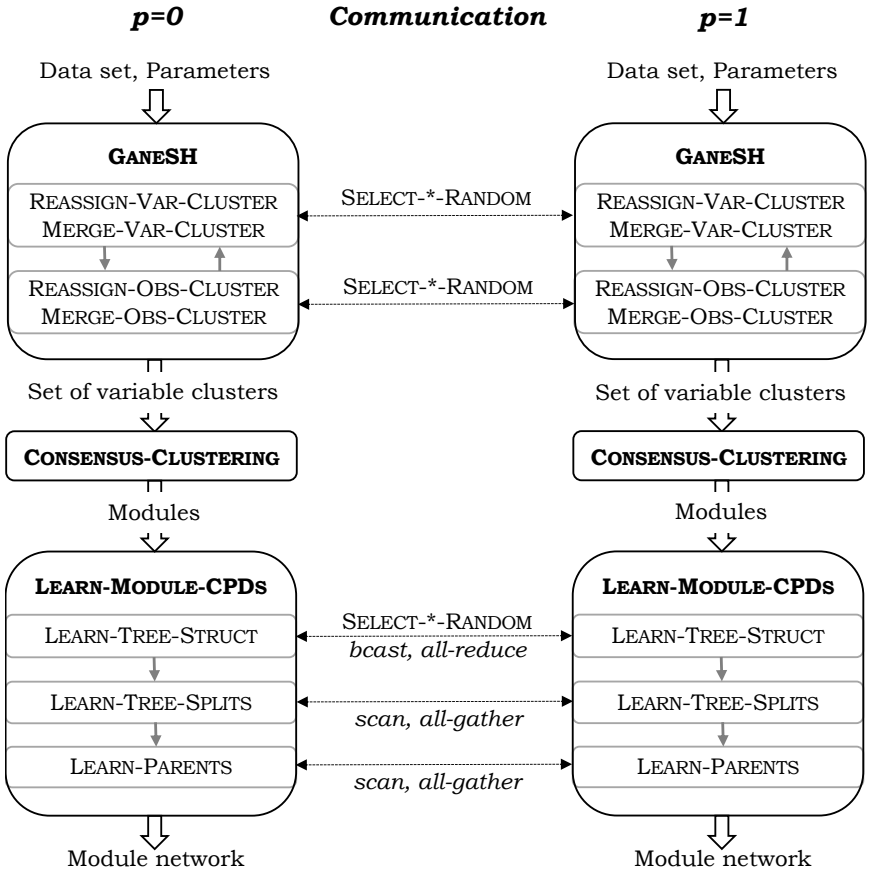


Figure 5.1: Schematic diagram showing the execution flow of our parallel algorithm for learning MoNets with two processors, using the parallel functions developed in section 5.2.

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 22. 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 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 5.1. 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.

5.3 Implementation

We implemented both the sequential and the parallel versions of the algorithm discussed in this section as part of an open-source software that we developed [31].

5.3.1 Sequential Implementation

Lemon-Tree software uses *Java* to implement the approach outlined by Bonnet et al. [71]. Even though any software written in *Java* requires compilation, it is referred to as an interpreted language [133]. 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 [133, 134, 135]. 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 subsubsection 5.4.2.

As discussed in subsection 2.4.2, *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 subsubsection 5.4.2.

5.3.2 Parallel Implementation

We implemented the parallel algorithms proposed in section 5.2 using the MPI conforming to the *MPI 3.1* standard. For generating random numbers in parallel, we use the *TRNG* library that provides multiple parallelizable PRNGs [136]. We used a multiple recursive generator [137] with 3 feedback terms and a Sophie-Germain prime modulus for the experiments reported in section 5.4. Note that our implementation can use any parallel PRNG supported by the library.

In order to implement the distributed random sampling functions described in subsection 5.2.1, 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 [136].

5.4 Experiments and Results

We performed our experiments on the Phoenix cluster at Georgia Tech. We provided a brief overview of the cluster resources in section 3.4 and use a maximum of 171 nodes of the cluster for the experiments reported in this section. 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 and bind one MPI process to each core.

5.4.1 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 [57, 138, 139] to plants and animals [112, 140]. Therefore, as in the previous chapters, we again demonstrate the scalability of our parallel implementations for MoNets in constructing genome-scale gene regulatory networks from the gene expression data sets introduced in subsection 3.4.1 and listed in Table 3.1. Note that, unlike the previous chapters, we use the raw expression values for learning MoNets.

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 5.1, 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 [49], and is outside the scope of this work. All the runs are repeated three times for different random seeds and the average run-times are reported.

5.4.2 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.

Comparison with Lemon-Tree

We compared the run-time of the modified *Lemon-Tree* with that of our optimized sequential implementation (both described in subsection 5.3.1) for constructing MoNets. Both *Lemon-Tree* as well as our implementation did not finish learning MoNet for *D1* 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 5.1 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.

Sequential Run-time Estimates for Large Data sets

Both the sequential implementations are not able to construct a MoNet from *D1* 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 for constructing MoNets using 30 smaller data sets constructed from *D1* 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.

Table 5.1: Comparison of the time taken by *Lemon-Tree* and our sequential implementation in constructing MoNets using the first n variables and m observations of *D1*, measured in seconds, and the corresponding speedup.

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

Figure 5.2 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 5.3, 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 run-time complexity (Equation 5.1), we observe that the growth rate with increasing m corresponds well to the

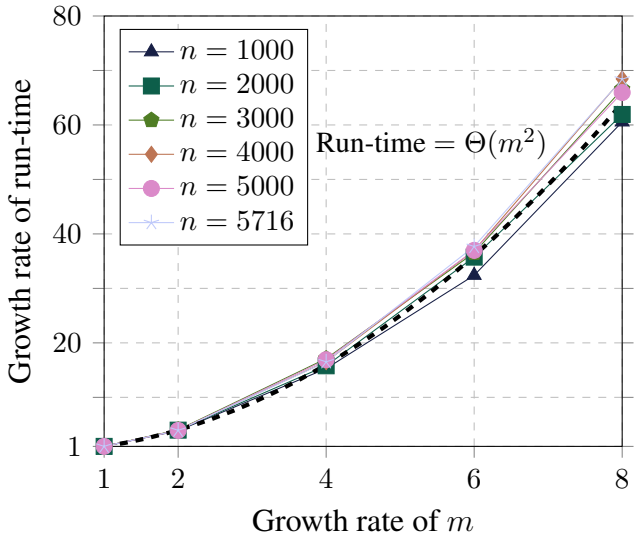


Figure 5.2: Plots of growth rate of sequential run-time for learning MoNets as the number of observations grow for data sets with different number of variables.

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 *D1* 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 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 *D1*. Similarly, we also estimate the lower bound on the run-time of our sequential implementation for *D2* 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.

The estimated minimum run-time of our sequential implementation for *D3* is $175,932.7\times$

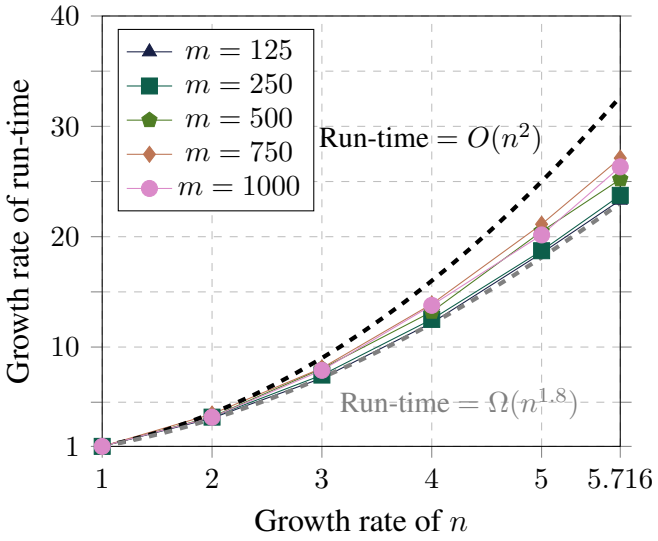


Figure 5.3: Plots of growth rate of the sequential run-time for learning MoNets as number of variables grow for data sets with different number of observations.

$(16,838/1,000)^2 \times (18,373/5,716)^{1.8}$ seconds or about 13 years. Even a perfect parallel implementation will require at least four weeks for learning MoNets from this data set on 4096 cores – the maximum number of cores that we used. Therefore, we limit our experiments in this chapter to *D1* and *D2*.

5.4.3 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. 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 *D1* in parallel takes 0.6 – 8.2 seconds and writing the corresponding network requires 1.3 – 1.8 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 run-time versus computational resources trade-off for these problems will help biologists choose the optimal trade-off for their specific needs. We compute the metrics defined in Equation 2.2 and use the run-time of our optimized sequential implementation as T_{seq} in all the cases, since it has been established as the faster sequential implementation in the previous section.

Strong Scaling for Small Data sets

Since the sequential run-time of our implementation for *D1* 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 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 5.5a with the 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 $m = 1,000$. 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 $m = 1,000$. The consensus clustering task takes less than one second in all the cases.

Effect of Weighted Distribution of Candidate Splits: Since learning the modules is the most time consuming task for all the data sets in our experiments, we first investigate the performance of weighted distribution of candidate splits in parallel that was described in subsubsection 11. For this purpose, we measure the total run-time for learning MoNets from these five data sets in parallel by varying the number of cores (p) from 2 to $1,024$ using

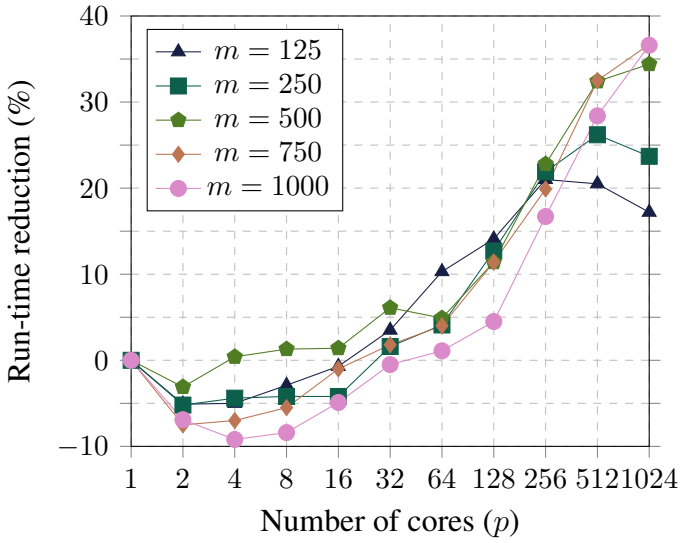


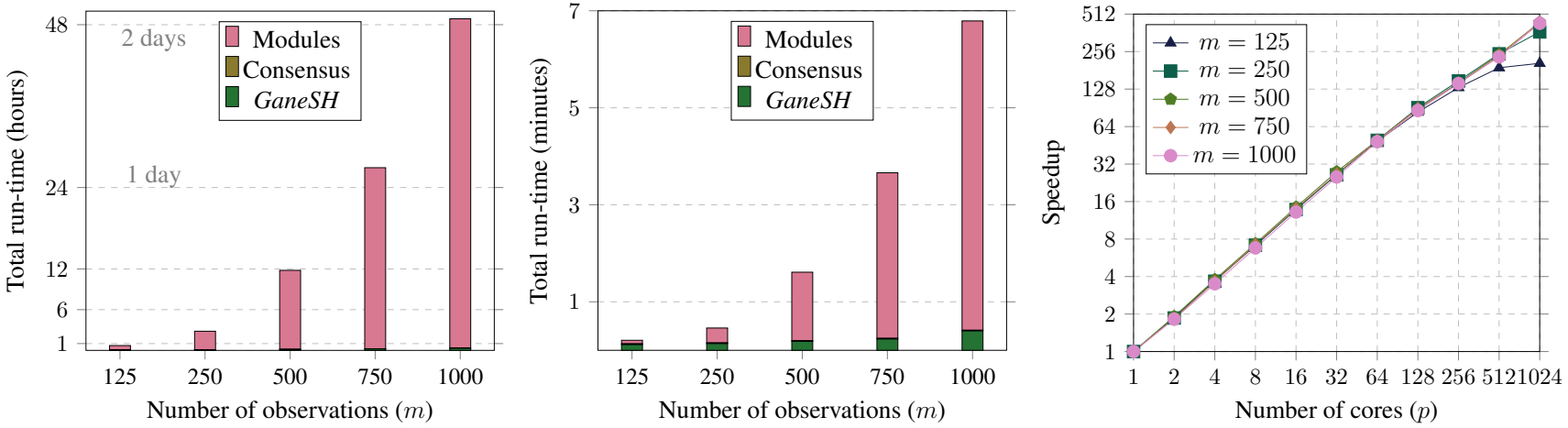
Figure 5.4: Plot of percentage reduction in the time required for learning MoNets for the five data sets using the weighted distribution scheme for candidate splits, as compared to the unweighted distribution scheme, on different number of cores.

both the unweighted and the weighted scheme. The percentage reduction in the run-time using the weighted scheme, as compared to the run-time using the unweighted scheme, is shown in Figure 5.4. We see that the run-time using the weighted scheme increases on smaller number of cores for all the data sets with a maximum increase of 9.2% for the data set with $m = 1000$ on 4 cores. When running on 64 cores or more, though, the run-time for all the data sets improves using the weighted scheme with a maximum reduction of 36.6% for both $m = 750$ and $m = 1000$ on 1024 cores. Therefore, we use the weighted distribution scheme in our final implementation and also for all the parallel scaling results presented in the remainder of this section.

Figure 5.5c shows the strong scaling speedup plots for the five 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.

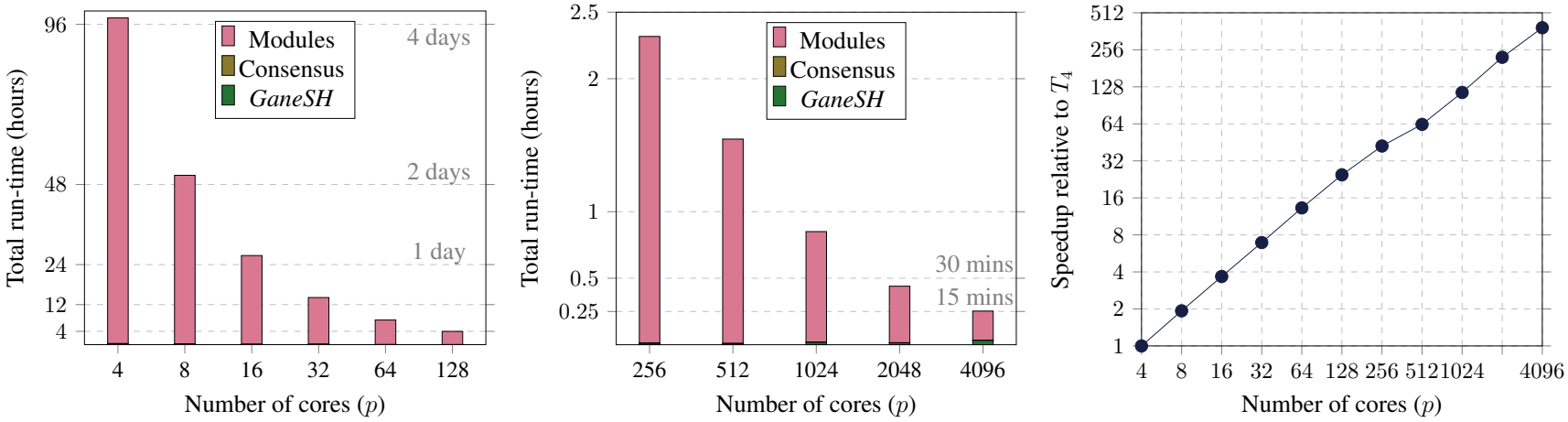
Our implementation achieves more than 48X speedup for all five data sets when using 64 cores, corresponding to scaling efficiency of 75% or more. However, the scaling tapers off as the number of cores is increased because of the load imbalance across processes in computing posterior probabilities for all the candidate splits. Since the number of discrete sampling steps (S) required for each split can not be estimated a priori, the time required for the split computations, i.e., $O(Sm)$, varies significantly across processes – even with a distribution of splits weighted by m. We measure the load imbalance using Equation 2.6 by substituting the time taken on a process for the load on the process. 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 three bigger data sets achieve similar speedups in the range of 431.8 – 441.3X when $p = 1024$.

The time required for learning MoNets from the five data sets using 1024 cores is shown in Figure 5.5b. Our parallel implementation reduces the run-time for the three larger data sets from 11.8, 26.9, and 48.9 hours to 1.6, 3.7, and 6.8 minutes, respectively, while the learning is completed in less then 30 seconds for the two smaller data sets. Even though Figure 5.5b shows a higher percentage of run-time in the *GaneSH* task on 1024 cores, when compared to Figure 5.5a, more than 90% of the run-time is still spent in learning the modules from the three larger data sets.



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

Figure 5.5: Plots showing the scalability of our implementation for learning MoNets from data sets with different number of observations subsampled from *D1*.



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

Figure 5.6: Plots showing the run-times of our implementation for learning MoNets from complete *D1* using different number of cores and the corresponding relative speedup.

Relative Scaling for D1

We used our parallel implementation to construct MoNets from *D1*. Since sequentially constructing MoNets from this data set is expected to take almost two weeks, we used a minimum of 4 cores for these experiments and discuss relative speedup and efficiency (as per Equation 2.4) with respect to T_4 for this data set in order to conduct the experiments in a reasonable time-frame. 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 5.6c.

We show the run-times obtained from the executions using 128 cores and fewer in Figure 5.6a and those using 128 to 4096 cores in Figure 5.6b, 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 more than 4 days using $p = 4$ to less than 4 hours using $p = 128$ with a relative speedup of 24.6X and more than 75% 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 15.2 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 subsubsection 5.4.3, the relative speedup from $p = 4$ to $p = 4096$ is 387X corresponding to a relative efficiency of 37.8%. Nevertheless, to construct a MoNet in a computational biology pipeline, a run-time of 15.2 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 15 minutes and the ideal possible run-time of 6 minutes (at 100% relative efficiency) for MoNet learning from data sets created by wet-lab biological experiments is immaterial, given that conducting these wet-lab experiments can take days.

Table 5.2: Parallel run-times for learning MoNets from *D2* using large number of cores and the corresponding relative speedup and efficiency.

Number of Cores (p)	Run-time (s)	Relative to T_{256}	
		Speedup	Efficiency (%)
256	109,489.8	1.0	100.0
512	60,187.8	1.8	91.0
1024	34,696.5	3.2	78.9
2048	18,091.5	6.1	75.6
4096	10,209.2	10.7	67.0

Relative Scaling for D2

We estimated, in subsubsection 5.4.2, that our optimized sequential implementation will require approximately 14 months for learning a MoNet for *D2*, 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 5.2 shows the time required for learning networks for *D2*. 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 2.8 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 *D1* is 57.2% on 4096 cores in subsubsection 5.4.3, the corresponding relative scaling efficiency for *D2* increases to 67%.

5.5 Summary of Contributions

In this chapter, we introduce a parallel, distributed-memory based approach for constructing large MoNets efficiently. We focused on parallelizing *Lemon-Tree*, which is more widely used for this purpose. We presented 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 constructed genome-scale gene networks for *S. cerevisiae* and *A. thaliana* with $5,716$ and $18,373$ genes, respectively. Our parallel implementation can construct a MoNet for *S. cerevisiae* in 15.2 minutes and for *A. thaliana* in 2.8 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.

CHAPTER 6

CONCLUSIONS

Motivated by the current need for interpretable alternatives to DL models, we investigated BNs and found that the lack of scalable methods for learning them is a major hurdle in ascertaining their feasibility for the purpose. Therefore, in this dissertation, we developed parallel algorithms for a variety of BN structure learning methods – including a method for learning a specialization of BNs known as MoNets. Specifically, we focused on parallelizing popular learning algorithms using which it was heretofore not feasible to learn large-scale networks.

In chapter 3, we presented a framework for parallelizing multiple *constraint-based* BN structure learning algorithms. Using the framework, we developed theoretically efficient parallel algorithms for five different algorithms that are categorized as *local-to-global*: *GS*, *IAMB*, *Inter-IAMB*, *MMPC*, and *SI-HITON*. We also improved the practical performance of these parallel algorithms by optimizing the CI testing and load balancing. The algorithms implemented using this framework are able to construct genome-scale gene regulatory networks for two model organisms, *S. cerevisiae* and *A. thaliana*, in less than 38 seconds on 2048 cores. This corresponds to a strong scaling speedup and efficiency of up to $1,745$X and 85.2%. Moreover, the parallel implementations show similar scalability for learning networks with an even larger number of variables from simulated data sets.

In chapter 4, we extended our framework for parallelizing *constraint-based* algorithms to support *global-search* algorithms. We used this extended framework to propose two different parallel algorithms for *PC-stable* – the most popular *global-search* algorithm. For implementing these algorithms, we proposed a novel algorithm-specific load-balancing technique and also reused some of the optimizations discussed in chapter 3 that helped these implementations outperform an optimized version of the previous best approach. The im-

plementations of our parallel algorithms are able to reduce the time taken by *PC-stable* to learn gene regulatory network for *S. cerevisiae* to 5.9 minutes using 4096 cores, as compared to a sequential run-time of 88.3 hours using our optimized implementation and more than seven days using the previous approaches.

In chapter 5, we focused on *score-based* methods and proposed a distributed memory parallelization for the construction of MoNets – a parameter-sharing specialization of BNs. We proposed the first parallel approach for the popular *Lemon-Tree* method by developing parallel algorithms for its different components. Our parallel implementation learns gene regulatory networks for *S. cerevisiae* and *A. thaliana* in 15.2 minutes and 2.8 hours using 4096 cores, as compared to an estimated 49 and 1561 days using the previous sequential implementation of *Lemon-Tree*, respectively.

The open-source implementations of our parallel algorithms are agnostic to the underlying application area. They can be used for any of the existing applications of the corresponding sequential algorithms, such as modeling climate networks [141, 142], identifying network failure causes [143], anti-discrimination learning [15, 16], detection of errors in quantum computing systems [144], and many more. However, we chose to demonstrate their scalability by constructing gene networks in all the cases because of the following two reasons. First, both BNs and MoNets have been successfully used in recovering gene networks from gene expression data sets in the past [18, 61]. Second, it is a rich application area that has big data sets and the need for constructing large-scale networks. In our experiments, we considered the genome-scale version of this problem, i.e., learning networks with tens of thousands of genes using thousands of gene expression studies that allowed us to demonstrate the performance of our algorithms. This will greatly aid biologists in their research on gene networks because it can potentially save weeks of their time during the iterative search for the optimal parameters to construct a network that best approximates the biological truth.

Our parallel algorithms are able to accomplish expeditious learning of high-dimensional

networks from large data sets using thousands of cores, a significant improvement over the prior state-of-the-art works. We expect that this will enable the adoption of BNs and MoNets in other fields related to the ones they have been successfully used in the past, e.g., single cell genomics [145] as well as applications that use parameter-sharing variations of BNs other than MoNets, where the time required for learning the networks has been a deterrent thus far. Ultimately, we hope that the work done as part of this dissertation will enable further investigation into the viability of BNs as interpretable ML models that can be used to replace black-box models for making high-stakes decisions.

6.1 Scope for Future Research

The parallel algorithms proposed in this dissertation need the complete data set to be available on every processor. While this causes duplication of data within the same node, it avoids the use of hybrid shared and distributed memory programming. This duplication is a non-issue because the learning algorithms are compute-bound due to their NP-hard nature, and the data sets are relatively small compared to the available memory size – the largest data set that we used for learning in our experiments is still only 785 MB. To scale to bigger data sets, the ability to query distributed data sets is required – a problem that is similar to parallel Online Analytical Processing (OLAP) [50]. Multiple previous works have developed solutions for executing OLAP queries over distributed data [146, 147]. These can be used as a promising starting point for future research into using distributed data sets for learning BNs.

The performance of our implementations can be enhanced using different approaches. Our implementations of the algorithms utilize CPUs for all the computations. One possible direction for future investigation is the use of accelerators for the computationally-intensive parts of the algorithms. There has been some prior work on accelerating CI testing using GPUs. However, the proposed solutions are limited to conducting tests of small conditioning set sizes [98] or only work for particular types of data sets [100] and additional re-

search is required for more general-purpose solutions. Dynamic load-balancing schemes, like work stealing and master-worker paradigm, can also be explored for improving the run-times of the implementations further.

Potential enhancements to our open-source software packages include implementation of more CI tests for both discrete as well as continuous data sets, incorporation of the state-of-the-art *score-based* approaches for BN learning, and development of a parallel version of *GENOMICA* for learning MoNets. Finally, while our *C++* implementations ensure optimal performance in practice, the software packages can be made more accessible by interfacing them with popular interpreted languages like *Python* and *R*. This will enable both ML as well as application-domain researchers to quickly learn BNs using our algorithms and evaluate their efficacy, while making minimal changes to their data analysis pipelines.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," *Nature Machine Intelligence*, vol. 1, no. 5, pp. 206–215, 2019.

[3] C. Rudin, C. Wang, and B. Coker, "The age of secrecy and unfairness in recidivism prediction," *Harvard Data Science Review*, vol. 2, no. 1, 2020.

[4] J. R. Zech, M. A. Badgeley, M. Liu, A. B. Costa, J. J. Titano, and E. K. Oermann, "Variable generalization performance of a deep learning model to detect pneumonia in chest radiographs: A cross-sectional study," *PLoS medicine*, vol. 15, no. 11, e1002683, 2018.

[5] M. McGough, "How bad is sacramento's air, exactly? google results appear at odds with reality, some say," *Sacramento Bee*, vol. 7, 2018.

[6] D. Martens, B. Baesens, T. Van Gestel, and J. Vanthienen, "Comprehensible credit scoring models using rule extraction from support vector machines," *European journal of operational research*, vol. 183, no. 3, pp. 1466–1476, 2007.

[7] B. Goodman and S. Flaxman, "European union regulations on algorithmic decision-making and a "right to explanation"," *AI magazine*, vol. 38, no. 3, pp. 50–57, 2017.

[8] D. Gunning and D. W. Aha, "Darpa's explainable artificial intelligence program," *AI Magazine*, vol. 40, no. 2, pp. 44–58, 2019.

[9] C. Yuan, H. Lim, and T.-C. Lu, "Most relevant explanation in bayesian networks," *Journal of Artificial Intelligence Research*, vol. 42, pp. 309–352, 2011.

[10] J. Pearl, "Bayesian networks: A model of self-activated memory for evidential reasoning," in *Proceedings of the 7th conference of the Cognitive Science Society, University of California, Irvine, CA, USA*, 1985, pp. 15–17.

[11] E. Kyrimi, S. McLachlan, K. Dube, M. R. Neves, A. Fahmi, and N. Fenton, "A comprehensive scoping review of bayesian networks in healthcare: Past, present and future," *arXiv preprint arXiv:2002.08627*, 2020.

[12] C. S. Vlek, H. Prakken, S. Renooij, and B. Verheij, "A method for explaining bayesian networks for legal evidence with scenarios," *Artificial Intelligence and Law*, vol. 24, no. 3, pp. 285–324, 2016.

[13] F. Taroni, A. Biedermann, S. Bozza, P. Garbolino, and C. Aitken, *Bayesian networks for probabilistic inference and decision analysis in forensic science*. John Wiley & Sons, 2014.

[14] A. Beresniak, E. Bertherat, W. Perea, G. Soga, R. Souley, D. Dupont, and S. Hugonnet, "A bayesian network approach to the study of historical epidemiological databases: Modelling meningitis outbreaks in the niger," *Bulletin of the World Health Organization*, vol. 90, 412–417a, 2012.

[15] L. Zhang, Y. Wu, and X. Wu, "Achieving non-discrimination in data release," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1335–1344.

[16] L. Zhang and X. Wu, "Anti-discrimination learning: A causal modeling-based framework," *International Journal of Data Science and Analytics*, vol. 4, no. 1, pp. 1–16, 2017.

[17] N. Friedman, M. Linial, I. Nachman, and D. Pe'er, "Using bayesian networks to analyze expression data," *Journal of computational biology*, vol. 7, no. 3-4, pp. 601–620, 2000.

[18] S. Imoto, T. Higuchi, T. Goto, K. Tashiro, S. Kuhara, and S. Miyano, "Combining microarrays and biological knowledge for estimating gene networks via bayesian networks," *Journal of bioinformatics and computational biology*, vol. 2, no. 01, pp. 77–98, 2004.

[19] J. Ramsey, M. Glymour, R. Sanchez-Romero, and C. Glymour, "A million variables and more: The fast greedy equivalence search algorithm for learning high-dimensional graphical causal models, with an application to functional magnetic resonance images," *International journal of data science and analytics*, vol. 3, no. 2, pp. 121–129, 2017.

[20] X. Sun, J. Dai, P. Liu, A. Singhal, and J. Yen, "Using bayesian networks for probabilistic identification of zero-day attack paths," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 10, pp. 2506–2521, 2018.

[21] D. M. Chickering, D. Heckerman, and C. Meek, "Large-sample learning of bayesian networks is np-hard," *Journal of Machine Learning Research*, vol. 5, no. Oct, pp. 1287–1330, 2004.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[24] M. Scutari, "Learning bayesian networks with the bnlearn r package," *Journal of Statistical Software*, vol. 35, no. i03, 2010.

[25] R. Scheines, P. Spirtes, C. Glymour, C. Meek, and T. Richardson, "The tetrad project: Constraint based aids to causal model specification," *Multivariate Behavioral Research*, vol. 33, no. 1, pp. 65–117, 1998.

[26] M. Kalisch, M. Mächler, D. Colombo, M. H. Maathuis, and P. Bühlmann, "Causal inference using graphical models with the r package pcalg," *Journal of Statistical Software*, vol. 47, no. 11, pp. 1–26, 2012.

[27] M. Scutari, "Bayesian network constraint-based structure learning algorithms: Parallel and optimized implementations in the bnlearn r package," *Journal of Statistical Software*, vol. 77, no. i02, 2017.

[28] A. Srivastava, S. Chockalingam, and S. Aluru, "A Parallel Framework for Constraint-based Bayesian Network Learning via Markov Blanket Discovery," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society, 2020, pp. 74–88.

[29] A. Srivastava, S. Chockalingam, M. Aluru, and S. Aluru, "Parallel Construction of Module Networks," in *2021 SC21: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, 2021.

[30] A. Srivastava, *ramBLe - A Parallel Framework for Bayesian Learning*, https://github.com/asrivast28/ramBLe, 2020.

[31] A. Srivastava, *ParsiMoNe - Parallel Construction of Module Networks*, https://github.com/asrivast28/ParsiMoNe, 2021.

[32] D. Heckerman, "A tutorial on learning with bayesian networks," *Innovations in Bayesian networks*, pp. 33–82, 2008.

[33] D. Chickering, D. Geiger, and D. Heckerman, "Learning bayesian networks: Search methods and experimental results," in *proceedings of fifth conference on artificial intelligence and statistics*, 1995, pp. 112–128.

[34] D. M. Chickering, "Learning equivalence classes of bayesian-network structures," *Journal of machine learning research*, vol. 2, no. Feb, pp. 445–498, 2002.

[35] T. Hrycej, "Gibbs sampling in bayesian networks," *Artificial Intelligence*, vol. 46, no. 3, pp. 351–363, 1990.

[36] D. Madigan, J. York, and D. Allard, "Bayesian graphical models for discrete data," *International Statistical Review/Revue Internationale de Statistique*, pp. 215–232, 1995.

[37] C. Meek, "Causal inference and causal explanation with background knowledge," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, 1995, pp. 403–410.

[38] P. Spirtes and C. Meek, "Learning bayesian networks with discrete variables from data.," in *KDD*, vol. 1, AAAI Press, 1995, pp. 294–299.

[39] D. Colombo and M. H. Maathuis, "Order-independent constraint-based causal structure learning," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3741–3782, 2014.

[40] D. Margaritis and S. Thrun, "Bayesian network induction via local neighborhoods," in *Advances in neural information processing systems*, MIT press, 2000, pp. 505–511.

[41] I. Tsamardinos, C. F. Aliferis, A. R. Statnikov, and E. Statnikov, "Algorithms for large scale markov blanket discovery.," in *FLAIRS conference*, vol. 2, AAAI Press, 2003, pp. 376–380.

[42] I. Tsamardinos, L. E. Brown, and C. F. Aliferis, "The max-min hill-climbing bayesian network structure learning algorithm," *Machine learning*, vol. 65, no. 1, pp. 31–78, 2006.

[43] C. F. Aliferis, I. Tsamardinos, and A. Statnikov, "Hiton: A novel markov blanket algorithm for optimal variable selection," in *AMIA annual symposium proceedings*, American Medical Informatics Association, 2003, p. 21.

[44] C. F. Aliferis, A. Statnikov, I. Tsamardinos, S. Mani, and X. D. Koutsoukos, "Local causal and markov blanket induction for causal discovery and feature selection for classification part i: Algorithms and empirical evaluation.," *Journal of Machine Learning Research*, vol. 11, no. 1, 2010.

[45] J. M. Peña, J. Björkegren, and J. Tegnér, "Scalable, efficient and correct learning of markov boundaries under the faithfulness assumption," in *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, Springer, 2005, pp. 136–147.

[46] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman, *Causation, prediction, and search*. MIT press, 2000.

[47] R. E. Neapolitan, *Learning bayesian networks*. Pearson Prentice Hall Upper Saddle River, NJ, 2004, vol. 38.

[48] S. Karan, M. Eichhorn, B. Hurlburt, G. Iraci, and J. Zola, "Fast counting in machine learning applications," in *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'18, AUAI Press, 2018, pp. 540–549.

[49] S. Misra, M. Vasimuddin, K. Pamnany, S. P. Chockalingam, Y. Dong, M. Xie, M. R. Aluru, and S. Aluru, "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, 2014, pp. 461–472.

[50] B. S. Anderson and A. W. Moore, "Adtrees for fast counting and for fast learning of association rules.," in *KDD*, AAAI Press, 1998, pp. 134–138.

[51] J. Xu and C. R. Shelton, "Intrusion detection using continuous time bayesian networks," *Journal of Artificial Intelligence Research*, vol. 39, pp. 745–774, 2010.

[52] K. C. Baumgartner, S. Ferrari, and C. G. Salfati, "Bayesian network modeling of offender behavior for criminal profiling," in *Proceedings of the 44th IEEE Conference on Decision and Control*, IEEE, 2005, pp. 2702–2709.

[53] S. M. Mahoney and K. B. Laskey, "Network engineering for complex belief networks," in *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*, 1996, pp. 389–396.

[54] D. Koller, "Probabilistic relational models," in *International Conference on Inductive Logic Programming*, Springer, 1999, pp. 3–13.

[55] D. Pe'er, A. Regev, G. Elidan, and N. Friedman, "Inferring subnetworks from perturbed expression profiles," *Bioinformatics*, vol. 17, no. suppl_1, S215–S224, 2001.

[56] R. S. Niculescu, T. M. Mitchell, R. B. Rao, K. P. Bennett, and E. Parrado-Hernández, "Bayesian network learning with parameter constraints.," *Journal of machine learning research*, vol. 7, no. 7, 2006.

[57] E. Segal, D. Pe'er, A. Regev, D. Koller, and N. Friedman, "Learning module networks," in *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, 2003, pp. 525–534.

[58] E. Segal, D. Pe'er, A. Regev, D. Koller, N. Friedman, and T. Jaakkola, "Learning module networks.," *Journal of Machine Learning Research*, vol. 6, no. 4, 2005.

[59] D. Koller and A. Pfeffer, "Object-oriented bayesian networks," in *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, 1997, pp. 302–313.

[60] E. Gyftodimos and P. A. Flach, "Hierarchical bayesian networks: An approach to classification and learning for structured data," in *Hellenic Conference on Artificial Intelligence*, Springer, 2004, pp. 291–300.

[61] E. Segal, M. Shapira, A. Regev, D. Pe'er, D. Botstein, D. Koller, and N. Friedman, "Module networks: Identifying regulatory modules and their condition-specific regulators from gene expression data," *Nature genetics*, vol. 34, no. 2, pp. 166–176, 2003.

[62] E. Segal, N. Friedman, D. Koller, and A. Regev, "A module map showing conditional activity of expression modules in cancer," *Nature genetics*, vol. 36, no. 10, pp. 1090–1098, 2004.

[63] E. Segal, N. Friedman, N. Kaminski, A. Regev, and D. Koller, "From signatures to models: Understanding cancer using microarrays," *Nature genetics*, vol. 37, no. 6, S38–S45, 2005.

[64] E. Segal, C. B. Sirlin, C. Ooi, A. S. Adler, J. Gollub, X. Chen, B. K. Chan, G. R. Matcuk, C. T. Barry, H. Y. Chang, *et al.*, "Decoding global gene expression programs in liver cancer by noninvasive imaging," *Nature biotechnology*, vol. 25, no. 6, pp. 675–680, 2007.

[65] X. Lu, X. Li, P. Liu, X. Qian, Q. Miao, and S. Peng, "The integrative method based on the module-network for identifying driver genes in cancer subtypes," *Molecules*, vol. 23, no. 2, p. 183, 2018.

[66] D. Pe'er, A. Tanay, A. Regev, and T. Jaakkola, "Minreg: A scalable algorithm for learning parsimonious regulatory networks in yeast and mammals.," *Journal of Machine Learning Research*, vol. 7, no. 2, 2006.

[67] A. Battle, E. Segal, and D. Koller, "Probabilistic discovery of overlapping cellular processes and their regulation," *Journal of Computational Biology*, vol. 12, no. 7, pp. 909–927, 2005.

[68] F. Markowetz and R. Spang, "Inferring cellular networks–a review," *BMC bioinformatics*, vol. 8, no. 6, pp. 1–17, 2007.

[69] B. Berger, J. Peng, and M. Singh, "Computational solutions for omics data," *Nature reviews genetics*, vol. 14, no. 5, pp. 333–346, 2013.

[70] A. Ahmad and H. Fröhlich, "Integrating heterogeneous omics data via statistical inference and learning techniques," *Genomics and Computational Biology*, vol. 2, no. 1, e32–e32, 2016.

[71] E. Bonnet, L. Calzone, and T. Michoel, "Integrative multi-omics module network inference with lemon-tree," *PLoS Comput Biol*, vol. 11, no. 2, e1003983, 2015.

[72] P. Kraaijeveld, M. J. Druzdzel, A. Onisko, and H. Wasyluk, "Genierate: An interactive generator of diagnostic bayesian network models," in *Proc. 16th Int. Workshop Principles Diagnosis*, Citeseer, 2005, pp. 175–180.

[73] D. Kasper, G. Weidl, T. Dang, G. Breuel, A. Tamke, A. Wedel, and W. Rosenstiel, "Object-oriented bayesian networks for detection of lane change maneuvers," *IEEE Intelligent Transportation Systems Magazine*, vol. 4, no. 3, pp. 19–31, 2012.

[74] V. J. Shute, M. Ventura, M. Bauer, and D. Zapata-Rivera, "Melding the power of serious games and embedded assessment to monitor and foster learning," *Serious games: Mechanisms and effects*, vol. 2, pp. 295–321, 2009.

[75] Z. M. Hira and D. F. Gillies, "A review of feature selection and feature extraction methods applied on microarray data," *Advances in bioinformatics*, vol. 2015, 2015.

[76] T. L. Griffiths, "Causes, coincidences, and theories," Ph.D. dissertation, stanford university, 2004.

[77] T. L. Griffiths and J. B. Tenenbaum, "Theory-based causal induction.," *Psychological review*, vol. 116, no. 4, p. 661, 2009.

[78] S. Ma, J. Li, L. Liu, and T. D. Le, "Mining combined causes in large data sets," *Knowledge-Based Systems*, vol. 92, pp. 104–111, 2016.

[79] T. Michoel, S. Maere, E. Bonnet, A. Joshi, Y. Saeys, T. Van den Bulcke, K. Van Leemput, P. Van Remortel, M. Kuiper, K. Marchal, *et al.*, "Validating module network learning algorithms using simulated data," *BMC bioinformatics*, vol. 8, no. 2, pp. 1–15, 2007.

[80] A. Joshi, R. De Smet, K. Marchal, Y. Van de Peer, and T. Michoel, "Module networks revisited: Computational assessment and prioritization of model predictions," *Bioinformatics*, vol. 25, no. 4, pp. 490–496, 2009.

[81] E. Azizi, E. Airoldi, and J. Galagan, "Learning modular structures from network data and node variables," in *International conference on machine learning*, PMLR, 2014, pp. 1440–1448.

[82] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[83] A. Joshi, Y. Van de Peer, and T. Michoel, "Analysis of a gibbs sampler method for model-based clustering of gene expression data," *Bioinformatics*, vol. 24, no. 2, pp. 176–183, 2008.

[84] T. Michoel and B. Nachtergaele, "Alignment and integration of complex networks by hypergraph-based spectral clustering," *Physical Review E*, vol. 86, no. 5, p. 056 111, 2012.

[85] S. Aluru, "Teaching parallel computing through parallel prefix," in *The International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, p. 7.

[86] P. Flick, *mxx*, https://github.com/patflick/mxx, 2015.

[87] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the mpi standard," *Communications of the ACM*, vol. 18, 1995.

[88] M. Koivisto and K. Sood, "Exact bayesian structure discovery in bayesian networks," *Journal of Machine Learning Research*, vol. 5, no. May, pp. 549–573, 2004.

[89] T. Silander and P. Myllymäki, "A simple approach for finding the globally optimal bayesian network structure," in *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence*, ser. UAI'06, Cambridge, MA, USA: AUAI Press, 2006, pp. 445–452, ISBN: 0974903922.

[90] Y. Tamada, S. Imoto, and S. Miyano, "Parallel algorithm for learning optimal bayesian network structure," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2437–2459, 2011.

[91] O. Nikolova, J. Zola, and S. Aluru, "Parallel globally optimal structure learning of bayesian networks," *Journal of Parallel and Distributed Computing*, vol. 73, no. 8, pp. 1039–1048, 2013.

[92] Y. Tamada, "Memory efficient parallel algorithm for optimal dag structure search using direct communication," *Journal of Parallel and Distributed Computing*, vol. 119, pp. 27–35, 2018.

[93] O. Nikolova and S. Aluru, "Parallel bayesian network structure learning with application to gene networks," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, pp. 1–9.

[94] A. L. Madsen, F. Jensen, A. Salmerón, H. Langseth, and T. D. Nielsen, "A parallel algorithm for bayesian network structure learning from large data sets," *Knowledge-Based Systems*, vol. 117, pp. 46–55, 2017.

[95] D. Stinson, *Combinatorial designs: constructions and analysis*. Springer Science & Business Media, 2007.

[96] T. Le, T. Hoang, J. Li, L. Liu, H. Liu, and S. Hu, "A fast pc algorithm for high dimensional causal discovery with multi-core pcs," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 16, no. 5, pp. 1483–1495, 2016.

[97] C. Schmidt, J. Huegle, P. Bode, and M. Uflacker, "Load-balanced parallel constraint-based causal structure learning on multi-core systems for high-dimensional data," in *The 2019 ACM SIGKDD Workshop on Causal Discovery*, PMLR, 2019, pp. 59–77.

[98] C. Schmidt, J. Huegle, and M. Uflacker, "Order-independent constraint-based causal structure learning for gaussian distribution models using gpus," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, ACM, 2018, pp. 1–10.

[99] C. Schmidt, J. Huegle, S. Horschig, and M. Uflacker, "Out-of-core gpu-accelerated causal structure learning," in *International Conference on Algorithms and Architectures for Parallel Processing*, Springer, 2019, pp. 89–104.

[100] B. Zarebavani, F. Jafarinejad, M. Hashemi, and S. Salehkaleybar, "Cupc: Cuda-based parallel pc algorithm for causal structure learning on gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 530–542, 2020.

[101] C. Hagedorn and J. Huegle, "Gpu-accelerated constraint-based causal structure learning for discrete data," in *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, SIAM, 2021, pp. 37–45.

[102] C. F. Aliferis, A. Statnikov, I. Tsamardinos, S. Mani, and X. D. Koutsoukos, "Local causal and markov blanket induction for causal discovery and feature selection for classification part ii: Analysis and extensions.," *Journal of Machine Learning Research*, vol. 11, no. 1, 2010.

[103] O. Nikolova and S. Aluru, "Parallel discovery of direct causal relations and markov boundaries with applications to gene networks," in *2011 International Conference on Parallel Processing*, IEEE, 2011, pp. 512–521.

[104] Y. Lu, X. Zhou, and C. Nardini, "Dissection of the module network implementation "lemontree": Enhancements towards applications in metagenomics and translation in autoimmune maladies," *Molecular BioSystems*, vol. 13, no. 10, pp. 2083–2091, 2017.

[105] A. Moirangthem, X. Wang, I. K. Yan, and T. Patel, "Network analyses–based identification of circular ribonucleic acid–related pathways in intrahepatic cholangiocarcinoma," *Tumor Biology*, vol. 40, no. 9, p. 1 010 428 318 795 761, 2018.

[106] F. A. Marchi, D. C. Martins, M. C. Barros-Filho, H. Kuasne, A. F. B. Lopes, H. Brentani, J. C. S. Trindade Filho, G. C. Guimarães, E. F. Faria, C. Scapulatempo-Neto, *et al.*, "Multidimensional integrative analysis uncovers driver candidates and biomarkers in penile carcinoma," *Scientific reports*, vol. 7, no. 1, pp. 1–11, 2017.

[107] E. Behdani and M. R. Bakhtiarizadeh, "Construction of an integrated gene regulatory network link to stress-related immune system in cattle," *Genetica*, vol. 145, no. 4, pp. 441–454, 2017.

[108] P. A. Alexandre, L. J. Kogelman, M. H. Santana, D. Passarelli, L. H. Pulz, P. Fantinato-Neto, P. L. Silva, P. R. Leme, R. F. Strefezzi, L. L. Coutinho, *et al.*, "Liver transcriptomic networks reveal main biological processes associated with feed efficiency in beef cattle," *BMC genomics*, vol. 16, no. 1, pp. 1–13, 2015.

[109] E. Kaitetzidou, J. Xiang, E. Antonopoulou, C. S. Tsigenopoulos, and E. Sarropoulou, "Dynamics of gene expression patterns during early development of the european seabass (dicentrarchus labrax)," *Physiological Genomics*, vol. 47, no. 5, pp. 158–169, 2015.

[110] S. Arhondakis, C. E. Bita, A. Perrakis, M. E. Manioudaki, A. Krokida, D. Kaloudas, and P. Kalaitzis, "In silico transcriptional regulatory networks involved in tomato fruit ripening," *Frontiers in plant science*, vol. 7, p. 1234, 2016.

[111] Y. Bai, L. Dougherty, L. Cheng, G.-Y. Zhong, and K. Xu, "Uncovering co-expression gene network modules regulating fruit acidity in diverse apples," *BMC genomics*, vol. 16, no. 1, pp. 1–16, 2015.

[112] V. Vermeirssen, I. De Clercq, T. Van Parys, F. Van Breusegem, and Y. Van de Peer, "Arabidopsis ensemble reverse-engineered gene regulatory network discloses interconnected transcription factors in oxidative stress," *The Plant Cell*, vol. 26, no. 12, pp. 4656–4679, 2014.

[113] L. Liu, W. Hu, C. Lai, H.-s. Jiang, W. Chen, W. Zheng, and Y. Zhang, "Parallel module network learning on distributed memory multiprocessors," in *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, IEEE, 2005, pp. 129–134.

[114] H. Jiang, C. Lai, W. Chen, Y. Chen, W. Hu, W. Zheng, and Y. Zhang, "Parallelization of module network structure learning and performance tuning on smp," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE, 2006, 8–pp.

[115] N. Noyes, K.-C. Cho, J. Ravel, L. J. Forney, and Z. Abdo, "Associations between sexual habits, menstrual hygiene practices, demographics and the vaginal microbiome as revealed by bayesian network analysis," *PloS one*, vol. 13, no. 1, pp. 1–25, 2018.

[116] D. V. Zhernakova, T. H. Le, A. Kurilshikov, B. Atanasovska, M. J. Bonder, S. Sanna, A. Claringbould, U. Võsa, P. Deelen, L. Franke, *et al.*, "Individual variations in cardiovascular-disease-related protein levels are driven by genetics and gut microbiome," *Nature genetics*, vol. 50, no. 11, pp. 1524–1532, 2018.

[117] O. Delaneau, M. Zazhytska, C. Borel, G. Giannuzzi, G. Rey, C. Howald, S. Kumar, H. Ongen, K. Popadin, D. Marbach, *et al.*, "Chromatin three-dimensional interactions mediate genetic effects on gene expression.," *Science*, vol. 364, no. 6439, eaat8266, 2019.

[118] Z. Zheng, S. Hey, T. Jubery, H. Liu, Y. Yang, L. Coffey, C. Miao, B. Sigmon, J. C. Schnable, F. Hochholdinger, *et al.*, "Shared genetic control of root system architecture between zea mays and sorghum bicolor," *Plant Physiology*, vol. 182, no. 2, pp. 977–991, 2020.

[119] A. Cano, M. Gómez-Olmedo, and S. Moral, "A score based ranking of the edges for the pc algorithm," in *Proceedings of the Fourth European Workshop on Probabilistic Graphical Models*, 2008, pp. 41–48.

[120] PACE, *Partnership for an Advanced Computing Environment (PACE)*, 2017.

[121] K. Tchourine, C. Vogel, and R. Bonneau, "Condition-specific modeling of biophysical parameters advances inference of regulatory networks," *Cell reports*, vol. 23, no. 2, pp. 376–388, 2018.

[122] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: practice and experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[123] M. G. Norman and P. Thanisch, "Models of machines and computation for mapping in multicomputers," *ACM Computing Surveys (CSUR)*, vol. 25, no. 3, pp. 263–302, 1993.

[124] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell system technical journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[125] S. K. Sahni, "Algorithms for scheduling independent tasks," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 116–127, 1976.

[126] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM Journal on Computing*, vol. 7, no. 1, pp. 1–17, 1978.

[127] M. H. Maathuis, D. Colombo, M. Kalisch, and P. Bühlmann, "Predicting causal effects in large-scale systems from observational data," *Nature methods*, vol. 7, no. 4, pp. 247–248, 2010.

[128] T. D. Le, L. Liu, A. Tsykin, G. J. Goodall, B. Liu, B.-Y. Sun, and J. Li, "Inferring microrna–mrna causal regulatory relationships from expression data," *Bioinformatics*, vol. 29, no. 6, pp. 765–771, 2013.

[129] J. Zhang, T. D. Le, L. Liu, B. Liu, J. He, G. J. Goodall, and J. Li, "Inferring condition-specific mirna activity from matched mirna and mrna expression data," *Bioinformatics*, vol. 30, no. 21, pp. 3070–3077, 2014.

[130] D. Marbach, J. C. Costello, R. Küffner, N. M. Vega, R. J. Prill, D. M. Camacho, K. R. Allison, M. Kellis, J. J. Collins, and G. Stolovitzky, "Wisdom of crowds for robust gene network inference," *Nature methods*, vol. 9, no. 8, pp. 796–804, 2012.

[131] T. D. Le, T. Xu, L. Liu, H. Shu, T. Hoang, and J. Li, "Parallelpc: An r package for efficient causal exploration in genomic data," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Springer, 2018, pp. 207–218.

[132] K. A. Heller and Z. Ghahramani, "Bayesian hierarchical clustering," in *Proceedings of the 22nd international conference on Machine learning*, 2005, pp. 297–304.

[133] L. Gherardi, D. Brugali, and D. Comotti, "A java vs. c++ performance evaluation: A 3d modeling benchmark," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2012, pp. 161–172.

[134] S. Sharma, *Performance comparison of java and c++ when sorting integers and writing/reading files.* 2019.

[135] H. Heyman and L. Brandefelt, *A comparison of performance & implementation complexity of multithreaded applications in rust, java and c++*, 2020.

[136] H. Bauke and S. Mertens, "Random numbers for large-scale distributed monte carlo simulations," *Physical Review E*, vol. 75, no. 6, p. 066 701, 2007.

[137] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.

[138] S. D. Shapira, I. Gat-Viks, B. O. Shum, A. Dricot, M. M. de Grace, L. Wu, P. B. Gupta, T. Hao, S. J. Silver, D. E. Root, *et al.*, "A physical and regulatory map of host-influenza interactions reveals pathways in h1n1 infection," *Cell*, vol. 139, no. 7, pp. 1255–1267, 2009.

[139] V. Vermeirssen, A. Joshi, T. Michoel, E. Bonnet, T. Casneuf, and Y. Van de Peer, "Transcription regulatory networks in caenorhabditis elegans inferred through reverse-engineering of gene expression profiles constitute biological hypotheses for metazoan development," *Molecular BioSystems*, vol. 5, no. 12, pp. 1817–1830, 2009.

[140] N. Novershtern, Z. Itzhaki, O. Manor, N. Friedman, and N. Kaminski, "A functional and regulatory map of asthma," *American journal of respiratory cell and molecular biology*, vol. 38, no. 3, pp. 324–336, 2008.

[141] M. Kretschmer, J. Cohen, V. Matthias, J. Runge, and D. Coumou, "The different stratospheric influence on cold-extremes in eurasia and north america," *npj Climate and Atmospheric Science*, vol. 1, no. 1, pp. 1–10, 2018.

[142] S. Samarasinghe, M. McGraw, E. Barnes, and I. Ebert-Uphoff, "A study of links between the arctic and the midlatitude jet stream using granger and pearl causality," *Environmetrics*, vol. 30, no. 4, e2540, 2019.

[143] S. Kobayashi, K. Otomo, K. Fukuda, and H. Esaki, "Mining causality of network events in log data," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 53–67, 2017.

[144] M. Sarovar, T. Proctor, K. Rudinger, K. Young, E. Nielsen, and R. Blume-Kohout, "Detecting crosstalk errors in quantum information processors," *Quantum*, vol. 4, p. 321, 2020.

[145] O. Stegle, S. A. Teichmann, and J. C. Marioni, "Computational and analytical challenges in single-cell transcriptomics," *Nature Reviews Genetics*, vol. 16, no. 3, pp. 133–145, 2015.

[146] A. Cuzzocrea, R. Moussa, and G. Xu, "Olap*: Effectively and efficiently supporting parallel olap over big data," in *International Conference on Model and Data Engineering*, Springer, 2013, pp. 38–49.

[147] F. Dehne and H. Zaboli, "Parallel real-time olap on multi-core processors," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, IEEE, 2012, pp. 588–594.