Implementation and Evaluation of a Parallel Algorithm for Structure Learning in Bayesian Networks

Huining Deng (huining.deng@helsinki.fi)

Master's thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, April 22, 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Huining Deng (huining.deng@helsinki.fi)

Työn nimi — Arbetets titel — Title

Implementation and Evaluation of a Parallel Algorithm for Structure Learning in Bayesian Networks

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	April 22, 2015	57

Tiivistelmä — Referat — Abstract

This thesis is about learning the globally optimal Bayesian network structure from fully observed dataset, by using score-based method. This structure learning problem is NP-hard, and has attracted the attention of many researchers. We first introduce the necessary background of the problem, then review various score-based methods and algorithms proposed in solving the problem. Parallelization has come under the spotlight during recent years, as it can utilize shared memory and computing power of multi-core supercomputers or computer clusters. We implemented a parallel algorithm Para-OS [TIM11], which is based on dynamic programming. Experiments were performed in order to evaluate the algorithm. We also propose an improved version of Para-OS, which separates the scoring phase totally from the learning phase, performs score pruning by using Sparse Parent Graph, in addition largely reduces the communication between processors. Empirical results shows the new version saves memory comparing to Para-OS, and provides good runtime with multi-treading.

Avainsanat — Nyckelord — Keywords
Bayesian networks, Parallel algorithms, Bayesian network models

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Background **3**
2.1 Examples of Bayesian Network 3
2.2 The Dual Nature of Bayesian Network 4

3 Structure Learning in Bayesian network **6**
3.1 Score-based Learning 6
3.2 Scoring Functions 7
3.3 The BDeu Scoring Function 8
3.4 Search Strategies 10
3.4.1 Approximate Methods 10
3.4.2 Optimal Methods and Dynamic Programming 11
3.4.3 Parallelized Methods 16

4 The Parallelized Algorithm Para-OS **17**
4.1 Split the Calculations 18
4.2 Calculation in a Single Processor 20
4.3 The Mapping Function 22
4.4 New Para-OS 24
4.4.1 Sparse Parent Graph 24
4.4.2 Parallelization 26

5 Implementation Process **26**
5.1 Using AD-tree to Collect Counts 27
5.2 Parallelization in MPI 30
5.3 Parallelization in OpenMP 32
5.4 Deciding Processor Sender-Receiver Pairs 33

6 Performance Evaluation **36**
6.1 Para-OS 37
6.1.1 σ Evaluation 38
6.1.2 Scalability Evaluation 38
6.1.3 Runtime Evaluation 40
6.1.4 Memory Consumption Evaluation 41
6.2 New Para-OS 43
6.2.1 Runtime Evaluation 43
6.2.2 Memory Consumption Evaluation 45
6.2.3 Various Datasets 48

7 Summary and Discussion **50**

8 Appendices **51**
8.1 Appendix 1: an example of full AD-tree 51

References **53**

1 Introduction

Bayesian networks (also called belief networks or Bayes networks), abbreviated as BNs, have gained increasing popularity as a powerful tool in a variety of fields involving data analysis, decision making, uncertainty reasoning, etc. BNs have been studied in basic science research as well as applied in practical real-world problem solving. In recent years, BNs have been employed to address a wide range of problems, such as computational biology [FLNP00], medical diagnosis [KJRSH97], speech recognition [ZR98], information retrieval [FDF95], semantic image understanding [LSS05], face recognition [Nef02], traffic prediction [HASL05], environmental modeling [Uus07], risk assessment and decision analysis [FN12], and forensic science [Bal11]. The name of BNs honors Thomas Bayes (1702 – 1761), who suggested Bayes' theorem for updating probabilities in the light of new evidence. Bayes' theorem is the foundation of BNs.

A Bayesian network is a graphical model that encodes the probability distribution over a set of random variables (or, for the sake of simplicity, we just say variables). To build a BN model for a studied domain, the development usually proceeds as follows [Kra09]. First, the model's objectives, as well as the scale of the system to be modeled, and other issues to be considered are defined. Second, a conceptual BN model is built by identifying the variables in the system and connecting the related variables with directed edges. The edges are established according to the causal relationships between the variables. Third, the BN model is parameterised, which means states and probabilities are assigned to each variable in the network. Fourth, the BN model is evaluated and tested. This process is done through discussions and cooperation of experts and stakeholders. The iterative process could be repeated several times before a useful BN model is constructed. At last the model can be put into use, for example, to perform scenario analysis, inference tasks, etc.

Once a BN model is established, one powerful usage of the model is to make predictions about how the world will behave. For example, a traffic forecasting system, JamBayes [HASL05], utilizes Bayesian network to model the probabilistic relationships between traffic congestion "hotspots" and contextual data such as the time of day, day of week, weather, holiday, etc. The model is capable of answering queries such as the expected time of a traffic jam in certain place, at what time will the traffic flow smoothly, etc.

In the second stage of a BN model development, a conceptual network structure is established. This BN structure can be constructed by hand, equipped with the knowledge from domain experts. However this task becomes difficult with large networks [KF09]. Sometimes human experts have to invest several months of time and effort into it, together with other tedious work like testing the results of queries produced by the model. In some other cases, there simply are not suitable experts. Therefore, it is

useful to employ machine learning methods to design algorithms that are capable of constructing Bayesian networks from empirical data.

Many algorithms have been developed over the years to tackle this model learning problem. The first goal of this thesis is to review different methods and algorithms in the field. In recent years, parallel algorithms have attracted the attention of researchers, as parallelization makes it possible for learning larger Bayesian networks. The second goal of the thesis is to provide an open-source implementation of the parallel algorithm *Parallel Optimal Search* (Para-OS) proposed in Tamada et al. [TIM11]. In order to verify the results from the original paper, the implemented algorithm is tested and evaluated.

The structure of the thesis is organized as follows. Chapter 2 provides the necessary background information on Bayesian networks. Chapter 3 discusses BN structure learning, with a focus on score-based learning. Chapter 4 explains the parallelized algorithm Para-OS in detail, and introduces the idea of a new version of Para-OS. Chapter 5 reports the implementation process of Para-OS and the new version. In Chapter 6, we record the testing process and present the comparison of the test results with the original paper, the comparison between Para-OS and the new version, etc. Chapter 7 is the conclusions of the thesis.

General Notation

$\mathcal{G}$	directed acyclic graph (DAG) of a BN
$\mathbf{D}$	the given dataset which is complete
$D_1, ..., D_m$	datapoints in $\mathbf{D}$
m	number of datapoints, i.e. $\lvert\mathbf{D}\rvert$
$\mathbf{V}$	set of variables/nodes in the domain: $\{X_1, ...X_n\}$
n	number of variables, i.e. $\lvert\mathbf{V}\rvert$
$X_1, ..., X_n$	variables in $\mathbf{V}$
PA_{X_i}	set of parent variables of X_i
$\mathbf{U}$	a variable set ($\mathbf{U} \subset \mathbf{V}$)
n_p	number of processors

Table 1: notation symbols used throughout the thesis

In this thesis project, we are focusing on discrete variables, which means each variable has finite states. The observed dataset is assumed to be complete, that is, there are no missing values. The dataset $\mathbf{D}$ contains datapoints $\{D_1, D_2, ..., D_m\}$, and each $D_i = \{d_1, d_2, ..., d_n\}$ is a value vector over a variable set $\mathbf{V} = \{X_1, X_2, ..., X_n\}$ with cardinality n. For example d_1 is an instantiation of variable X_1 in a data point D_i. The dataset is an $n * m$ matrix, where a row corresponds to a data point, and a column represents the value vector of a variable. Variables are represented by capital letters

(such as X_i), and variable sets or vectors are emphasized by bold capital letters (such as $\mathbf{V}$). Variables and nodes of the graph are interchangeable. Table 1 contains notation used through out the thesis.

2 Background

Uncertainty is not only the essence of romance, but also the innate character of the real world. Most AI application domains and reasoning tasks involve some degree of uncertainty. Uncertainty arises because it is hard to obtain complete and reliable information – our observations of the world are often partial, noisy (with errors), and inconsistent [KF09]. Probability, which express a degree of belief using a number between 0 and 1, allow us to measure our belief in the likelihood of an event. Hence uncertainty can be handled by probability theory, which is a tested theory and has been established as one of the foundations of the sciences [FN12].

Graph-based representations have been efficient for doing inferences in AI systems.

> "Graphical models are a marriage between probability theory and graph theory. They provide a natural tool for dealing with two problems that occur throughout applied mathematics and engineering – uncertainty and complexity..." – Jordan [Jor98].

A graphical model is an intuitive and compact representation of probabilistic relationships between variables. Graphical models have proved to work effectively on reasoning tasks given the available information. From another angle, these models help us to gain global insight based on local observations.

Bayesian networks belong to the family of graphical models. BNs are based on *directed acyclic graphs* (DAGs). The graphical nature of BN makes it easy to understand for people from a wide variety of backgrounds. BNs can display relationships between variables clearly and intuitively. As mentioned above, BNs can handle uncertainty though probability theory. Therefore, BNs provide a useful and robust tool in modeling. Below we introduce the nature of BNs through examples.

2.1 Examples of Bayesian Network

A classical example taken from Neapolitan [N$^+$04] helps to describe the use of BNs. Suppose we have the following observation: an individual's smoking history can directly affect whether this individual ends up having some diseases, for example bronchitis or lung cancer. A positive chest X-ray report is a direct result from examining a patient who has lung cancer, but bronchitis has no direct influence on a chest X-ray. Bronchitis and lung cancer patients both experience fatigue. We extract five variables from this

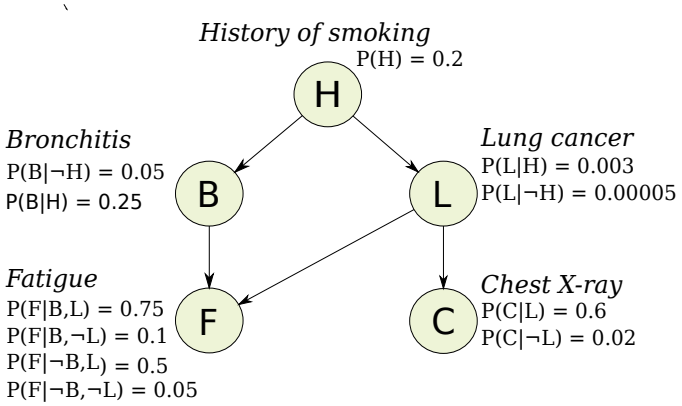


Figure 1: A Bayesian network of the smoking-diseases example. [N$^+$04]

situation: history of smoking (H), bronchitis (B), lung cancer (L), fatigue (F), and chest X-ray (C). All of them have binary values. The BN of this example is constructed as in Figure 1. We can see that the DAG is combined with conditional probability distributions (CPDs), in order to capture and quantify the dependencies between variables.

Bayesian networks can be used to model any situation for which we can find direct influences between variables. Now, we look at another BN example that describes a daily life situation. Sushi is one of my favorite foods, so sometimes I have a craving for it. If money is not tight that day, I will definitely buy sushi for dinner. However, there is a small chance I still buy sushi even though money is tight. Eating sushi often causes a mild stomach problem for me, because of the raw fish (but it does not diminish my love for sushi). If I buy sushi for dinner, I will be spared the cooking chore for that day. If I do not cook in the evening, I will most likely have time to play my favored online-game. There are six binary variables here: craving-for-sushi(C), sufficient-money(M), buy-sushi(B), stomach-problem(S), no-cooking(N) and play-online-game(G). The BN that models this situation, together with the associated CPDs, is depicted in Figure 2.

2.2 The Dual Nature of Bayesian Network

From the graphical structure of the above examples, we can see that a BN explicitly describes the direct dependencies between a set of variables $\mathbf{V} = \{X_1, ...X_n\}$. From another angle, we can also say that a BN represents a joint probability distribution over these variables. Therefore, a BN consists of the following two parts.

(i) a DAG, which contains a set of nodes and edges. Each node corresponds to a variable, and each edge displays a direct probability dependency between the two nodes it connects. An edge from node X_i to node X_j represents an assumption that there is a direct influence of X_i on X_j. We call X_i a

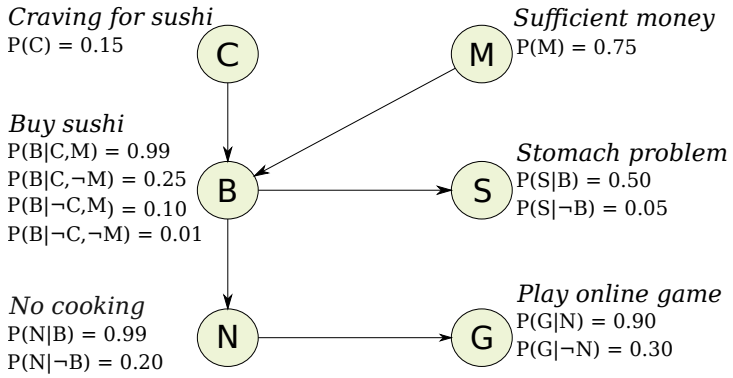


Figure 2: A Bayesian network of the sushi-for-dinner example.

parent node of X_j, and X_j a *child* node of X_i. Each node has its associated *parent set* in the graph, and a node's parent set can be empty. For a variable $X_i \in \mathbf{V}$, we denote its parent set as PA_{X_i}.

(ii) CPDs of all the variables – each variable $X_i \in \mathbf{V}$ is associated with a set of probability distributions $p(X_i|PA_{X_i})$, one for each of the possible configurations of PA_{X_i}. If X_i has no parent, then $p(X_i|PA_{X_i}) = p(X_i|\emptyset)$. In general, a DAG annotated with CPDs makes a Bayesian network. The DAG encodes the following factorized joint distribution over $\mathbf{V}$: $p(X_1, X_2, ..., X_n) = \prod_{i=1}^{n} p(X_i|PA_{X_i})$. Here we only need the probability values of each node conditioned on its parent set, because BNs have built-in independence assumption, which is referred to as *Markov property*[1]. The joint distributions represented in the above smoking-diseases example are: $p(F, C, B, L, H) = p(F|B, L)p(C|L)p(B|H)p(L|H)p(H)$.

The DAG structure of a Bayesian network can be discovered through *structure learning* based on the observed dataset, which is the focus of this thesis. The knowledge we gain from an observed dataset can reveal the dependence and independence of variables. The process of estimating the CPDs is called *parameter learning*, given a DAG structure and dataset. Parameter learning is out of the scope of the thesis.

After a Bayesian network model is built, we can use it to perform Bayesian inference – answering probabilistic queries. BNs have some advantages in completing the tasks of inference [FN12]. For example, whenever a new evidence becomes available, the probability distributions in a BN will be updated. The update can propagate not only from a parent node to a child node, but also vice versa due to the Bayes' theorem. In addition to backward reasoning, BNs can also discard old beliefs in the light of new evidence.

[1]Each variable in a BN is conditionally independent of its non-descendants given its parent set. Markov property is related to the structural property of BN *d-separation*. Both concepts can be found in text books or articles on Bayesian networks, for example [Pea88].

However Bayesian inference is not the focus of this article. Hence we will not discuss more about it here.

3 Structure Learning in Bayesian network

In many cases, applying Bayesian networks to a real-world problem requires discovering the network structure of the problem at hand by learning it from data. The search goal is to find an optimal network that is the best fit for the training data.

Two different network structures that represent precisely the same set of conditional independence relationships are considered equivalent (or, belong to the same equivalence class). Hence we say *an* optimal network, because there could exist more than one of such networks that are equivalent. Different networks from the equivalence class are not distinguishable based on the dataset [KF09]. Therefore the search objective of structure learning is to discover one of the equivalent networks.

Most of the learning algorithms fall into one of the following three categories. The first one is score-based learning, which measures the network structures by scores and searches for one that produces the best score. The second category is constraint-based learning, which employs conditional independence tests and analyzes the dependence/independence relationships between the variables. The goal is to find a network that explains the conditional independences as well as possible. The third category is hybrid methods, which combine conditional independence tests and score measuring. In this thesis we will focus on score-based learning, as it is a commonly-used technique, and the Para-OS algorithm belongs to this category as well.

3.1 Score-based Learning

Score-based learning methods use a scoring function to measure how well a candidate network structure fits to the given dataset. Each structure is associated with a real number score, computed by the chosen scoring function with respect to the dataset. In this framework, the problem of structure learning is treated as an optimization problem. We have the following input: a fully observed dataset $\mathbf{D}$, a chosen scoring function $Score(\cdot)$, and a search space formed by possible network structures. The expected output is a network structure that optimizes the score, which means it is the best fit for $\mathbf{D}$.

In learning optimal BN structures, the number of possible network structures in the solution space grows exponentially with the number of variables. We are faced with a combinatorial optimization problem. With this huge search space ($O(n2^{n(n-1)})$ possible DAGs [YM13]), score-based structure learning is proven to be a NP-hard problem for networks of indegree (number

of parents for any node in the DAG) as small as two [KF09], and NP-complete when BDe is used [Chi96].

In score-based learning, an algorithm for learning the optimal BN structure can be viewed as consists of two phases. The first is the scoring phase, which is responsible for finding the best parent sets by calculating the scores. The second is the searching phase, where the task is to search for the optimal network by using the results of the scoring phase. There are two key ingredients in score-based learning: scoring functions and search strategies [YM13]. Next we will review scoring functions first.

In the following discussion we define the network score of $\mathcal{G}$ as $DagScore(\mathcal{G})$. A *local score* is calculated solely for a variable X_i and its parent set PA_{X_i}, with respect to $\mathbf{D}$. It measures how well PA_{X_i} serves as parents of X_i in the given dataset. We denote a local score as $Score(X_i, PA_{X_i})$.

3.2 Scoring Functions

There are many scoring functions presented in the literature. Some well-known ones are Bayesian Dirichlet (BD) score and its variants BDe and BDeu [CH92, HGC95, Bun91], minimum description length (MDL) (or equivalently, Bayesian information criterion – BIC) [LB94, Sch78], Akaike Information Criterion (AIC) [Aka73], factorized normalized maximum likelihood function (fNML) [SRKM08], and mutual information tests (MIT) [DC06]. Depending on the chosen scoring function, the optimal score could be the minimum or the maximum score.

The scoring functions mentioned above have a common property – they are decomposable, which means they can be applied separately to each node and its parent set. Hence the score of a network $\mathcal{G}$ can be decomposed into an aggregate of local scores:

$$DagScore(\mathcal{G}) = \sum_{i=1}^{n} Score(X_i, PA_{X_i}) \tag{1}$$

Another property is *score-equivalence*. A scoring function is said to be *score-equivalent* if it assigns the same score for all DAGs that belong to the same equivalence class. With this property, the equivalent networks always yield the same score regardless which one of them is chosen. The aforementioned scoring functions are score-equivalent except BD, fNML and MIT. Score-equivalence is mentioned frequently in the literature. However, when searching for optimal BN structures, score-equivalence is not a crucial property for scoring functions [YC02, DC06].

The Maximum Likelihood (ML) scoring function can also be used to search for network structures that maximize the probability of observing the given dataset. At first glimpse, it looks natural to choose ML. However, ML encounters problems in structure learning. Because adding an edge to a network never decreases the likelihood (score), ML always prefers a more

complex network over a simpler one. It follows that the resulting networks are more complex than necessary – they are usually fully connected DAGs, which no more edges can be added without causing directed cycles. These complete networks suffer from overfitting. It means they fit the training data too precisely. Therefore, they poorly generalize the underlying probability distribution and perform badly on new data cases that are not in the training data. The need of further refinements of ML leads to the development of other scoring functions for BN structure learning.

In order to reduce the extent of overfitting, most of the aforementioned scoring functions penalize complex networks more than sparse ones. Scoring functions can be written as a log-likelihood (LL) function, which adopts a penalty term [SRKM08, LMY12]: $score(\mathbf{D}|\mathcal{G}) = LL(\mathbf{D}|\mathcal{G}) - \sum_{i=1}^{n} Penalty(X_i, \mathcal{G}, \mathbf{D})$. Here $LL(\mathbf{D}|\mathcal{G})$ represents the log probability of dataset $\mathbf{D}$ given network $\mathcal{G}$. The penalty term is used for favoring simpler models. Different scoring functions differ in the penalty term. Even though complex networks have a good LL score, the penalty term could reduce the overall score to be lower than the scores of simpler networks. It may seem that using the penalty term makes the score biased towards sparse structures. However when the dataset is big enough, the more complex structures also have chances to be chosen. In this way, the score provides a trade-off between the fit to data and the model complexity.

In practice, different scoring functions typically find different "optimal" networks, due to their different statistical inference foundations. However, once a score-equivalent scoring function is chosen, all the optimal learning algorithms discover equivalent networks [LMY12]. The choice of scoring function is an important and interesting problem under study, as there are so many of them available.

Next we focus on the introduction of BDeu, which is used in the implementation part of the thesis project. For further knowledge and analysis of other scoring functions, one could refer to papers such as [YC02, DC06, DCJ11, LMY12].

3.3 The BDeu Scoring Function

The Bayesian Dirichlet (BD) scoring function was first introduced by Cooper and Herskovits [CH92]. BD and its derivations have been popular scoring functions.

First we introduce some notations that are necessary for understanding BD in Table 2. It is necessary to gather *sufficient statistics* (or, *counts* in layman's terms) from dataset $\mathbf{D}$. We need two types of count, denoted as n_{ijk} and n_{ij} respectively, and n_{ij} is calculated from n_{ijk} (Table 2).

In addition to the assumption of complete data, BD also requires some other important assumptions, for example *parameter independence* and *Dirichlet priors* [HGC95]. Furthermore, the BD metric also requires a

r_i	the number of states of discrete variable X_i
x_{ik}	the k-th value of X_i ($1 <= k <= r_i$)
$q_i = \prod_{X_j \in PA_{X_i}} r_j$	the number of possible configurations of the parent set PA_{X_i}
y_{ij}	the j-th configuration of PA_{X_i} ($1 <= j <= q_i$)
n_{ijk}	the number of datapoints in $\mathbf{D}$ where X_i takes its k-th value x_{ik} and the variables in PA_{X_i} take their j-th configuration y_{ij}
$n_{ij} = \sum_{k=1}^{r_i} n_{ijk}$	the number of datapoints in $\mathbf{D}$ where the variables in PA_{X_i} take their j-th configuration y_{ij}
α_{ijk}	hyper-parameter for each variable-parents combination
$\alpha_{ij} = \sum_{k=1}^{r_i} \alpha_{ijk}$	hyper-parameter for each parent set configuration

Table 2: notation symbols for BD scoring function family

so-called *hyper-parameter* α_{ijk}, which is non-negative, to be specified for each variable-parents pair (i.e. for each of the possible i, j, k combinations). Another more general hyper-parameter α_{ij} is calculated from α_{ijk} (Table 2), similar to the way n_{ij} calculated from n_{ijk}.

The idea of BD is to evaluate the joint probability of the network and the given dataset: $p(\mathcal{G}, \mathbf{D})$. Here we omit the statistical process of deriving the equation. Equipped with the above notation, the BD score is defined as follows [HGC95]:

$$BD(\mathcal{G}) = p(\mathcal{G}) \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})}, \tag{2}$$

where $p(\mathcal{G})$ is the prior probability of the network $\mathcal{G}$, that is, a prior over network structures tells us which ones are more preferable. The gamma function Γ appears in the expression because of the assumption of Dirichlet priors.

Specifying hyper-parameters for all possible variable-parents combinations is an arduous and impractical job, and in addition BD is not score-equivalent. Therefore, further assumptions are adopted in order to address these issues and improve BD. One crucial assumption is *likelihood equivalence*, which means that the scores of equivalent networks are equal. This likelihood-equivalence specialization of BD is called BDe [HGC95]. A single hyper-parameter called *equivalent sample size* (ESS) is introduced, and all the required hyper-parameters α_{ijk} can be calculated from ESS. Hence, the construction of the score is largely simplified. BDe is score-equivalent, and it has the same equation as BD.

With the assumption that every network in the search space has equal probability to be the optimal one, $p(\mathcal{G})$ becomes uniform and disappears in the

computation. Furthermore, a particular case of BDe of using uninformative hyper-parameter, i.e. $\alpha_{ijk} = \frac{ESS}{r_i q_i}$, is called BDeu, where ESS is the only hyper-parameter that needs to be specified by the user. The logarithm is taken to simplify the computation, and the BDeu score [Bun91, CH92] is induced from BD as follows:

$$BDeu(\mathcal{G}) = \sum_{i=1}^{n} \sum_{j=1}^{q_i} (\log \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} + \sum_{k=1}^{r_i} \log \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})}) \qquad (3)$$

BDeu is also score-equivalent. BDeu is a score that penalizes complex networks, even the penalty term is not obvious from the above equation. For the interested reader, the penalty term can be found, for example, from Silander et al. [SRKM08]. Cooper and Herskovits [CH92] also proposed a particular case of the BD score, where an uninformative prior is used, i.e. $\alpha_{ijk} = 1$, called the *K2* score.

The network structures learned by using BDeu is sensitive to the setting of ESS [LMY12, SKM07]. The density of the network seems to be correlated with the ESS value. Typically, sparser networks are found with lower ESS value and denser ones with higher ESS value. Therefore, specifying an appropriate value for ESS requires information about the network density. BDeu is usually a good scoring function with an appropriate ESS value [SRKM08, LMY12]. In the case of lacking the necessary knowledge to estimate the network density, BDeu is probably not a suitable choice, or an uninformative value of 1 is used.

3.4 Search Strategies

As mentioned before, score-based BN structure learning is NP-hard [Chi96]. Even though the general problem is intractable, the size of the search space can be reduced by setting some restrictions. One can assume a predetermined ordering on variables and only search for network structures that are consistent with this ordering, or one can limit the indgree of the nodes in the network, etc. These restrictions are often combined with various algorithms. There is a wide literature on scored-based structure learning algorithms. Next we explore some of them based on their search strategies.

3.4.1 Approximate Methods

Due to the complexity of structure learning, exhaustive search is considered infeasible. Early research resorted to local search algorithms, which attempt to find good network but do not guarantee the optimality. Therefore, they are approximate methods. Local search strategies include greedy hill climbing, stochastic search, genetic algorithms, etc.

Greedy hill climbing [Glo90, TBA06] starts the search by selecting an initial network structure, which can be the empty network, a randomly

generated one, or a network constructed by using domain knowledge. Upon applying a single operation (insert/remove/reverse an edge $X_i \to X_j$), the score of the network is recomputed. Thanks to the decomposability of the scoring function, we only need to recompute the local scores of those variables whose parents have been altered. The associated changes in the network score are as follows: for insertion of "$X_i \to X_j$", we need to compute $Score(X_j, PA_{X_j} \cup \{X_i\})$; for deletion of "$X_i \to X_j$", we need to evaluate $Score(X_j, PA_{X_j} \backslash \{X_i\})$; and for reversal of "$X_i \to X_j$", two local scores $Score(X_j, PA_{X_j} \backslash \{X_i\})$ and $Score(X_i, PA_{X_i} \cup \{X_j\})$ are to be updated. The decomposability saves much computation during the search. By evaluating the improvement of different operations, the one that leads to the best score is then selected. This process is repeated until no modification can improve the score. However, the returned network typically just reaches a *local maximum*. There are several strategies to improve the result of greedy hill climbing. For example, *tabu search* with random starts was proposed by Glover [Glo90]. With randomization, we have better chances to escape from the local maxima. Tabu search forbids applying an edge operation that reverses the effect of recent changes. In addition, the search does not stop while the score cannot be improved anymore. Instead it continues some pre-specified steps in the hope of discovering a better structure.

Other approximate methods include stochastic search such as Markov chain Monte Carlo [MAPV96, MLL99], simulated annealing [Hec98], variable neighborhood search [DCP01]; genetic algorithms [LKMY96, HGPS02], ant colony optimization [DCFLGP02], etc.

Approximate methods can handle larger networks than exact methods (see next section), which is their advantage. However, due to the problem of easily getting stuck in local maxima, approximate methods have a major drawback – the quality of the network found is unknown.

3.4.2 Optimal Methods and Dynamic Programming

Several exact learning algorithms have emerged in learning optimal Bayesian networks based on dynamic programming [OIM04, KS04, SM05, SM06]. These algorithms guarantee the optimality of returned networks. An optimal BN can be found in $O(n2^n)$ time and space, i.e., still exponential in the number of variables. The number of variables that these algorithms can handle is still limited to several dozens. Other optimal methods include for example A* [YM13], integer linear programming (ILP) [JSGM10, Cus11] and branch-and-bound [DCJ11]. Next, we explain how dynamic programming tackles the structure learning problem to discover a globally optimal network structure.

3.4.2.1 Dynamic Programming

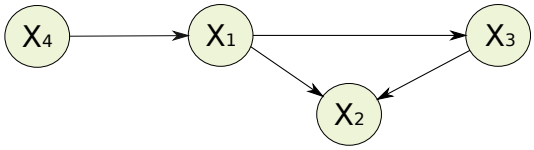


Figure 3: A simple DAG which is compatible with the ordering: X_4, X_1, X_3, X_2.

The idea of dynamic programming is based on the following observation. Define a *sink* in a DAG to be a variable that has no outgoing edges, i.e. no children. The acyclicity property ensures that there is at least one sink in the DAG, as every directed path in a DAG must have finite length and ends at a sink. Hence a DAG can be represented as an *ordering* of the variable set $\mathbf{V}$, with the constraint that each variable precedes its child variables. For example, a DAG of four variables in Figure 3 is compatible with the ordering: X_4, X_1, X_3, X_2, where X_2 is a sink [SM06]. Once an optimal ordering of $\mathbf{V}$ is discovered, the corresponding optimal Bayesian network can be constructed by letting each variable choose the best parent set from its predecessors. As the "last" variable in an ordering of $\mathbf{V}$ must be a sink, finding an optimal ordering of $\mathbf{V}$ is equivalent to locating the best sink incrementally, as explained in more detail later.

The dynamic programming method can be divided into two phases – the scoring phase and the searching phase [YM13]. In the scoring phase, each variable finds its optimal parent set. A variable X_i's parent set PA_{X_i} can be chosen from the parent candidate set $\mathbf{V}\backslash\{X_i\}$. The subset of $\mathbf{V}\backslash\{X_i\}$ that yields the best score is the optimal parent set for X_i, as defined in Equation 4 below[2].

$$BestScore(X_i, \mathbf{V}\backslash\{X_i\}) = \min_{PA_{X_i} \subseteq \mathbf{V}\backslash\{X_i\}} Score(X_i, PA_{X_i}). \tag{4}$$

Assuming a variable set $\mathbf{U} \subseteq \mathbf{V}\backslash\{X_i\}$, Equation 4 can be written as a recursive function, as shown below in Equation 5. This recursive function allows us to proceed incrementally. We can start from an initial case where $\mathbf{U}$ is the empty set, and increase the cardinality of $\mathbf{U}$ by one at each iteration, until the cardinality reaches n, where $n = |\mathbf{V}|$.

$$BestScore(X_i, \mathbf{U}) = \min\{Score(X_i, \mathbf{U}), \min_{Y_i \in \mathbf{U}} BestScore(X_i, \mathbf{U}\backslash\{Y_i\})\} \tag{5}$$

In the searching phase, the optimal ordering of $\mathbf{V}$ can also be found in an incremental way, by locating the sink at each iteration. An ordering of a

[2]The best score is either the maximized score (max) or the minimized score (min), depending on the chosen scoring function. For example MDL uses min and BD uses max. In compliance with [TIM11], here we use the minimum score to indicate the best score.

variable set corresponds to a network whose quality is measured by a score. Suppose the best sink in the variable set $\mathbf{U}$ is X_s, and an optimal ordering (i.e. the subnetwork) of $\mathbf{U}\backslash\{X_s\}$ is known. Then an optimal network for $\mathbf{U}$ can be obtained by: (i) add X_s to the end of the ordering; (ii) let X_s select its optimal parent set from $\mathbf{U}\backslash\{X_s\}$, which is done by Equation 5. Therefore, in order to produce the optimal network of $\mathbf{U}$, the best sink X_s must be the variable that maximizes the sum of the subnetwork score of $\mathbf{U}\backslash\{X_s\}$ and the local score for finding the best parent set of X_s from $\mathbf{U}\backslash\{X_s\}$. The recursion is defined in recursive Equation 6. For $\mathbf{U} \subseteq \mathbf{V}$, with $DagScore(\{\}) = 0$, we have:

$$DagScore(\mathbf{U}) = \min_{X_s \in \mathbf{U}} \{DagScore(\mathbf{U}\backslash\{X_s\}) + BestScore(X_s, \mathbf{U}\backslash\{X_s\})\}. \tag{6}$$

The recursion of Equation 6 proceeds in a similar fashion as Equation 5 – starting from a single variable set (a trivial case), at each iteration we increase the cardinality of $\mathbf{U}$ by one. Repeat this process until the cardinality reaches n, at which point we have the sink of $\mathbf{V}$ and hence the optimal Bayesian network structure.

The two recursions, in which Equation 6 calls Equation 5, can be combined to tackle the problem of learning optimal Bayesian network structure in a *levelwise* fashion. In order to explain this levelwise idea of dynamic programming, we introduce the order graph [YM13], which is a Hasse diagram that visualizes the state space. An example of a four-variable diagram is illustrated in Figure 4.

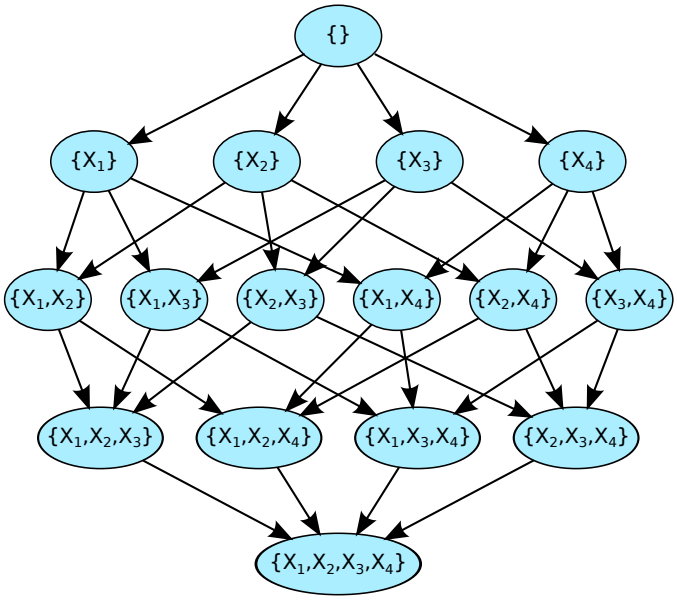


Figure 4: An order graph of four variables [YM13]

The *start* node in the order graph of Figure 4 is the one with the empty

set, and the *goal* node is the one with the full set of all four variables. We can see that the nodes of the same level represent the variable sets of the same size. Coming from top to bottom, at each level the size of the variable sets is increased by one, corresponding with the recursive process described above. An edge pointing from a predecessor node $\mathbf{U}$ to a successor node $\mathbf{U} \cup \{X_i\}$ represents the score of selecting an optimal parent set for X_i by using $\mathbf{U}$ as the candidate set, i.e. $BestScore(X_i, \mathbf{U})$. Because a new node is added along with each edge, a path from the start node to the goal node yields an ordering of all the variables. For example, the following path $\{\} \to \{X_4\} \to \{X_1, X_4\} \to \{X_1, X_3, X_4\} \to \{X_1, X_2, X_3, X_4\}$ indicates the ordering: X_4, X_1, X_3, X_2. The score of a path is straightforward – the sum of the scores of all the edges along the path. We also calculate the optimal parent set for each sink along the path.

To see how the levelwise recursion works, let us consider node $\{X_1, X_3, X_4\}$ on the fourth layer of the order graph in Figure 4. There are three ways to construct the network for this node: by using (i) subnetwork $\{X_1, X_3\}$ and sink X_4; (ii) subnetwork $\{X_1, X_4\}$ and sink X_3; (iii) subnetwork $\{X_3, X_4\}$ and sink X_1. The optimal ordering for each possible subnetwork is already found and saved in the previous level. Now we need to find out which variable is the sink, and select its optimal parents from the associated subnetwork. Then the optimal network for $\{X_1, X_3, X_4\}$ is discovered [YM13]. This process is done from top to bottom in the order graph, level by level, until we reach the goal and obtain the optimal Bayesian network for $\mathbf{V}$.

Many research use dynamic programming over reduced search spaces, for example some kind of structural constraints [KPIM10, OM03, PIM08]. On a different front, Yuan and Malone [YM13] solve the problem from a shortest path perspective. Next we take a look at their method.

3.4.2.2 A* and other methods

In the order graph, it is intuitive that the path that yields that minimum score can be seen as the shortest path. Indeed Yuan and Malone [YM13] innovatively maps the problem of learning optimal BN structure to the problem of finding a shortest path, which is then tackled by an A* algorithm. The search space for A* is the order graph. A* utilizes an open list to store the search frontier, and a closed list to store the expanded nodes. During the search a node $\mathbf{U}$ in the order graph is evaluated by a cost function $f(\mathbf{U})$, which is the sum of $g(\mathbf{U})$ – the cost from the start node to $\mathbf{U}$, and $h(\mathbf{U})$ – the estimated cost from $\mathbf{U}$ to the goal node. The cost $h(\mathbf{U})$ is calculated by a heuristic function, for which Yuan and Malone proposes *k-cycle conflict heuristic*. The rest of the variables $\mathbf{V} \backslash \mathbf{U}$ is divided into several exclusive acyclic group, but cycles between the groups are allowed, which is a relaxation of the original DAG acyclic constraint. Here k refers to

the size of each group can be up to k. The heuristic cost of $\mathbf{U}$ is the sum of the heuristic values of these acyclic groups. the heuristic value of a group is the cost of the shortest path from node $\mathbf{V}\backslash\mathbf{U}$ to the goal node in the order graph. The k-cycle conflict heuristic has a dynamic version and a static version, and both are admissible and consistent. Using heuristic function as guidance, A* only explores the most promising nodes in the order graph. In this way only a part of the order graph is built in finding the optimal solution, comparing to the whole order graph in the dynamic programming approach. Hence, their algorithm may bring significant saving in time and memory. If the heuristic values can be computed efficiently, this A* approach outperforms the dynamic programming approach. Overall, their algorithm shows promising empirical results.

Another approach formulates the exact BN structure learning problem as integer linear programming (ILP), with a limit on parent set sizes (bounded in-degree) [JSGM10, Cus11]. The BN structure learning problem is formulated as a linear programming (LP) problem. Let a binary variable $I(PA_{X_i} \rightarrow X_i)$ $= 1$ iff PA_{X_i} are the parent set for X_i in a DAG. The LP objective function from Cussens [Cus11] can be defined as follows:

Instantiate the $I(PA_{X_i} \rightarrow X_i)$ to

maximize: $$\sum_{X_i, PA_{X_i}} Score(X_i, PA_{X_i}) * I(PA_{X_i} \rightarrow X_i)$$

subject to $I(PA_{X_i} \rightarrow X_i) = 1($ for each $X_i \in \mathbf{V})$ representing a DAG.

The LP objective function from Jaakkola et al. [JSGM10] expresses similar idea. The key difficulty is ruling out cycles in the graphs, and for this purpose some constraints are added iteratively. The method used for searching is called *branch and cut*, which solves an LP relaxation. During solving they add linear constraints (in the form of *cutting plane*) that drive towards an integer solution. If the result is integral, then it is guaranteed that the found BN structure is optimal. Otherwise a branch and bound method is used for searching to ensure the optimality in Jaakkola et al. [JSGM10]. Cussens [Cus11] focused on finding good cutting planes for an optimal solution.

De Campos et al. [DCZJ09, DCJ11] proposed a branch-and-bound algorithm. They start by constructing a network which is a relaxation of DAG, i.e. may contain cycles. Then the cycles are broken by enforcing some edge in a cycle to be absent or present. They use the inclusion-exclusion principle of combinatorics to ensure that there is no overlapping among subcases, and all the subgraphs in the search space are covered. During the search, a branch consists a group of subgraphs. If the greatest score of this branch is not better than the best known score, then the whole branch is discarded. Otherwise, the branch enters the cycle-checking phase, and cycle-removing phase if necessary. The algorithm is anytime – it can stop at any point and provide the best solution so far. If run to the end, a globally

optimal BN is guaranteed.

3.4.3 Parallelized Methods

The exact algorithms face the challenge of scaling up the size of the solvable problems. As large computing multi-core architectures become widespread, parallelization was emerged as a natural choice. Many algorithms presented in the literature are parallelizable. Here we introduce several of them. Tamada et al. [TIM11] proposed a parallelized version (called *Para-OS*) of the dynamic programming algorithm from Ott and Miyano [OIM04]. This parallelized algorithm is also the one implemented and tested in this thesis project. Details of Para-OS will be introduced in next chapter.

Malone and Yuan [MY12] presented a *bounded error, anytime parallel* (BEAP) algorithm, which is a parallelization of the exact A* algorithm. BEAP has anytime behavior, which means the algorithm can quickly produce a solution when it is interrupted at any time prior to completion, and the quality of the solution improves when the algorithm keeps searching for longer time. As described above in Section 3.4.2.2, A* evaluates the cost f of a node $\mathbf{U}$ in order graph by $f(\mathbf{U}) = g(\mathbf{U}) + h(\mathbf{U})$. BEAP applies Weighted A* (WA*), which weights the heuristic function $h(\mathbf{U})$ by a number ϵ ($\epsilon \geq 1$, for example $\epsilon = 1.05$). It is guaranteed that the globally optimal solution has a cost of at most ϵ times the cost of the found solution, hence the name "bounded error." A range of ϵ values are selected, among them the smallest ϵ equals 1, which represents the unweighted exact A* algorithm. Each of the ϵ values is assigned to a process for running WA*. The idea of BEAP is to run WA* parameterized by different weight in parallel. Each process does not need to communicate with others during the calculation. Processes with larger ϵ values finish faster than the ones with lower ϵ values. Each completed process gives an upper bound on the optimal solution. Given enough time for the process parameterized by weight 1 to finish, BEAP guarantees to find an optimal BN structure.

When using dynamic programming for structure learning, the main computational bottleneck is the memory. Parviainen and Koivisto [PK13] recently proposed a parallelized algorithm, which can trade longer running time for less memory consumption and can be easily parallelized. The idea is to break the search problem into several independent subproblems. As mentioned before, a DAG has a topological ordering on the variable set $\mathbf{V}$. Each variable has its parent variables among the predecessors in this ordering, except the "first" variable. Their idea is to specify a *partially ordered set* (P) on $\mathbf{V}$. A partially ordered set is compatible with some DAGs in the search space. Therefore a collection of partially ordered sets $\mathcal{P}$ can be used to divide the search space, that is, $\mathcal{P}$ covers all the possible DAG structures in the search space. Now each $P \in \mathcal{P}$ represents a subproblem, which can be solved independently on a process without communication with

other subproblems. From a partially ordered set P one can derive a set of ideals. An *ideal* I refers to a subset of P that is "closed under taking smaller elements" (for two elements $x, y \in P$ and $x < y$, if $y \in I$, then $x \in I$). The algorithm trims the dynamic programming method to run only across the set of ideals. The search goal of each subproblem is to find the best DAG that is compatible with P that represents that subproblem. The result of a subporblem is a DAG with score. Solving all of the subproblems will lead to the final optimal BN, which is simply the DAG that yields the best score. This setting provides an efficient parallelization scheme. Each P is assigned to a processor, hence we just need $|\mathcal{P}|$ number of processors. Parviainen and Koivisto have proven that the time and space complexity of dynamic programming grows roughly linearly in the number of the ideals. The number of ideals could be much smaller than the number of all the subsets of $\mathbf{V}$, which the dynamic programming method needs to process. Hence, their algorithm may bring great improvement. The tradeoff between time and memory can be controlled by what kind of partially ordered sets are chosen to cover the search space. They also suggested a partial order system called *parallel bucket orders*, which provides good tradeoff in practice.

Both BEAP and the Parviainen-Koivisto algorithm implement independent calculation in processors. There is no communication needed during the calculation, which is in stark contrast to Para-OS. A big percentage of the running time of Para-OS is used in passing messages around, waiting for synchronization, etc. We will discuss more about it later.

4 The Parallelized Algorithm Para-OS

The Para-OS algorithm is introduced in Tamada et al. [TIM11]. The original non-parallelized version of the algorithm can be found from Ott and Miyano [OIM04], which is the same dynamic programming algorithm that has been described in Section 3.4.2.1, for discovering the global optimal structure of Bayesian network with complete search. As the name suggests, Para-OS parallelizes the dynamic programming to let it run distributively over many processors, utilizing the memory of individual nodes in the computer cluster. The algorithm neither limits the indegree of the graph, nor assumes any predetermined variable ordering. It is successfully used to search the optimal structure of a 32-node network [TIM11].

In learning an optimal Bayesian network structure, the search space grows exponentially with the size of the problem. When applying exact learning methods we need to save the results from all of the intermediate calculations, which leads to the main bottleneck – memory. One solution is to use supercomputers that are equipped with several terabytes memory, but they are still expensive and not easily accessible. On the other hand, massively parallel computer clusters are inexpensive and more available. Therefore, Para-

OS employs parallelism to distribute the calculation over many processors of a computer cluster, and save the intermediate results in memory of individual nodes. Para-OS also tries to minimize the communications between independent parallel processors. However this introduces some redundant calculations.

4.1 Split the Calculations

The key idea in Para-OS is to split and parallelize the calculations. The first phase of Para-OS is to compute the raw local scores, which is a heavy part of the calculation. The scoring function $Score(X_i, PA_{X_i})$ is applied, in an exhaustive fashion, to all of the variables and their possible parent sets. As mentioned before, for n variables we have to compute $n2^{n-1}$ raw local scores. Communication between processors is not needed in this stage, as the computation of the local score of each variable, given its parents, is independent of other variables. Thus the splitting of the local score calculations is straight forward – they are equally divided onto processors, and computed independently. The results are stored distributively, and can be retrieved later when necessary.

The main technical question in Para-OS is how to split the calculation of dynamic programming, which is solved in the second phase of Para-OS. As described in Section 3.4.2.1, the dynamic programming algorithm proceeds in a levelwise fashion. There are a group of variable sets of the same cardinality to be processed at each level. For example in the order graph of Figure 4, there are six variable sets of cardinality $k = 2$ to be calculated at the third level: $\{X_1, X_2\}, \{X_1, X_3\}, \{X_2, X_3\}, \{X_1, X_4\}, \{X_2, X_4\}$, and $\{X_3, X_4\}$. Let us refer them as *Current* sets, as they are the current tasks at hand for a specific layer of the order graph. The task is to split these six Current sets into groups, in order to be distributed. Para-OS uses the concepts of *Superset* to complete this split task[3]. The separation of the above six Current sets is achieved as follows, by using three Supersets of cardinality $k = 3$ (see Figure 5): (i) Superset $\{X_1, X_2, X_3\}$ is assigned to the processor of index 1, where it is used to generate three Current sets $\{X_1, X_2\}, \{X_1, X_3\}$, and $\{X_2, X_3\}$; (ii) from Superset $\{X_1, X_2, X_4\}$ we can derive three Current sets $\{X_1, X_2\}, \{X_1, X_4\}$, and $\{X_2, X_4\}$ in processor 2; (iii) Superset $\{X_1, X_3, X_4\}$ is for generating Current sets $\{X_1, X_3\}, \{X_1, X_4\}$, and $\{X_3, X_4\}$ in processor 3. One question might be raised here is why the set $\{X_2, X_3, X_4\}$ is not in consideration. It is because all six Current sets that need to be calculated are covered in these split portions from (i) to (iii). Adding the set $\{X_2, X_3, X_4\}$ will just introduce redundant calculation. There are already some redundancy generated from (i) to (iii), for example $\{X_1, X_2\}$ is calculated in both processor 1 and processor 2. In fact there

[3]For some variable sets $X \subset Y$, we say that Y is the Superset of X, and X is the Subset of Y.

are nine Current sets generated, and among them three sets are repetitive – $\{X_1, X_2\}, \{X_1, X_3\}$, and $\{X_1, X_4\}$). Due to the repetition they will be calculated twice in different processors.

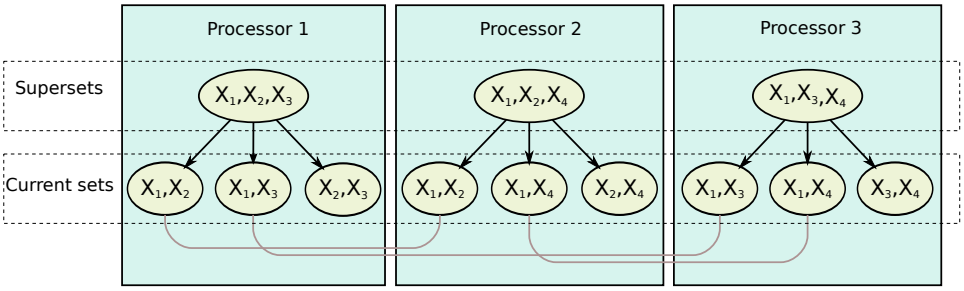


Figure 5: Illustration of the calculation split of 6 variable sets of cardinality $k = 2$ over 3 processors, with redundant calculation of 3 variable sets. (The repetitive variable sets are connected by red lines.)

In the above example, the key insight is that a group of Current sets can be derived from a Superset. Loosely speaking, we can say that a Superset forms "a group of" Current sets. In Para-OS a Superset has cardinality $k+\sigma$, where σ is used for controlling the split portion size – therefore controlling the trade-off between communication and memory. For calculating a specific level of the order graph, the idea of distribution goes as follows: first, we need some Supersets that will be used to generate all Current sets of this level (with some redundancy); second, each Superset is assigned to a processor, where it is used to derive the group of Current sets of size k, i.e. a collection $\mathcal{C} = \{Current \subset Superset : |Current| = k\}$[4]; at last, all the Current sets in $\mathcal{C}$ are calculated in that processor (discussed in the next section). The minimum number of such Supersets we need is combination $\binom{n-\sigma}{k}$, in order to ensure that all of the generated subsets together cover the necessary Current sets [TIM11]. In addition, the split portion size is $|\mathcal{C}| = \binom{k+\sigma}{k}$. In other words, the search space of level k can be split into $\binom{n-\sigma}{k}$ portions of calculation, with the size of each portion is $\binom{k+\sigma}{k}$. In the previous example, we have $\sigma = 1$, as the Supersets are one-size larger than the Current sets. We can easily verify that $\binom{n-\sigma}{k} = \binom{4-1}{2} = 3$, and $\binom{k+\sigma}{k} = \binom{2+1}{2} = 3$. Indeed the six Current sets from the third level of the order graph are separated into three portions, with the size of each portion is three. Tamada et al.[TIM11] also prove that the amount of redundant calculation is at most $O(n^\sigma)$. The time and space complexity of the algorithm is $O(n^{\sigma+1}2^n)$.

To summarize, the task is to split the Current sets that should be calculated at the layer k of the order graph, and the partition is achieved by using Supersets. Executing the algorithm by looping over k from 1 to n (the case of $k = 1$ is trivial), we traverse through all of the layers in the

[4]Notation $|\cdot|$ refers to the cardinality of a set or vector.

order graph, from top to bottom. Hence, this strategy successfully divides the calculations of dynamic programming into different processors. The reason Para-OS partitions the work in this way is due to the consideration of reducing communications between individual processors, which is discussed in the next section. Generating Supersets is discussed in Section 4.3.

4.2 Calculation in a Single Processor

In previous section we discussed the calculation of the raw local scores, and splitting the Current sets in a layer k of the order graph. The split portions are used in individual processors for the calculation of Equations 5 and 6 – the former is to search for the optimal parent set from the candidate set, the latter is to search for the optimal subnetwork of the candidate set. Communication between processors is necessary at this stage. Some former results are needed, and they are retrieved from other processors. We can see from Equation 6 that the computation of a Current set requires collecting subnetworks ($DagScore(\cdot)$) of one-size smaller subsets, and $BestScore(\cdot)$ of the sink. The calculation of Equation 5 asks for some results of $Score(\cdot)$ and $BestScore(\cdot)$ from previous layers of the order graph.

Let us first look at the retrieval in Equation 6 ($DagScore(\cdot)$), which requires one-size smaller subnetworks. If the retrieved subnetworks can be re-used by more than one Current set in the same processor, then we reduce the cost of information exchange between processors. Subnetworks of Current are retrieved by using one-size smaller *Subsets* of Current. The relationships between Supersets, Current sets and Subsets are illustrated in Figure 6, which is an extension of Figure 5.

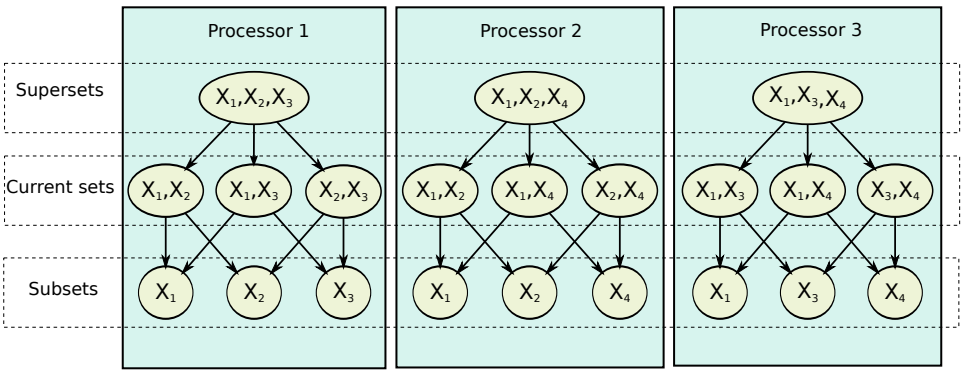


Figure 6: Para-OS utilizes Supersets and Subsets to parallelize the calculation of Current sets. Supersets are used for calculation split, and Subsets for data retrieval.

Continuing the previous example, let us look at processor 3, which uses Superset $\{X_1, X_3, X_4\}$ as input and is responsible for the calculations of Current sets $\{X_1, X_3\}, \{X_1, X_4\}$, and $\{X_3, X_4\}$. The retrieving and computation

process is illustrated in Figure 7. The collections of one-size smaller Subsets for Current sets are $\{\{X_1\},\{X_3\}\}$, $\{\{X_1\},\{X_4\}\}$, and $\{\{X_3\},\{X_4\}\}$, respectively. Because they are in overlap, we only need to retrieve the subnetworks of three sets $\{X_1\},\{X_3\}$ and $\{X_4\}$. As it is shown in Figure 7 the values of $DagScore(\{X_1\})$, $DagScore(\{X_3\})$, and $DagScore(\{X_4\})$ are retrieved from other processors. This overlap of the retrieved subnetworks limits the communication costs of sending intermediate results. It is proven in Tamada et al. [TIM11] that for each split portion, the number of needed subsets is $\binom{k+\sigma}{k-1}$. In our simple example, we can verify that $\binom{k+\sigma}{k-1} = \binom{2+1}{2-1} = 3$, and we indeed need to fetch 3 subnetworks.

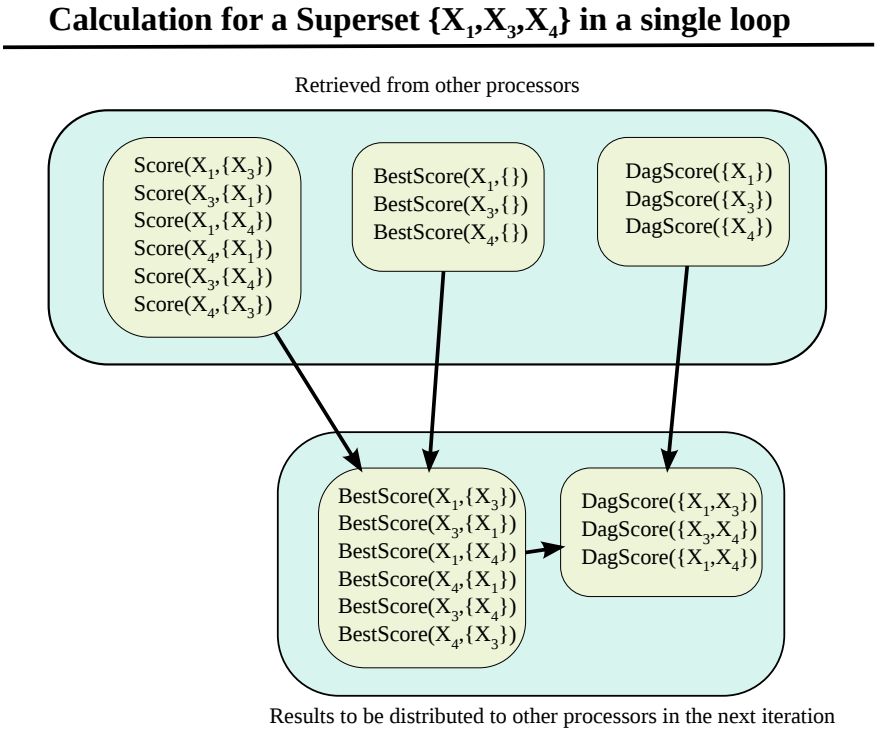


Figure 7: Illustration of the calculation for Superset $\{X_1, X_3, X_4\}$ in Processor 3 for $k = 2$ (adapted from [TIM11]).

The information required by Equation 5 are previous results from $Score(\cdot)$ and $BestScore(\cdot)$. For example, for variable set $\{X_1, X_3\}$, which is to be calculated in processor 3, we should retrieve $Score(X_1, \{X_3\})$ and $BestScore(X_1, \{\})$ in order to obtain $BestScore(X_1, \{X_3\})$. Similar values are retrieved for other variable sets in processor 3 (Figure 7). Here again some of the retrieved data can be shared for calculation of the variable sets in the same processor. For example $BestScore(X_1, \{\})$ is needed for the calculation of $BestScore(X_1, \{X_3\})$ and $BestScore(X_1, \{X_4\})$. At the end of this single loop in processor 3, we obtain the optimal subnetworks of $\{X_1, X_3\}, \{X_1, X_4\}$,

and $\{X_3, X_4\}$. The results of $DagScore(\cdot)$ and $BestScore(\cdot)$ of this loop are also stored distributively, and can be retrieved later when necessary.

4.3 The Mapping Function

During the whole parallelized calculation, the intermediate results are stored in a distributed fashion over the processors. Therefore we need some mapping functions to indicate on which processor particular results reside. There are three types of information that need to be saved and retrieved: the local scores – $Score(X_i, PA_{X_i})$, the optimal parent set of X_i from a candidate set $\mathbf{U}$ – $BestScore(X_i, \mathbf{U})$, and the optimal subnetwork of a variable set $\mathbf{U}$ – $DagScore(\mathbf{U})$. All of them involve a variable set. Thus we need to map between a variable set and a processor index. Para-OS has two mapping functions to obtain the processor index for storing and retrieving the intermediate results, and both exploit *reverse lexicographical index* (RLI) function.

In Para-OS, the variables and processor indices are both represented as natural numbers. The RLI function maps from a vector of natural numbers (which is a variable set) to a natural number index. Table 3 shows the results of $\text{RLI}(\mathbf{X}, n)$, where $\mathbf{X}$ is a vector of natural numbers. We can see that the indices produced are in reverse lexicographic order, hence the name of the function. For some vector $\mathbf{X} = \{x_1, x_2, ..., x_k\}$ with $|\mathbf{X}| = k$, $\text{RLI}(\mathbf{X}, n)$ can be calculated as follows.

$$\text{RLI}(\mathbf{X}, n) = \sum_{i=1}^{k} \binom{n - x_i}{k - i + 1} + 1 \tag{7}$$

The computation of RLI includes some "a choose b" combinations. A n-by-n square matrix called *the index table* is used to tabulate the results of such combinations. The idea is that after the table is computed, the calculation of RLI is mainly about looking up values in it. For example, by applying Equation 7 we have $\text{RLI}(\{1, 3, 5\}, 6) = \binom{5}{3} + \binom{3}{2} + \binom{1}{1} + 1 = 15$. The results of $\binom{5}{3}$, $\binom{3}{2}$, and $\binom{1}{1}$ are retrieved from the index table, instead of being calculated on the fly. The index table makes the RLI calculation linear, as we only need to check the table k times and return the sum. The index table is initialized by setting all values of the first column to zero, and values of the first row to be natural numbers $\{0, 1, 2, 3, ...\}$. After that, the value of each cell in the rest of the table is computed as the sum of the cell above it and the cell to its left (See Table 4 for an example of 5-by-5 index table). Let $T(x, y)$ indicate the position in the index table with row x and column y. Once the index table is ready, the result of $\binom{n-x_i}{k-i+1}$ can be obtained from the table at $T(k - i + 1, n - k - x_i + i + 1)$.

The Supersets, which are used for splitting calculations in Para-OS, are generated by RLI^{-1} – the inverse function of RLI. Function RLI^{-1} maps

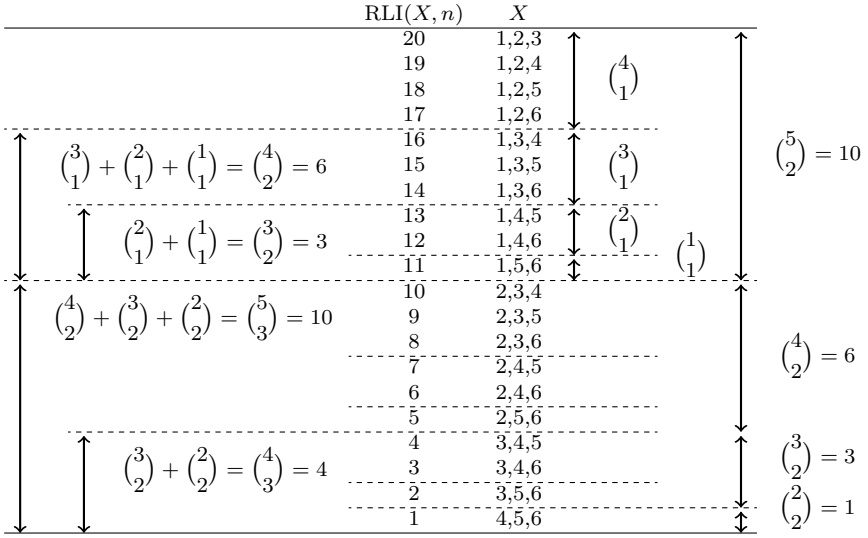


Table 3: RLI calculation for $n = 6$ and $k = 3$ [TIM11].

0	1	2	3	4
0	1	3	6	10
0	1	4	10	20
0	1	5	15	35
0	1	6	21	56

Table 4: Example of 5-by-5 index table.

from an index to a vector of natural numbers. In addition, RLI^{-1} is used for producing the possible parent sets in the local scores collection stage. If the output of RLI is used as input for RLI^{-1}, the result will be the original vector $\mathbf{X}$, i.e. $\mathbf{X} = \text{RLI}^{-1}(\text{RLI}(\mathbf{X}, n), n, k)$. With $r = \text{RLI}(\mathbf{X}, n)$, the inverse function recovers an original element X_i in $\mathbf{X}$ as follows.

$$X_i = n - k - j + i + 1, \text{where}$$
$$j = \arg\max_j T(k{-}i{+}1, n{-}k{-}j{+}i{+}1) < (r{-}\sum_{\alpha=1}^{i-1} T(k{-}\alpha{+}1, n{-}k{-}x_\alpha{+}\alpha{+}1)). \tag{8}$$

Repetition of the above equation k times with $i = 1, ..., k$ will restore $\mathbf{X}$. The computation of RLI^{-1} also utilizes the index table. X_i can be obtained once the j value is found, as the other values n, k, i in the equation are already known. Locating the j value mainly involves looking up values in the index table. Therefore, RLI^{-1} also runs in linear time, once the index table is ready.

To summarize, RLI and its inverse function RLI^{-1} ensure a bijection. These two functions are used in the algorithm because the conversion between the vector and the index is linear. Furthermore, it is sufficient to compute the index table only once at the beginning of the program.

From the way RLI table is created we can see that the numbers in RLI table grows quite fast as the number of variables n grows (for example, for $n = 30$ the largest number in the table is $29,065,024,282,889,672$). Using UINT64 to represent numbers, one number in RLI table already overflow when $n = 35$. Therefore, it limits the usability of RLI mapping to $n \leq 34$.

4.4 New Para-OS

Para-OS has some limitations. The first issue is the intense data communication. As described above, Para-OS does data retrieval and storage of $Score(\cdot)$, $BestScore(\cdot)$, and $DagScore(\cdot)$ (together with the associated parent sets and subnetworks), which means it uses quite much time in data communication. This is verified by the implementation of Para-OS. Hence, it would be preferable to reduce the communication during parallelization. The second issue is that Para-OS does not perform score pruning, which could reduce memory consumption and computation. The scoring phase is intervened with the searching phase in a levelwise fashion in Para-OS. Score pruning is not easily applicable without retrieving previous results. It is unwise to introduce more data communication. Therefore it makes sense to separate the scoring phase totally from the searching phase, as it is done in many other algorithms in the field. We propose and implemented a new version of Para-OS to address these issues.

4.4.1 Sparse Parent Graph

The dynamic programming algorithm requires calculation of a large amount of $Score(\cdot)$ for getting the optimal parent sets. For n variables, there are $n2^{n-1}$ possible pairs of variable-and-parent-set combination, therefore same amount of raw local scores. This task grows exponentially in n, and requires a lot of memory. That is why score pruning can play an important role.

In order to see how to prune, let us start from the situation before pruning. Raw local scores $Score(\cdot)$ calculated from a variable and all of its possible parent sets can form a Hasse diagram, called the *parent graph*. For instance, results of $Score(X_1, PA_{X_1})$ for all the possible PA_{X_1} form the parent graph for X_1, as illustrated in Figure 8 (a). Each node in the parent graph contains a parent set and the score. The results of $Score(\cdot)$ are then propagated to calculate $BestScore(\cdot)$ (Equation 4) in Figure 8 (b). The propagation works as follows. Assuming $\mathbf{U}$ to be the candidate parent set, for computing $BestScore(X_1, \mathbf{U})$ we need to examine all the possible subsets of $\mathbf{U}$, and select the one that yields the best score. For example, the empty set node

in (a) has score 10, which means $Score(X_1, \{\}) = 10$; the node $\{X_4\}$ in (a) tells us $Score(X_1, \{X_4\}) = 13$, and this node's score is updated in (b) to be 10, because the empty set is a subset of $\{X_4\}$ and it produces a better score; node $\{X_2, X_4\}$ in (a) has score 9, and it is updated to be 8 in (b), because a subset $\{X_2\}$ yields a better score: $Score(X_1, \{X_2\}) = 8$; at the same time $\{X_2\}$ is recorded to be the optimal parent set for X_1 chosen from candidate set $\{X_2, X_4\}$, and so forth.

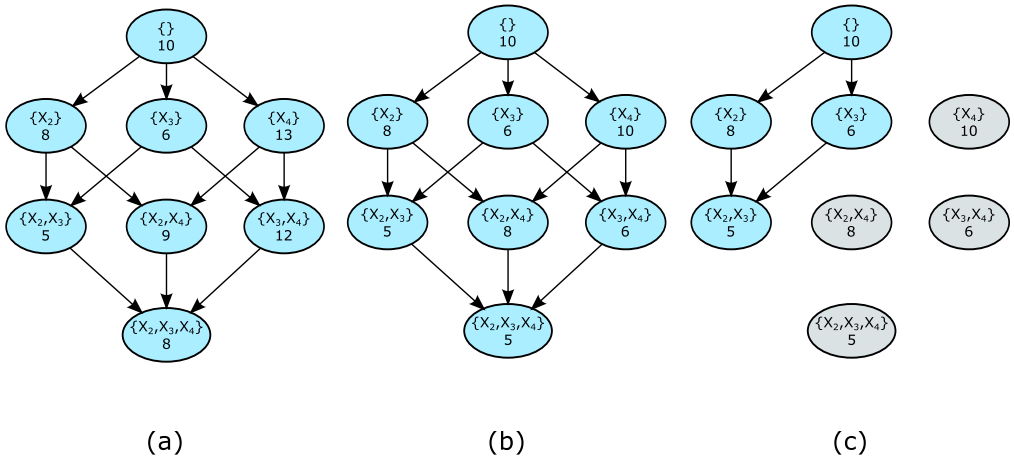


Figure 8: Raw local scores $Score(\cdot)$ of X_1 and its possible parent sets form the parent graph in (a), which is propagated to $BestScore(\cdot)$ in (b). After score pruning we get the sparse parent graph in (c) ([YM13]).

Score pruning can be done based on the following observation. Assume PA_{X_i} and PA'_{X_i} are both parent sets for X_i with the constraint that $PA_{X_i} \subset PA'_{X_i}$. If $Score(X_i, PA_{X_i})$ is better than $Score(X_i, PA'_{X_i})$, then PA'_{X_i} cannot be the parent set for X_i in an optimal BN. Because replacing PA'_{X_i} by PA_{X_i} as parents for X_i in a DAG can lead to a better network score, and does not introduce cycle. In other words, a candidate parent set cannot be the optimal parent set, if one of its subsets has a better score [Tey05, DCJ11, YM13]. Therefore by pruning away those nodes whose predecessors provide better score from Figure 8 (b) , we obtain a smaller graph in (c), which requires much less space to store. This graph is called the *sparse parent graph* (SPG), which is proposed in Yuan and Malone [YM13]. Each variable has one parent graph, hence one SPG. The scores and the optimal parent sets information provided by SPG are already sufficient for the searching phase. Hence we only need to save the SPG for each variable in memory at the end of the scoring phase. The parent graph consists of exponential number of nodes, and many of them share the same scores. With these duplicate scores pruned out, the SPG is much more efficient in space and search-time. Para-OS actually saves the parent graph of Figure 8 (b). Compare to SPG we can see the apparent reduction in memory usage.

$parents_{X_1}$	$\{X_2, X_3\}$	$\{X_3\}$	$\{X_2\}$	$\{\}$
$scores_{X_1}$	5	6	8	10

Table 5: Sorted scores and parent sets for X_1 after pruning parent sets which are not possibly optimal [YM13].

Yuan and Malone [YM13] also introduce how to implement SPG in order to enable efficient search. Two parallel lists are used to store the SPG, with one list for the scores and another for the associated optimal parent sets. Table 5 illustrates an example to store the SPG of Figure 8 (c). The lists are sorted by the scores in ascending order. The reason for sorting is for fast look-ups. When looking for $BestScore(X_1, \mathbf{U})$ we just scan the table from the beginning, and the first subset of $\mathbf{U}$ encountered is the optimal parent set we need.

4.4.2 Parallelization

In order to minimize the communication between processors, we decide to avoid data retrieval of SPG – the corresponding part of which in Para-OS is $Score(\cdot)$ and $BestScore(\cdot)$. The idea goes as follows. In the scoring phase, each processor is responsible for calculating SPG for one variable, and the SPG is saved locally. For instance, processor 1 would compute and save the SPG of variable X_1. The whole scoring phase is completed before the searching phase starts. In the searching phase, the calculation is divided by the sink (see Figure 9), in order to avoid the data retrieval of SPG. For example, the calculation of variable set $\{X_1, X_2, X_3\}$ is divided as follows. The subnetwork whose sink is X_1 is calculated on processor 1, where $BestScore(X_1, \{X_2, X_3\})$ is computed by the local SPG of X_1 and $DagScore(X_2, X_3)$ is retrieved from other processor. The result returned by processor 1 is the sum of these two scores. Similarly, subnetworks whose sink is X_2 is calculated on processor 2, and subnetwork whose sink is X_3 is calculated on processor 3. The three scores returned by processor 1, 2 and 3 are then compared and reduced to the best score by using MPI_REDUCE. The best score and the associated subnetwork of $\{X_1, X_2, X_3\}$ are then saved to a processor determined by the mapping function (Figure 9). In this way the data communication is limited to retrieving and saving $DagScore(\cdot)$ and the subnetworks.

5 Implementation Process

The implementations of Para-OS and the new version are both in the C programming language. The reason to choose C is that parallelization will be implemented with *Message Passing Interface* (MPI), and there is

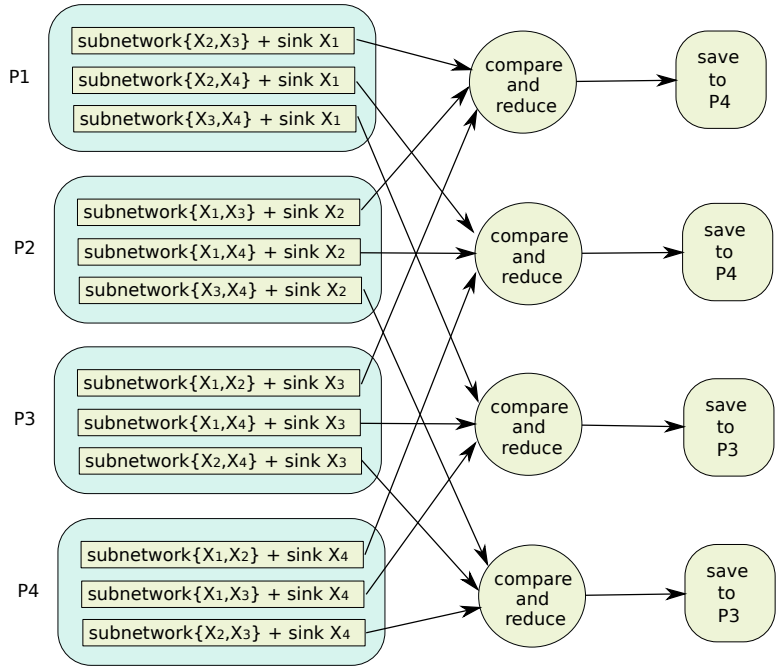


Figure 9: A 4-variable example of parallelization of the searching phase in New Para-OS. For example, P1 (Processor 1) is responsible for the calculation with sink X_1, P2 (Processor 2) for the calculation with sink X_2, etc. The results of same network returned by different processors are compared and reduced to the best one, which is saved to a processor whose ID computed by the mapping function.

robust MPI libary in C. BDeu is chosen as the scoring function, as it is also used in Tamada et al. [TIM11]. BDeu produces maximized score by default, and we multiply it by -1 to transform it to minimized score. An uninformative value of 1 is used as ESS in BDeu. Gathering counts efficiently from dataset is an important task, and it is completed by using AD-tree [ML98]. Distributed data communication is quite intense, especially in Para-OS. Processors have to send queries and replies to each other. In order to speed up the communication, an algorithm is employed for deciding processor sender-receiver pairs during the communication. We did not implement score pruning in Para-OS. Because local scores are stored in a highly dispersed way, score pruning would either introduce more data communication between processors, or we would need to change the algorithm.

5.1 Using AD-tree to Collect Counts

Scoring functions typically require collecting frequent counts of the relevant variables from the dataset. For a local score $Score(X_i, PA_{X_i})$, we need the

count of each particular instantiation of X_i and PA_{X_i}, i.e. all the n_{ijk} in Equation 3 for each possible combination of i, j, k. In the implementation, these are collected by using a data structure called AD-tree [ML98].

The counting task usually involves constructing *contingency tables* (CTs). A CT is a table that displays the multivariate frequency distribution (i.e. counts) of the variables. For a simple dataset with three variables, an example of CT(X_1, X_2, X_3) is depicted in Figure 10. A CT directly shows the count of each instantiation of the relevant variables. Once we have the CT, we will have all the n_{ijk} needed.

Dataset

Row	X_1 X_2 X_3
1	1 1 1
2	1 1 2
3	1 2 1
4	2 2 2
5	2 1 1
6	2 1 2
7	2 2 1
8	3 2 2
9	3 1 1

CT(X_1, X_2, X_3)

X_1	X_2	X_3	#
1	1	1	1
1	1	2	1
1	2	1	1
1	2	2	0
2	1	1	1
2	1	2	1
2	2	1	1
2	2	2	1
3	1	1	1
3	1	2	0
3	2	1	0
3	2	2	1

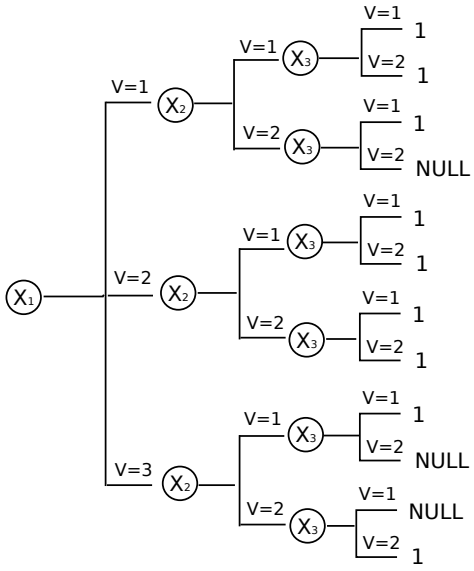


Figure 10: The contingency tables CT(X_1, X_2, X_3) (middle) of a simple dataset (left), and its actually implementation as a tree strucutre (right); the symbol # represents count (adapted from [ML98]).

The *sparse* AD-tree is the actual data structure used for building CTs efficiently. For explanatory purposes we first introduce the *full* AD-tree. The full AD-tree using our simple dataset of Figure 10 is given in Appendix 1. The tree consists of two kinds of nodes: *AD-nodes*, which store the counts that match a particular variable instantiation, and *Vary-nodes*, which are used for expanding the states of a variable. Each Vary-node X_i has r_i subtrees, where r_i is the number of states of X_i. Vary-nodes are children of AD-nodes. If an AD-node has parent Vary X_i, then its children are: Vary X_{i+1}, Vary X_{i+2}, ..., Vary X_n. The full tree explicitly stores all the counts of possible variable instantiations. To look up a count in the tree is straight forward. For example, the count of an instantiation ($X_2 = 2, X_3 = 1$) follows this path: the root node → Vary-node X_2 → AD-node $X_2 = 2$ → Vary-node X_3 → AD-node $X_3 = 1$ (Appendix 1), and the count is 2.

The number of nodes in the full AD-tree grows very fast with the number

of variables (exactly $\prod_{i=1}^{n}(r_i + 1)$ nodes). A sparse version of AD-tree is clearly needed in order to collect the counts efficiently. To construct a sparse AD-tree, the general principle is to not allocate memory for those counts that are deducible from other counts. First, nodes with counts of zero are erased, and simply a NULL is stored instead. Second, a variable called *the most common value* (MCV) is introduced, which stores the most common value of X_i in the Vary-node. Furthermore, the count in the MCV-th subtree can be deduced from counts of other subtrees. Therefore the MCV-th subtree is replaced by NULL. Third, it is not worth expanding the subtree when the count is too small. For an AD-node with count less than a user-defined threshold, a *leaf-list* is maintained instead of a subtree. A leaf-list is a list of pointers into the dataset, simply listing the datapoints that match the instantiation of the current AD-node. In this way the AD-tree is not expanded fully to its leaves. The sparse AD-tree of the simple dataset in Figure 10 is shown in Figure 11. Comparing the sparse version with the full tree in Appendix 1, we can see that the full tree has 38 nodes while the sparse one has only 14 (among which 4 of them are NULL nodes).

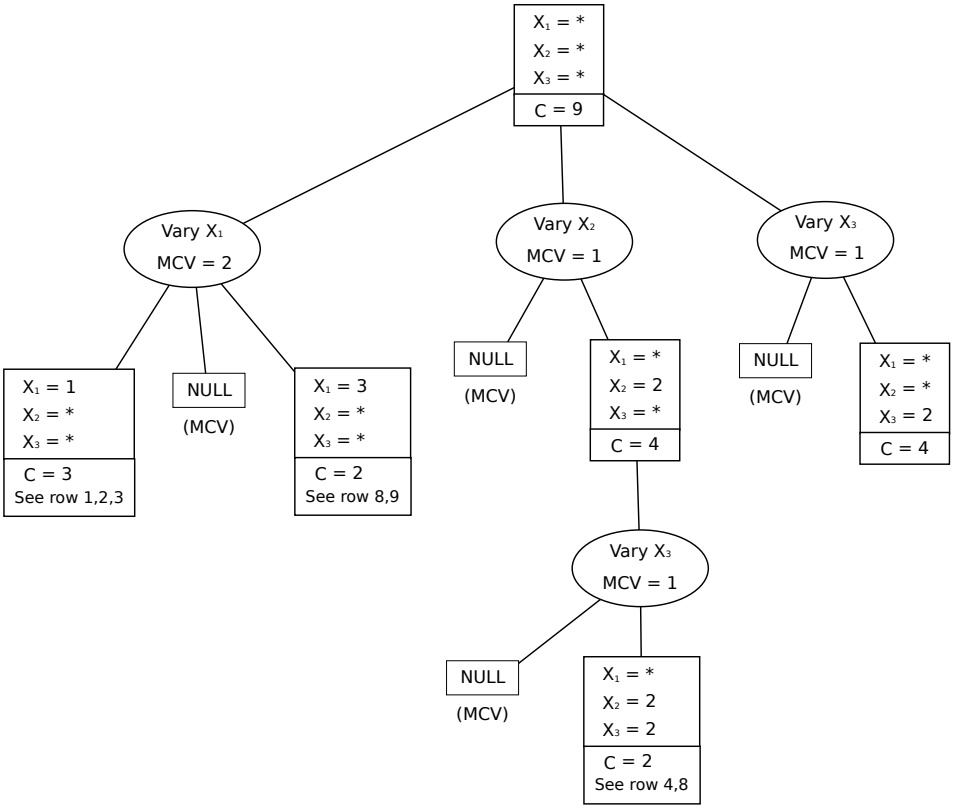


Figure 11: The sparse AD-tree built from the dataset in Figure 10 (adapted from [ML98]).

Given a sparse AD-tree, a CT can be computed recursively. For example $\mathrm{CT}(X_2|X_1 = 1)$, $\mathrm{CT}(X_2|X_1 = 2)$ and $\mathrm{CT}(X_2|X_1 = 3)$ can be combined to produce $\mathrm{CT}(X_1, X_2)$ (see Figure 12). The count in the MCV-th subtree can be calculated from other subtrees. For example when constructing $\mathrm{CT}(X_1, X_2)$ in Figure 12 from the sparse tree in Figure 11, the MCV-th subtree $X_1 = 2$ is deliberately set to NULL. All other counts in the CT are easily found from the sparse tree (in this particular case by checking leaf-lists), except $\#(X_1 = 2, X_2 = 1)$ and $\#(X_1 = 2, X_2 = 2)$. Both of them can be deduced as follows:

$$\begin{aligned}\#(X_1 = 2, X_2 = 1) &= \#(X_2 = 1) - \#(X_1 = 1, X_2 = 1) - \#(X_1 = 3, X_2 = 1) \\ &= 5 - 2 - 1 = 2.\end{aligned}$$

$$\#(X_1 = 2, X_2 = 2) = \#(X_1 = 2) - \#(X_1 = 2, X_2 = 1) = (9 - 3 - 2) - 2 = 2.$$

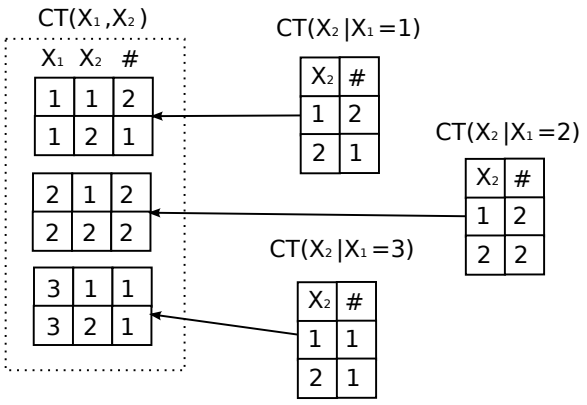


Figure 12: The recursive building of $\mathrm{CT}(X_1, X_2)$ from three smaller conditional CTs (adapted from [ML98]).

5.2 Parallelization in MPI

Message Passing Interface (MPI) is a standardized and portable system for message-passing on parallel computers. Most of the TOP500[5] supercomputers have MPI installed. The MPI implementations are well optimized and use the fast interconnects provided by supercomputers. Para-OS has a great amount of communication between the processors, so it benefits from the use of these fast interconnects. MPI mainly supports Fortran, C and C++ programming language.

MPI supports both point-to-point and collective communication. It works by creating communication channels between independent processors. In the beginning of the program, there is a single global communicator which can

[5]http://www.top500.org/

be used in communicating to all of the processors. Each of the processors is assigned a numeric rank from 0 to $n_p - 1$. It is also possible to split the main communicator into subsets, which consist of only some of the processors. After creating communication channels (or using the initial ones), there are different ways to actually send and receive messages. In addition to synchronous and asynchronous versions of normal *send* and *receive*, there are multiple other operations. For example, it is possible to *reduce*, which is to reduce values on all processors to a single value (e.g. by taking the maximum value or the sum); and *barrier*, which is to block until all the processors to arrive at that position.

We give a simple example of MPI in the following C code (Listing 1). This example shows a simple transaction between two MPI processes. MPI works with single executable, and inside the executable code we need to decide which process does what. Processes are asigned with MPI ranks which can be acquired using MPI_Comm_rank. In this example process with rank 0 will send count number of INTEGERS to process rank 1, which will receive these INTEGERS. In the end all the MPI processes will wait until all of them reach barrier. There could be more than two processes runing this piece of code, and the processes other than 0 and 1 would just directly go to the barrier to wait for processes 0 and 1.

Listing 1: C code example for MPI

```
int rank, tag = 1, count = 10;
int send_buffer[10] = {1,2,3,4,5,6,7,8,9,10};
int receive_buffer[10];
MPI_Status status;
if (rank == 0) {
    MPI_Send(send_buffer, count, MPI_INT, 1, tag,
             MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recieve(receive_buffer, count, MPI_INT, 0, tag,
                MPI_COMM_WORLD, &status);
}
MPI_Barrier(MPI_COMM_WORLD);
```

The challenges in the implementation process include handling deadlocks, retrieving data efficient by cutting as much communication as possible, algorithm design choices – such as considering multi-threading versus single-threading, and asynchronous send versus synchronous send, etc.

Para-OS distributes the calculation of a level of dynamic programming over several processors. There are two different types of MPI transfers. One is *data request*, which asks for data from other processors and results in a reply containing the requested data (after the queried processor has found it). Another type is *data update*, which sends the results of calculation to other processors to be stored or updated. In the case of data request, the mapping function guarantees that the requested data is found from the queried processor. If it is not found, then there is an error in the program.

The data communication for the searching phase mainly happens in a function named PROCESS-S (Algorithm 1 from [TIM11]), which could run on a processor more than once (during the same level of dynamic programming). In addition, different processors may perform their work at different speed. Therefore, the MPI communication for data retrieval happens in a intervening/asynchronous fashion – the data is retrieved whenever it is needed. MPI communication for data update is also done in the same way – the result is sent for storage whenever it is produced. During the implementation, we noticed this one-by-one sending method generated quite much communication, and made it hard to debug because the communication is spread throughout the calculation code. Therefore, we group these intervening data retrieval and storing MPI calls into their own phase.

The work for a single processor is divided into three phases: data request phase, calculation phase and result storing phase. These three phases are also illustrated in the example of Figure 7, where the upper rectangle box contains the data to be retrieved in the data request phase, and the lower rectangle box contains the results from the calculation phase – these results are sent for storage in the last phase. In the data request phase, we calculate what data need to be retrieved for each processor. Then, all of the sends for the same destination are grouped into one big send. This design makes the program "cleaner" – easier to implement and debug. It can also speed up the communication when there are multiple queries between two processors. For example, if Processor 1 (P1) needs three pieces of data from Processor 2 (P2), originally P1 would send three queries to P2 and receive three replies – together six MPI transfers. Now with the grouping, P1 would send only one query to P2 and receive one reply – two MPI transfers. All of the necessary data are retrieved in this phase, before the calculation starts. The results are calculated during the calculation phase. In the result storing phase, sends for the same destination are also combined into one big send in the similar way.

5.3 Parallelization in OpenMP

The implementation of New Para-OS uses a hybrid parallelisation strategy utilizing both MPI and OpenMP. OpenMP is a way to parallelize tasks in unified memory architecture machines like computing nodes in cluster and supercomputing environment. In such environments a number of physical processors are divided to multiple processor cores.

OpenMP works with the help of compiler support to simplify the multithread programmin in C, C++ and Fortran. As different computers have different number of available processing cores, OpenMP can be dynamically adjusted on runtime about how many threads will be created. In the program we use OpenMP to parallelise BDeu-score calculation and the searching phase. Of these two parts the BDeu-score calculation acquires most of the benefits because it can be parallelized easily. Next we illustrate how OpenMP

achieves parallelization through an example piece of code.

The piece of code showned in Listing 2 embeded two *pragmas*, which is a method (specified by the C standard) for providing the compiler with additional inputs. The first pragma (#pragma omp parallel for schedule(dynamic, 100)) conveys the following information: tells OpenMP to parallelize the following *for loop*, by using *dynamic* way of scheduling and chunk size 100[6]. The *count* number of items is divided in chunks of 100 which are computed in parallel by the threads. The number of threads is set as environment variable before running the program. The purpose of the second pragma (#pragma omp critical) is to create a critical section, which gives the order "only one thread can execude the following code at a time".

Listing 2: C code example for OpenMP

```
#pragma omp parallel for schedule(dynamic, 100)
for (int j = 0; j < count; j++) {
    uint64_t parent_set = parent_set_array[j];
    float score = calculate_BDeu_local_score(variable,
        parent_set, hyper_param, adn, dataset);
#pragma omp critical
    save_to_sparse_parent_graph(spg, parent_set, score);
}
```

In general, we realize the parallelisation with OpenMP by adding C pragmas to tell how parallelisation should be done, and to remove any writable global state in order to prevent two threads access and change same memory area at the same time.

5.4 Deciding Processor Sender-Receiver Pairs

During data communication, each processor knows "where the data I need are located/where I should send data to be stored" through the mapping function, but each processor does not know "who will need data from me/who will send data to me to be stored." This situation arises in both Para-OS and New Para-OS. To make things simple, each processor sends inquiries to everyone else in the communication stage. This scenario leads to the natural question of how to pair the processors, because communication in a serialized fashion where only one pair is exchanging data at a time would be too slow. If we can match all of the processors in k pairs at each time step, then all k pairs will communicate at the same time.

Suppose there are four processors: P0, P1, P2 and P3. All of them can finish the task of talking with everyone else by using three time steps, as shown in Matrix 1: at time step one we have P0 $\leftrightarrow$ P1 (that is, P0 sends query to P1, at the same time P1 sends query to P0) and P2 $\leftrightarrow$ P3; at time

[6]Here 100 is an example. It is not necessary the best value, but using value 1 will result in poor performance.

step two we have P0 ↔ P2 and P1 ↔ P3; at time step three P0 ↔ P3 and P1 ↔ P2.

For the case of eight processors, the pairing matrix is illustrated in Matrix 2. At time step one we have P0 ↔ P1, P2 ↔ P3, P4 ↔ P5 and P6 ↔ P7; at time step two P0 ↔ P2, P1 ↔ P3, P4 ↔ P6 and P5 ↔ P7, and so forth. Observing the matrix we can divide it into four 4-by-4 sub-matrices, where the top-left one is same as the bottom-right one, and the top-right one is same as the bottom-left one. This repetition is recursive – each of the sub-matrix can be further divided into four, until we reach 2-by-2 sub-matrices. Hence this sender-receiver pairing matrix can be generated in a recursive way. A recursive algorithm (Algorithm 1 GET_PID) takes in three parameters: row index *row*, column index *column*, the number of processors n_p, and returns the processor id in position $(row, column)$ of the matrix. That is, given a processor (row) and a time step $(column)$, the algorithm finds the pairing for that processor and time step. For example, in Matrix 2 calling GET_PID$(4, 2, 8)$ asks whom Processor 4 in time step 2 should talk with, in a 8-processor scenario. The function returns the answer Processor 6. The best case of GET_PID has time complexity 1, and the worst case calls the recursive procedure $(\log_2 n_p - 1) * 2$ times. For instance when $n_p = 64$ the procedure gets called 10 times in the worse case. This simple algorithm assumes the number of processors is in the power of 2, which fits our testing plan for Para-OS. We next describe a generalization to cases where the number of processors is not in power of 2.

0	1	2	3
1	0	3	2
2	3	0	1
3	2	1	0

0	1	2	3	4	5	6	7
1	0	3	2	5	4	7	6
2	3	0	1	6	7	4	5
3	2	1	0	7	6	5	4
4	5	6	7	0	1	2	3
5	4	7	6	1	0	3	2
6	7	4	5	2	3	0	1
7	6	5	4	3	2	1	0

Matrix 1 (left): four processors; Matrix 2 (right): eight processors. Index of each row starts from 0 in a matrix. The element of index 0 in each row indicates the processor of this row. The element of index 1 tells whom the processor will communicate with at time step one; the element of index 2 tells whom the processor will communicate with at time step two, and so forth.

In New Para-OS the number of processors in use is the same as the number of variables because the design of the parallelization. It is unrealistic to limit the number of variables to be in power of 2. Because the pairing matrix is highly symmetrical, when the number of processors is not in power

Algorithm 1 an algorithm for determining processor pair

```
1: procedure GET_PID(row, column, n_p)
2:     if row = 0 then
3:         return column
4:     else if row = 1 then
5:         if column mod 2 = 0 then
6:             return column + 1
7:         else
8:             return column − 1
9:         end if
10:    else if column = row then
11:        return 0
12:    else if column + row = n_p − 1 then
13:        return n_p − 1
14:    else if row < n_p/2 and column < n_p/2 then
15:        return GET_PID(row, column, n_p/2)
16:    else if row >= n_p/2 and column < n_p/2 then
17:        return GET_PID(column, row, n_p)
18:    else if row >= n_p/2 and column >= n_p/2 then
19:        return GET_PID(row − n_p/2, column − n_p/2, n_p/2)
20:    else if row < n_p/2 and column >= n_p/2 then
21:        return GET_PID(row, column − n_p/2, n_p/2) + n_p/2
22:    else
23:        Report error
24:    end if
25: end procedure
```

of 2 it could happen that one side of the pair is missing. Therefore we should take care of these missing sides of processor-pairs. The idea is to rise the number of time steps from n_p to the next power of 2 (build a matrix that "pad" to a power of 2), but skip pairs that include the padded processors. An example is the case with five processors. A matrix that "pad" to a power of 2 would include Processors 5 to 7, but they are skipped because Processor 4 is the highest processor ID in use (see Matrix 3). Here Processor 4 does not fit to the normal symmetry, and we handle remaining cases sequentially (see Figure 13 (right)). Therefore, in the case of five processors we require seven time steps. See Matrix 4 for the case of six processors.

0	1	2	3	4	x	x	x
1	0	3	2	x	4	x	x
2	3	0	1	x	x	4	x
3	2	1	0	x	x	x	4
4	x	x	x	0	1	2	3

0	1	2	3	4	5	x	x
1	0	3	2	5	4	x	x
2	3	0	1	x	x	4	5
3	2	1	0	x	x	5	4
4	5	x	x	0	1	2	3
5	4	x	x	1	0	3	2

Matrix 3 (left): five processors; Matrix 4 (right): six processors (x indicates the skipped processors, which means no communication at that time step).

Ideally the algorithm should be called n_p times in the general case. Unfortunately our solution needs to use "padded" processors, which results in calling the algorithm j times more, where j is the number of "padded" processors. In the sequential part of the communication (for example the communication with Processor 4 in Figure 13 (right)), we simply lose the concurrency. Still the overall time spent in communication is faster than the pure serial case (see example of five processors in Figure 13).

Blocking communication is used for MPI in order to simplify program design. That is, all MPI commands such as send and receive block execution of the program until the commands finish. For instance after sending a query, the sender blocks until the reply is received.

6 Performance Evaluation

The experiments of both Para-OS and New Para-OS were run on the supercomputer Taito[7] at CSC[8], which possesses 794 Haswell processors (12-core Intel Xeon Haswell E5-2690v3 64bits), with 64 GB memory available per processor[9]. The datasets used for experiments are from the UCI machine

[7]https://research.csc.fi/taito-phase2-hardware

[8]IT Center for Science Ltd. www.csc.fi

[9]Taito also has Sandy Bridge processors but our experiments were run on the Haswell processors.

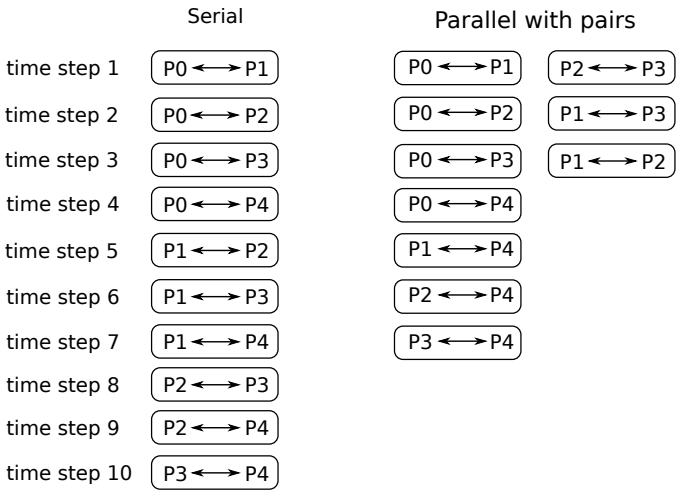


Figure 13: A 5-processor example of serial style data communication (left) versus parallel style with pairs (right). In the parallel style Processor 4 (P4) is left out from pairing and we handle communication with P4 sequentially.

learning repository and processed as csv files, and are available online[10]. Most of the experiments were run with binary datasets Alarm and Horse, with single-valued columns eliminated. Some preliminary tests showed that non-binary datasets are slower to process.

For the experiments, each core in a CPU is treated as a single processor, which means one MPI process runs in a single core. For the case of OpenMP, one MPI process uses multiple cores where each one has one OpenMP thread running on it.

BDeu is quite sensitive to the value of the ESS. Here we used the uninformative value 1 for testing. In the experiments we also noticed that it takes quite long time to calculate BDeu with network size larger than 20. However we did not implement other scoring functions.

6.1 Para-OS

We considered four experimental setups: (1) σ evaluation for different values of σ with fixed n (network size), n_p (number of processors) and m (number of datapoints); (2) scalability evaluation with different n_p with fixed n, σ and m; (3) runtime (wall-clock time) evaluation for various number of variables n with fixed n_p, m and σ; (4) memory consumption evaluation for various n and n_p (with fixed m and σ).

[10] *Published datasets* from https://sites.google.com/site/bmmalone/files/urlearning

6.1.1 σ Evaluation

We performed the σ experiments with $n = 23$, $m = 50$, $n_p = 256$ and $\sigma = 1, ..., 10$ on Taito. The dataset is generated by extracting 23 columns and 50 rows from binary dataset Alarm (ignoring single-valued columns). The results are shown in Figure 14 and Table 6. In the original Para-OS paper [TIM11] σ was tested with values ranging from 1 to 5 with $n = 23$ and $n_p = 256$. The memory increases as σ changed from 1 to 5. Our results are consistent with theirs. These results are expected because larger σ values mean each processor is assigned with more tasks – more order graph nodes. However, when σ value grows too large, for example over 6 in our experiments, the average memory consumption starts to drop. This is because of the interaction of the parallelization and σ. In particular, all calculations for the order graph levels $n - \sigma$ onward are executed on a single processor. For example, if $n = 23$ and $\sigma = 10$, then from level 13 onward all calculations will be on one processor[11]. Therefore caution should be used when selecting σ. This is probably why they did not test σ value larger than 5 in [TIM11].

In the σ experiments ($\sigma = 1, ..., 5$) of [TIM11], runtime decreased as σ changed from smaller to larger value. However we get the opposite result (Figure 14). We suspect this is because runtime in their implementation is communication bound, while our implementation is more CPU-bound due to the communication optimizations described in Section 5.2. There are redundant calculation in the parallelization, and larger σ values increase the redundancy. Hence, our runtime increases slightly each time σ value grows by one. Consequently, in our implementation $\sigma = 1$ yields the best runtime and memory use, as there is the least overlapping of calculation and the smallest amount of tasks assigned to each processor. In other words, the space-communication trade-off controlled by σ (described in [TIM11]) is changed due to the more efficient communication in our implementation. Because $\sigma = 1$ produces the best results, we decide to run rest of the experiments with this σ value.

6.1.2 Scalability Evaluation

In order to repeat the scalability test in [TIM11], we performed the experiments of network size $n = 23$ for $n_p = 1, 8, 16, 32, 64, 128, 256, 512, 1024$ on Taito, and $\sigma = 1$ (the test in [TIM11] used $\sigma = 3$). The dataset used is the extracted 23-column, 50-row Alarm, same as in the σ experiments. The results are displayed in Table 7. According to [TIM11], speedup $S(n_p) = T(1)/T(n_p)$, where T refers to runtime. The parallelization efficiency $E(n_p) = S(n_p)/n_p$, and $E(n_p) >= 0.5$ is considered good parallelization.

In [TIM11] they drew the conclusion that $n_p = 256$ is the most efficient case for Para-OS (the efficiency was 0.74 for $n_p = 256$ and 0.59 for $n_p = 512$

[11] To see this, please refer to the peseudocode of Para-OS [TIM11].

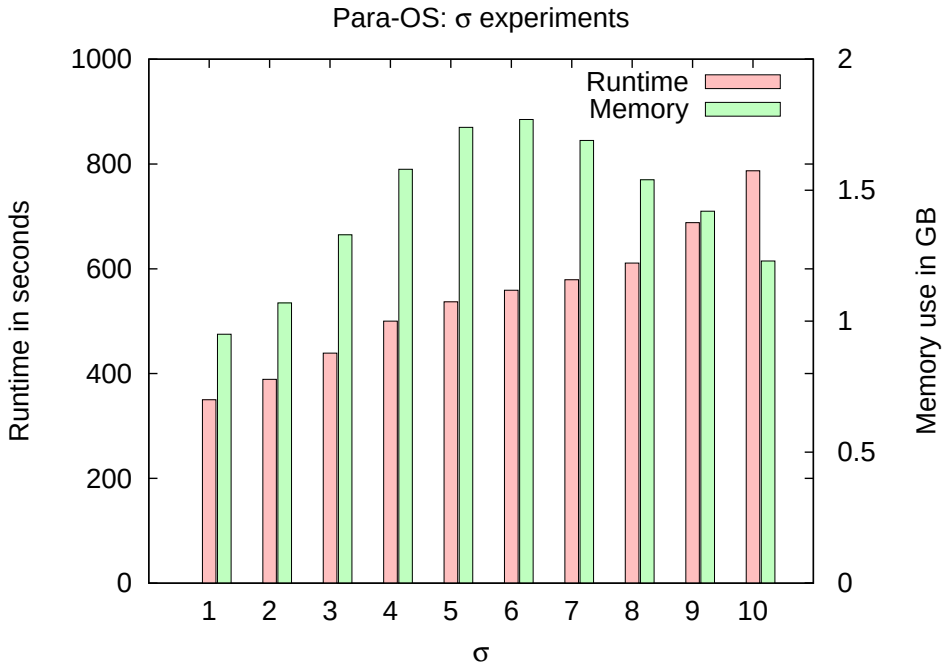


Figure 14: σ experiments of Para-OS: runtime and average memory consumption (per processor) with $n = 23$, $n_p = 256$ and $\sigma = 1, ..., 10$ (in logarithmic scale).

σ	1	2	3	4	5
M	0.95	1.07	1.33	1.58	1.74
T	350	389	439	500	537
σ	6	7	8	9	10
M	1.77	1.69	1.54	1.42	1.23
T	559	579	611	688	787

Table 6: Runtime (T) and memory requirement (M) results of σ experiments ($n = 23$, $n_p = 256$) for Para-OS. Memory in the table refer to average memory consumption per processor, and in GB; runtimes are in seconds.

in their result). However in our experiments both $n_p = 256, 512$ yield good efficiency (0.78 and 0.70, respectively). As shown in Table 7, our efficiencies drop slower than theirs when n_p grows larger than 128. We can say that our communication part of the program is more efficient than theirs, and our program scales better (however our program is less memory efficient). Figure 15 depicts the runtime and efficiency trends. The memory requirements per processor for both $n_p = 256, 512$ are also better (less than 1 GB) than other n_p. Therefore we decide to perform the runtime and memory experiments mainly on $n_p = 256, 512$.

n_p	M	T	S	E	E([TIM11])
1	8.86	72029	1.0	1.0	1.0
8	3.36	9508	7.58	0.97	0.96
16	2.49	4802	15.00	0.94	0.94
32	1.89	2625	27.44	0.86	0.92
64	1.45	1303	55.28	0.86	0.89
128	1.14	677	106.39	0.83	0.83
256	0.96	362	198.98	0.78	0.74
512	0.85	202	356.58	0.70	0.59
1024	0.80	153	470.78	0.46	0.39

Table 7: Results of Para-OS's scalability test on various number of n_p for network size $n = 23$. Memory (M) in the table refer to maximum memory consumption per processor in GB. Runtimes (T) are in seconds. S represents speedup ($S(n_p) = T(1)/T(n_p)$) and E represents efficiency ($E(n_p) = S(n_p)/n_p$). The efficiencies reported in [TIM11] are also shown in the table for comparison.

6.1.3 Runtime Evaluation

Runtime (wall-clock time) experiments for various network size are run by using 256, 512 and 1024 processors on Taito, with $\sigma = 1$. We used two groups of binary datasets Alarm (run on $n_p = 256, 512, 1024$) and Horse (run on $n_p = 256, 512$), and they are generated by extracting n binary columns and 50 rows from the original datasets Alarm/Horse. Tamada et al. used $n_p = 256$ and $\sigma = 3$ for runtime test in [TIM11]. Figure 16 illustrates runtimes of $n = 16, ..., 25$ in logarithmic scale, and results are recorded in Table 8.

Figure 16 shows that the runtime increases exponentially as n grows. The timing results suggest that using 1024 processors for small datasets ($n < 23$) is not worthwhile, as initialization of the MPI processes takes longer time than actual running. The implementation does not calculate the communication time and scoring time separately, unfortunately. In Table

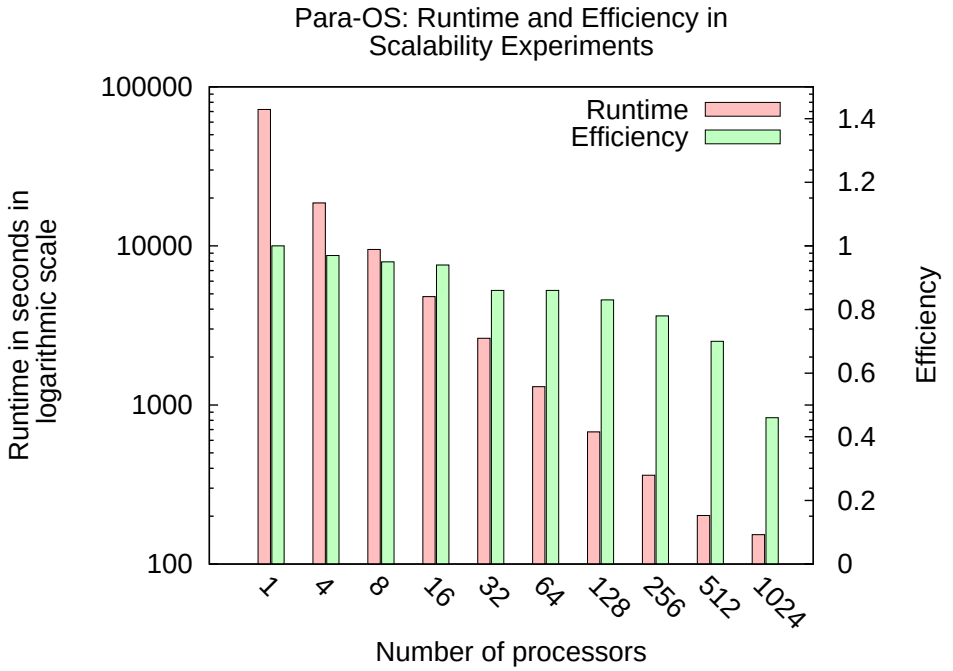


Figure 15: Scalability experiments of Para-OS: runtime and speedup for $n = 23$, $m = 50$ and $\sigma = 1$.

8, we also present the results from [TIM11] for a general comparison. In [TIM11] they used their own artificially generated dataset, which is not publicly available. From some preliminary tests we see that different datasets (for example, binary or non-binary) can affect runtime. Some other factors such as computing power of different supercomputers can play a role in affecting runtime too. However, we can say that our timing results are on par with those reported by Tamada et al.[TIM11], since their timing results are not significantly different.

6.1.4 Memory Consumption Evaluation

The next set of experiments evaluates the memory consumption for $n = 16, ..., 25$ and $n_p = 256, 512$, with $\sigma = 1$. Two groups of binary datasets Alarm and Horse were used, same as in the runtime experiments. The maximum memory consumption per processor are depicted in Figure 17. The results of dataset Horse can be found from Table 9 (the results produced by the same n in Alarm and Horse are nearly identical, therefore no need to repeat both).

We can see that the memory use is exponential as n grows, which is expected. The memory-use per processor for $n_p = 512$ is slightly less than $n_p = 256$. This is because the intermediate data are stored in a dispersed fashion, hence the case of $n_p = 512$ has less the load per processor than

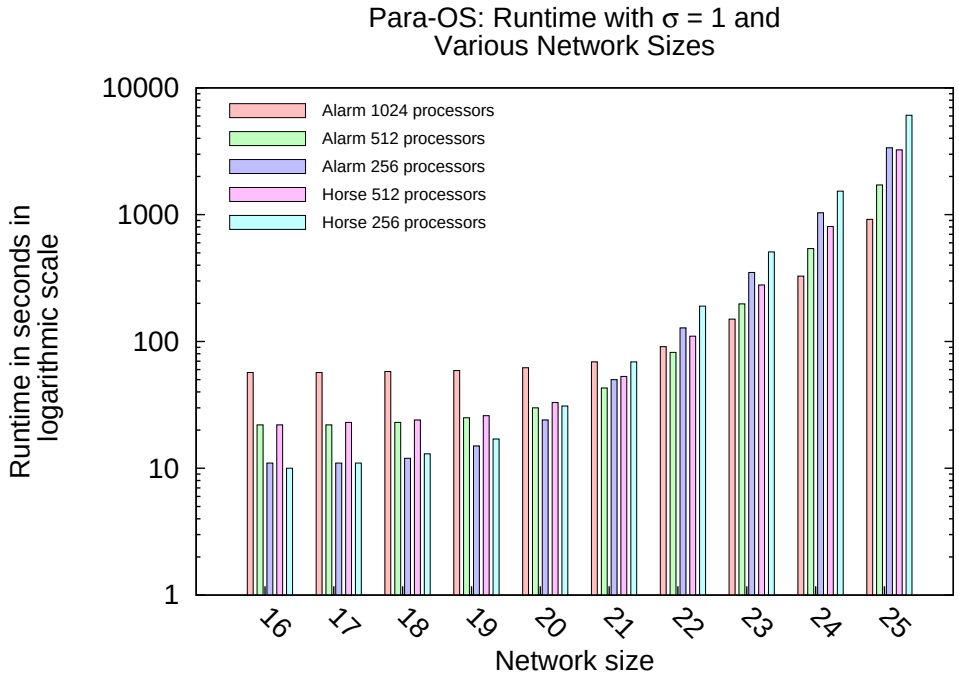


Figure 16: Runtime experiments of Para-OS (in logarithmic scale) with $n_p = 256, 512, 1024$ and $\sigma = 1$ using two datasets Alarm and Horse ($n_p = 1024$ only ran with Alarm).

n	$T_A(256)$	$T_A(512)$	$T_A(1024)$	$T_H(256)$	$T_H(512)$	$T(256)$[TIM11]
16	11	22	57	10	22	-
17	11	22	57	11	23	-
18	12	23	58	13	24	-
19	15	25	59	17	26	-
20	24	30	62	31	33	106
21	50	43	69	69	53	262
22	128	82	91	190	110	546
23	350	198	150	508	279	1073
24	1035	541	328	1535	807	2645
25	3367	1713	919	6082	3241	5386

Table 8: Runtime (in seconds) experiments on $n_p = 256, 512$ with datasets Alarm/Horse, and $n = 1024$ on Alarm. T_A represents runtime for dataset Alarm, and T_H for dataset Horse. Rutimes on $n_p = 256$ reported in [TIM11] are also shown in the table for comparison.

$n_p = 256$. However the total memory-use for $n_p = 512$ is more than $n_p = 256$. This is because the amount of redundant calculation increases with n_p, unfortunately. In addition, a larger n_p also means more communication between processors, which also results in more memory-use due to our communication strategy. The memory consumption of Para-OS presented in [TIM11] is much less than ours, which highlights the different memory-communication tradeoffs made by the implementation. The difference in memory-use may be caused by the following several factors. We use a datastructure (UINT64) that allows $n > 32$ (which raises the memory-use significantly compared to UINT32). The communication optimizations described in Section 5.2 increases the memory consumption. Using Hash table for the intermediate results storage may not be a compact enough solution, and the efficiency of Hash table could be improved by better hashing algorithm.

We tested Para-OS on various datasets of 25 variables and 100 datapoints as well. The results will be presented in next section and compare with the results of New Para-OS. To summarize the experiments of Para-OS, it is obvious that their implementation of Para-OS in [TIM11] makes a different memory-communication tradeoff than our implementation. Through experiments it has become quite evident that many choices made during code writing phase affect the performance of Para-OS. Many of these implementation details are not obvious in [TIM11], such as how to choose communication patterns between MPI nodes, data structures used for storing the intermediate results, etc. Because of this, some of our experiment results differ from theirs.

6.2 New Para-OS

For New Para-OS, the default design was to use the same number of processors as the network size. In addtion to $n_p = n$, we were able to use multi-threading in the experiments. For the scoring phase, we used threading to parallelize the BDeu score calculation; while for the learning phase, we parallelized the search in SPG.

6.2.1 Runtime Evaluation

We performed two sets of runtime experiments for various network sizes. The first set is the runtime test with no threading, and the second set utilizes 12 and 24 threads on each processor. Threading is implemented by OpenMP as described in Section 5.3. Datasets were generated by extracting $n = 17, ..., 26$ columns and $m = 50$ rows from the binary dataset Alarm, ignoring single-valued columns. The experiments were run on Taito. Figure 18 illustrates the runtimes, and the results are found from Table 10 together with the speedup and efficiency of 12-threads per physical processor. As expected, runtime growth is exponential. Without threading, runtime for 26-node network

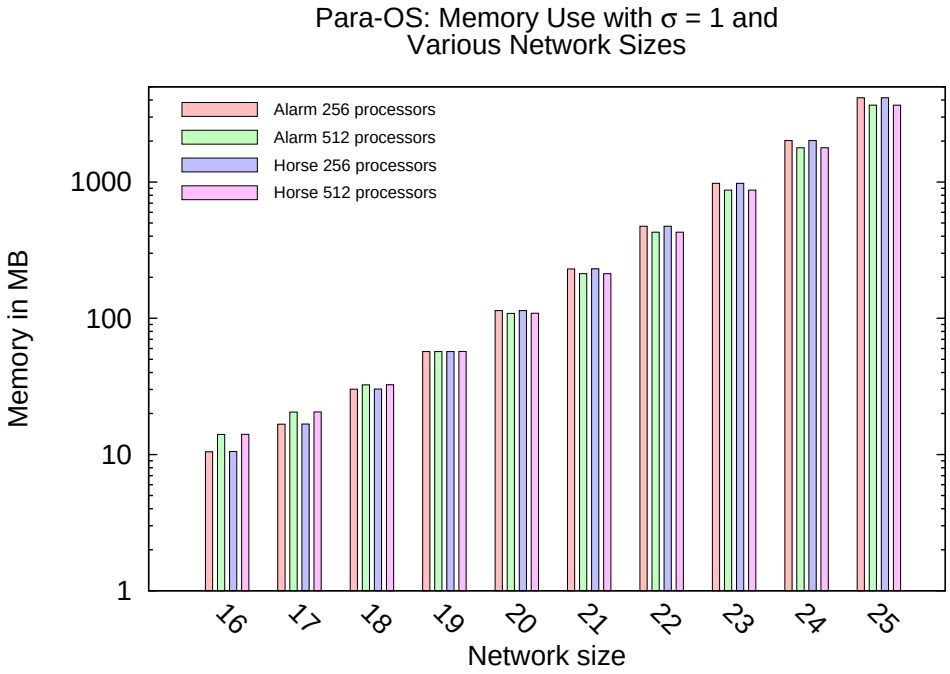


Figure 17: Para-OS's maximum memory consumption per processor (in GB) on $n_p = 256, 512$ with datasets Horse.

n	$M_{max}(256)$	$M_{total}(256)$	$M_{max}(512)$	$M_{total}(512)$	$M(256)$[TIM11]
16	0.01	2.60	0.01	6.96	-
17	0.02	4.16	0.02	10.17	-
18	0.03	7.54	0.03	16.19	-
19	0.06	14.23	0.06	28.51	-
20	0.11	28.41	0.11	54.31	5.77
21	0.23	57.58	0.21	106.18	8.00
22	0.46	118.42	0.42	214.07	10.97
23	0.96	244.46	0.85	436.35	15.01
24	1.97	503.90	1.74	891.55	20.55
25	4.06	1038.77	3.58	1832.45	28.50

Table 9: Results of Para-OS's memory consumption experiments in GB (using dataset Horse). M_{max} refers to maximum memory consumption per processor, and M_{total} the total memory-use. The last column copies the results on $n_p = 256$ from [TIM11] for comparison.

was 80972 seconds (about 22 and a half hours); while with 12 threads, it dropped to 8435 seconds (about 2.3 hours). Judging from the efficiency, we can say that for network sizes larger than 20, it is worthwhile to apply this 12-threaded version.

Timing results of 24 threads were not much faster than 12 threads, for small n it was even slower. This is because the way multi-threading works on modern multiprcocessor NUMA (Non Uniform Memory Access) architecture. When the number of cores were increased from 12 to 24, half of these core have to access across the local NUMA boundary, which slows down the calculation. In addition, creation and destroying of the threads are not free, hence when number of threads raises the cost of handling them also increases. According to the results, it is more benificial to employ more threads for larger networks. For example, 26-variable dataset benefits most from using 24 threads in our results – runtime drops from 12-thread's 8435 seconds (about 2.3 hours) to 24-thread's 5988 seconds (about 1.6 hours).

In the runtime experiments, we also recorded strictly the runtime of scoring phase (including local score calculations, score pruning and creation of the SPGs) and the learning phase (including expanding order graph and communication) (Figure 19 and Table 11). In addition to these two phases, time is also used in setting up MPI, reading in dataset, and building AD-tree, etc. From the results we can see that scoring phase quickly becomes dominant in runtime as n grows (especially in the results of no threading). Therefore it is worthwhile to investigate further parallelism on this phase. Calculating local scores is what really takes time in the scoring phase, and all current exact methods for learning optimal BN structures must perform this step. Once we acquire the local scores, creating the SPGs basically just consists of sorting the scores. The results of multi-threading (especially 24 threads) show that the scoring phase benefits more from parallelization than the learning phase. This is because the BDeu scoring function is more easily parallelizable than the learning phase.

6.2.2 Memory Consumption Evaluation

Next we performed memory consumption experiments for network sizes $n = 17, ..., 26$, where $n_p = n$. Figure 20 depicts the maximum memory consumption per processor and total memory-use, and Table 10 shows the results in GB.

The memory consumption also grows exponentially (for $n > 20$), as expected. However comparing to our implementation of Para-OS, this new version provides a sharp reduction on memory use. For example, to run a dataset of 25 variables on 25 processors, the total memory-use of New Para-OS is 14.50 GB, while the same for Para-OS is 1 TB when $n_p = 256$. By observing how much the memory consumption grows from 25-variable network to 26-variable network, we extrapolate that a 30-variable network

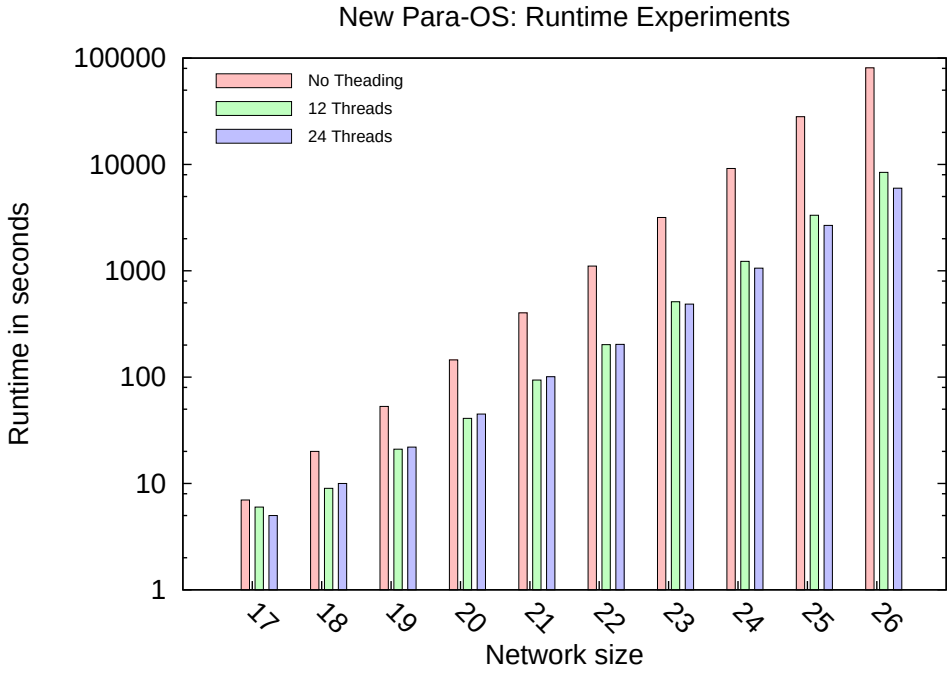


Figure 18: Runtime results (in seconds) of New Para-OS for $n = 17, ..., 26$, $m = 50$ and $n_p = n$ with no threading, 12 threads and 24 threads.

n	T	T_{12}	T_{24}	S_{12}	E_{12}	M_{max}	M_{total}
17	7	6	-	1.17	0.10	0.25	4.28
18	20	9	10	2.22	0.19	0.25	4.55
19	53	21	22	2.52	0.21	0.26	4.86
20	145	41	45	3.54	0.29	0.26	5.22
21	402	94	101	4.28	0.36	0.27	5.71
22	1106	202	203	5.48	0.46	0.29	6.44
23	3174	512	486	6.20	0.52	0.34	7.71
24	9153	1228	1060	7.45	0.62	0.42	9.98
25	28163	3331	2676	8.45	0.70	0.58	14.50
26	80972	8435	5988	9.60	0.80	0.90	23.22

Table 10: Results of runtime and memory-use experiments for New Para-OS. Runtimes are in seconds (T represents runtime without threading, T_{12} runtime with 12 threads, and T_{24} runtime with 24 threads). S_{12} and E_{12} represent the speedup and efficiency of applying 12 threads, respectively. M_{max} refers to the maximum memory consumption per processor in GB, and M_{total} the total memory-use.

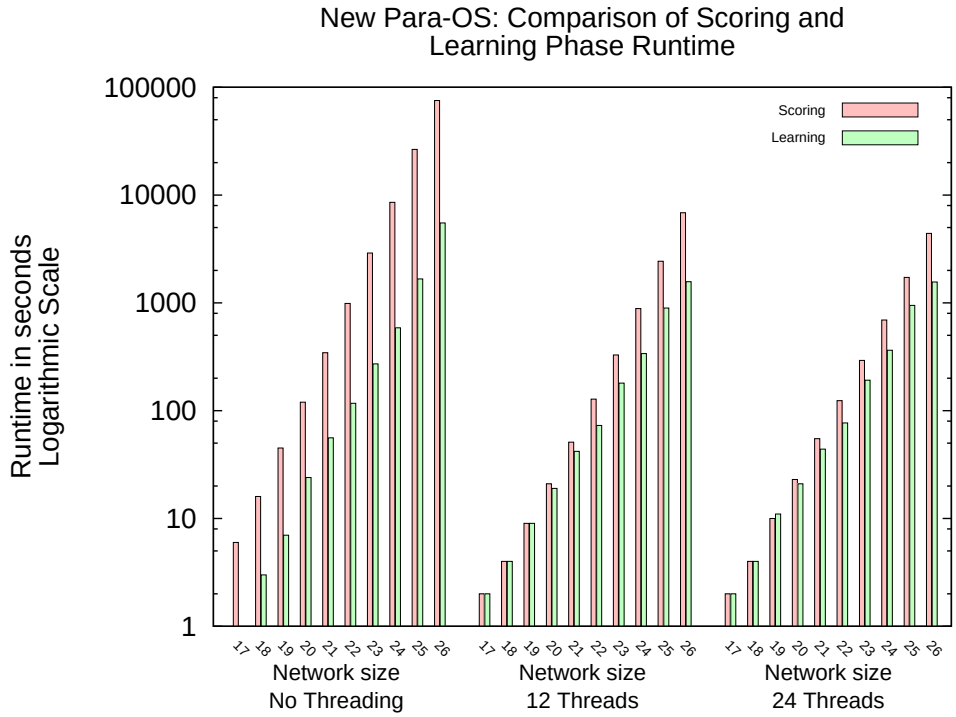


Figure 19: Runtime used by scoring phase and learning phase for $n = 17, ..., 26$, $m = 50$ and $n_p = n$ with no threading, 12 threads and 24 threads.

n	17	18	19	20	21	22	23	24	25	26
T(learning)	1	3	7	24	56	117	272	588	1667	5522
T(scoring)	6	16	45	120	345	989	2901	8563	26493	75446
T_{12}(learning)	2	4	9	19	42	73	180	339	897	1572
T_{12}(scoring)	2	4	9	21	51	128	329	885	2432	6861
T_{24}(learning)	2	4	11	21	44	77	192	364	949	1564
T_{24}(scoring)	2	4	10	23	55	124	293	694	1724	4418

Table 11: Comparison of runtime (in seconds) used by learning phase and scoring phase in New Para-OS's runtime experiments.

would use over 5 GB (maximum) memory per processor and a 35-variable network would need over 46 GB per processor.

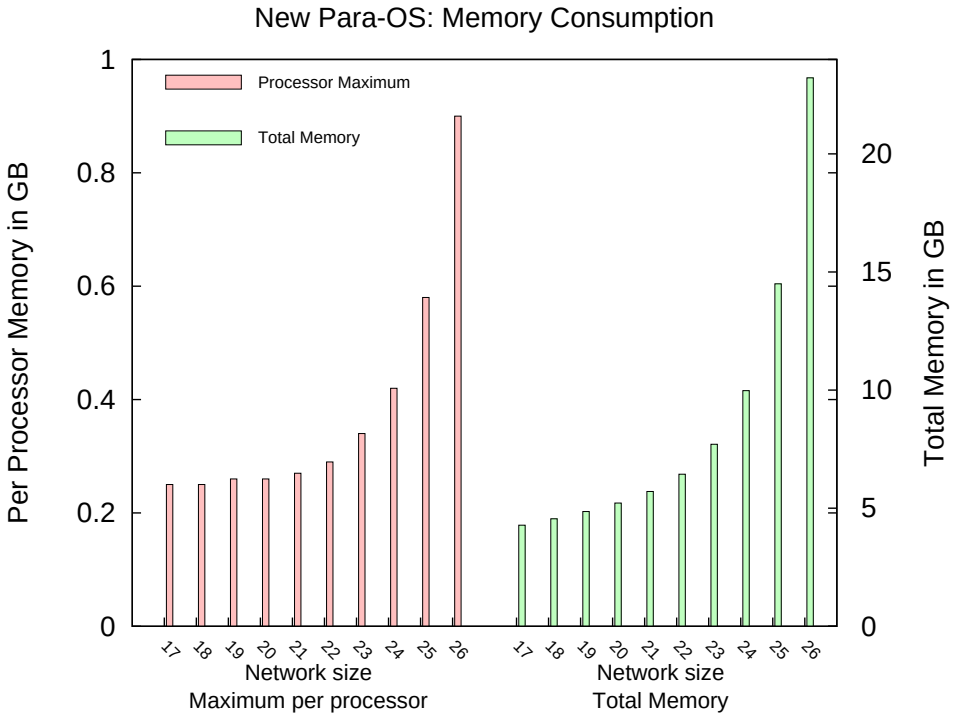


Figure 20

6.2.3 Various Datasets

We tested both Para-OS and New Para-OS for runtime on 5 various datasets of 25 variables and 100 datapoints. The datasets were generated by extracting 25 columns and 100 rows from original datasets, and single-valued columns were allowed. All of the datasets are binary. The results are displayed in Table 12. We can see that choosing different datasets effects directly on runtime, because the complexity of the datasets decides how much time used on building AD-tree, and calculating BDeu local scores, etc. Para-OS and the new version are equally fast for processing these datasets, but as we presented above, the latter is much more memory efficient.

One of the proposed advantages of Para-OS is that it does not use an upper bound on the size of parent sets. Neither does New Para-OS set this upper bound. In Table 13 we lists the largest number of parent set size.

We can further optimize the implementation to reduce memory usage, for example, by modifying hash-table to be more compact or by choosing more efficient data-structure without damaging speed. However at the moment, it seems for New Para-OS that speed is a more limiting factor than memory.

dataset	Alarm	Autos	Bands	Flag	Soybeans
Para-OS(256)	2502	5355	1039	10598	8665
Para-OS(512)	1285	2627	547	5460	4188
New Para-OS(12-thread)	2706	5185	1676	10144	6778
New Para-OS(24-thread)	2788	4542	2083	7637	5726

Table 12: Runtimes (in seconds) of Para-OS ($n_p = 256, 512$) and New Para-OS (12 and 24 threads) on various datasets of 25 variables and 100 datapoints.

dataset	Alarm	Autos	Bands	Flag	Soybeans
largest $\|PA_{X_i}\|$	10	9	13	13	8

Table 13: This table shows the largest size of parent sets PA_{X_i} in the learned networks.

Extrapolating from the experiments, a network of 30 variables using 12 threads per processor would take about 4 and a half days, and using 24 thread would need 3 and a half days. The time limit for one parallel batch job in Taito is 72 hours (3 days).

Comparing New Para-OS with our Para-OS implementation, for the 25-variable dataset Alarm Para-OS took 3367 seconds (about 0.9 hours) by running on 256 processors, and the maximum memory consumption per processor is about 4 GB. For the same dataset, New Para-OS took 28163 seconds (about 7.8 hours) by using 25 processors with no-threading, and 3331 seconds (about 0.9 hour) on 25 processors with 12-threads per physical processor. Maximum memory consumption per processor is 0.58 GB for New Para-OS. We can see that runtime for both programs were similar, but memory consumption dropped dramatically in the new program.

Comparing New Para-OS with the Para-OS in [TIM11] is not straightforward. However, in [TIM11] their Para-OS used total memory 28.50 GB with $n_p = 256$ to run a 25-variable dataset (their own artificially generated dataset), and here with New Para-OS we used total memory 14.50 GB for our 25-variable dataset. For 26-variable dataset they used 40.25 GB total memory and our number is 23.22 GB. Even though the datasets and n_p are different, we can still say that using SPG and limiting the communication (on sending local scores) are an effective way to save memory. For runtime comparison, their Para-OS in [TIM11] took 5386 seconds (about 1 and a half hours) with $n_p = 256$ for their 25-variable dataset, and our New Para-OS with 12-threads took 3331 seconds (about 0.9 hours) with $n_p = 25 * 12 = 300$ for our 25-variable dataset. We can see that New Para-OS has a better wall-clock time. It performs better as it is able to take advantage of multi-threading.

7 Summary and Discussion

In this thesis, we implemented a parallelized algorithm Para-OS, ran experiments with it and compared the results with those presented in the original paper [TIM11]. We managed to repeat most of the experiments recorded in [TIM11]. We conclude that, while Para-OS does work, it is not very efficient. In our opinion, some drawbacks of Para-OS are: redundant calculation, lack of score pruning (which also increases memory usage, for storing all of the local scores), interleaved score calculation and learning (thus score pruning is not easily applicable without introducing more data communication or changing the algorithm), and too much communication between processors.

Our implementation of Para-OS is based on the original Para-OS description [TIM11]; however, based on their results, we opted for different memory-communication tradeoffs. Thus, the experimental results are not consistent. Some factors leading to these differences are, for example, data-structure used to store the intermediate results, communication strategies, etc.

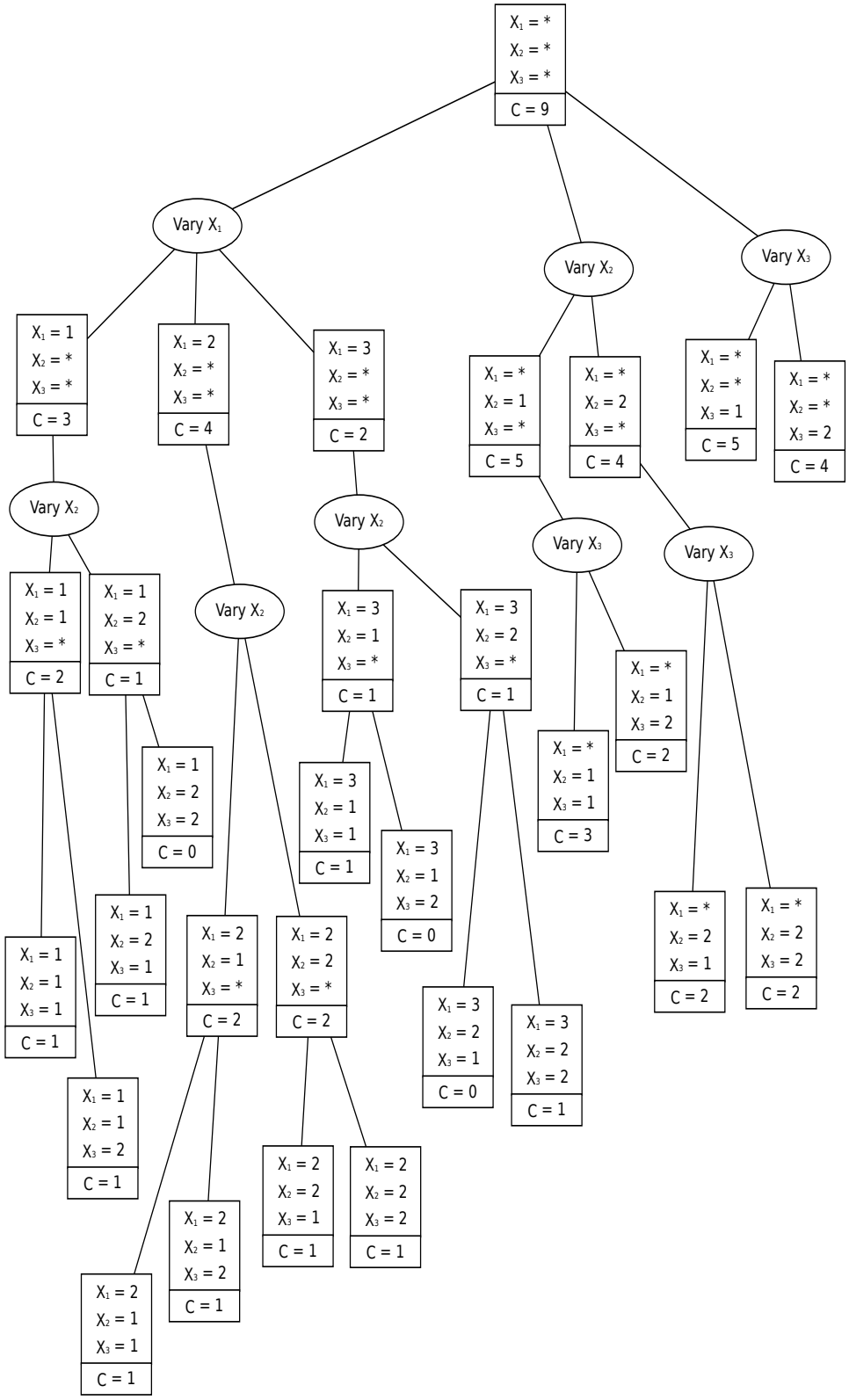
We also proposed and implemented a new version of Para-OS which separates the scoring phase totally from the learning phase, utilizes sparse parent graphs which can apply score pruning, and limits the communication between processors to be information about subnetworks. Experimental results show that the new version saves memory compared to our implementation of Para-OS and the results reported in [TIM11], and with threading it also provides a good runtime.

There are still several opportunities for improvement in New Para-OS. (i) We can introduce further parallelism. There is one SPG per variable. Currently the calculation of SPGs is parallelized by using MPI, in a way that one processor is responsible for the SPG of one variable. After that we use OpenMP and threading to further parallelize the task on each processor. However the number of threads available for one physical processor is limited by the number of cores the processor has. To gain further parallelism, we can split the calculation of one SPG into several processors by using MPI. The key is to split the space of possible parent sets. For example, we can use 3 processors (each has 12 threads) to calculate all the scores needed for one SPG, and a partial SPG is built on each physical processor and merged into one at the end. (ii) Another improvement is to implement further score pruning. For building SPG we need to calculate the scores for all possible parent sets, and some of them are later pruned away. De Campos [DCJ11] introduced a method to enable earlier pruning – to skip the calculation of some scores [DCJ11]. He proposed a criterion for investigating a possible parent set. If the criterion is met, then the calculation of the score can be safely abandoned, as it is ensured that this parent set can not be optimal. (iii) Techniques could also be used to speed up the learning phase, for example approaches for parallelizing breadth-first search ([BM11], [LS10]).

The drawback of New Para-OS is that it has to be run on a supercomputer to enable fast communication between processors. New Para-OS is a small experiment on improving Para-OS, and we did not manage to increase the solvable network size. Various algorithms and clever methods have been proposed in recent years, but BN structure learning remains an NP-hard problem. For large network size one still needs to resort to approximate methods. Exact algorithms are improving, for example by utilizing tradeoffs between space and time, parallelization, score pruning, etc. It is an area under intensive research and will bring great benefits on AI and many other fields once breakthroughs are made.

8 Appendices

8.1 Appendix 1: an example of full AD-tree


X1 = *
X2 = *
X3 = *
C = 9
Vary X1
Vary X2
Vary X3
X1 = 1
X2 = *
X3 = *
C = 3
X1 = 2
X2 = *
X3 = *
C = 4
X1 = 3
X2 = *
X3 = *
C = 2
X1 = *
X2 = 1
X3 = *
C = 5
X1 = *
X2 = 2
X3 = *
C = 4
X1 = *
X2 = *
X3 = 1
C = 5
X1 = *
X2 = *
X3 = 2
C = 4
Vary X2
Vary X2
Vary X3
Vary X3
X1 = 1
X2 = 1
X3 = *
C = 2
X1 = 1
X2 = 2
X3 = *
C = 1
Vary X2
X1 = 3
X2 = 1
X3 = *
C = 1
X1 = 3
X2 = 2
X3 = *
C = 1
X1 = *
X2 = 1
X3 = 2
C = 2
X1 = 1
X2 = 2
X3 = 2
C = 0
X1 = *
X2 = 1
X3 = 1
C = 3
X1 = 3
X2 = 1
X3 = 1
C = 1
X1 = 3
X2 = 1
X3 = 2
C = 0
X1 = 1
X2 = 1
X3 = 1
C = 1
X1 = 1
X2 = 2
X3 = 1
C = 1
X1 = 2
X2 = 1
X3 = *
C = 2
X1 = 2
X2 = 2
X3 = *
C = 2
X1 = 3
X2 = 2
X3 = 1
C = 0
X1 = 3
X2 = 2
X3 = 2
C = 1
X1 = *
X2 = 2
X3 = 1
C = 2
X1 = *
X2 = 2
X3 = 2
C = 2
X1 = 1
X2 = 1
X3 = 2
C = 1
X1 = 2
X2 = 2
X3 = 1
C = 1
X1 = 2
X2 = 2
X3 = 2
C = 1
X1 = 2
X2 = 1
X3 = 2
C = 1
X1 = 2
X2 = 1
X3 = 1
C = 1

References

[Aka73] Akaike, Hirotugu: *Information theory and an extension of the maximum likelihood principle.* In *Second international symposium on information theory*, pages 267–281. Akademinai Kiado, 1973.

[Bal11] Balding, David J: *Bayesian Networks and probabilistic inference in forensic science.* Law, Probability and Risk, 10(4):355–358, 2011.

[BM11] Buluç, Aydin and Madduri, Kamesh: *Parallel breadth-first search on distributed memory systems.* In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.

[Bun91] Buntine, Wray: *Theory refinement on Bayesian Networks.* pages 52–60. Morgan Kaufmann, 1991.

[CH92] Cooper, Gregory F and Herskovits, Edward: *A Bayesian method for the induction of probabilistic Networks from data.* Machine learning, 9(4):309–347, 1992.

[Chi96] Chickering, David Maxwell: *Learning Bayesian Networks is NP-complete.* In *Learning from data*, pages 121–130. Springer, 1996.

[Cus11] Cussens, James: *Bayesian Network learning with cutting planes.* In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011)*, pages 153–160, 2011.

[DC06] De Campos, Luis M: *A scoring function for learning Bayesian Networks based on mutual information and conditional independence tests.* The Journal of Machine Learning Research, 7:2149–2187, 2006.

[DCFLGP02] De Campos, Luis M, Fernandez-Luna, Juan M, Gámez, José A, and Puerta, José M: *Ant colony optimization for learning Bayesian Networks.* International Journal of Approximate Reasoning, 31(3):291–311, 2002.

[DCJ11] De Campos, Cassio P and Ji, Qiang: *Efficient structure learning of Bayesian Networks using constraints.* The Journal of Machine Learning Research, 12:663–689, 2011.

[DCP01] De Campos, Luis M and Puerta, J Miguel: *Stochastic local algorithms for learning belief Networks: Searching in the space*

of the orderings. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 228–239. Springer, 2001.

[DCZJ09] De Campos, Cassio P, Zeng, Zhi, and Ji, Qiang: *Structure learning of Bayesian Networks using constraints.* In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 113–120. ACM, 2009.

[FDF95] Fung, Robert and Del Favero, Brendan: *Applying Bayesian Networks to information retrieval.* Communications of the ACM, 38(3), 1995.

[FLNP00] Friedman, Nir, Linial, Michal, Nachman, Iftach, and Pe'er, Dana: *Using Bayesian Networks to analyze expression data.* Journal of computational biology, 7(3-4):601–620, 2000.

[FN12] Fenton, Norman and Neil, Martin: *Risk Assessment and Decision Analysis with Bayesian Networks.* CRC Press, 2012.

[Glo90] Glover, Fred: *Tabu search: A tutorial.* Interfaces, 20(4):74–94, 1990.

[HASL05] Horvitz, Eric, Apacible, Johnson, Sarin, Raman, and Liao, Lin: *Prediction, expectation, and surprise: Methods, designs, and study of a deployed traffic forecasting service.* In *UAI'05*, pages 275–283, 2005.

[Hec98] Heckerman, David: *A tutorial on learning with Bayesian Networks.* Springer, 1998.

[HGC95] Heckerman, David, Geiger, Dan, and Chickering, David M: *Learning Bayesian Networks: The combination of knowledge and statistical data.* Machine learning, 20(3):197–243, 1995.

[HGPS02] Hsu, William H, Guo, Haipeng, Perry, Benjamin B, and Stilson, Julie A: *A permutation genetic algorithm for variable ordering in learning Bayesian Networks from data.* In *GECCO*, volume 2, pages 383–390, 2002.

[Jor98] Jordan, Michael Irwin: *Learning in graphical models.* Adaptative computation and machine learning series. London, 1998.

[JSGM10] Jaakkola, Tommi, Sontag, David, Globerson, Amir, and Meila, Marina: *Learning Bayesian Network structure using lp relaxations.* In *International Conference on Artificial Intelligence and Statistics*, pages 358–365, 2010.

[KF09] Koller, Daphne and Friedman, Nir: *Probabilistic Graphical Models: Principles and Techniques.* MIT Press, 2009.

[KJRSH97] Kahn Jr, Charles E, Roberts, Linda M, Shaffer, Katherine A, and Haddawy, Peter: *Construction of a Bayesian Network for mammographic diagnosis of breast cancer.* Computers in biology and medicine, 27(1):19–29, 1997.

[KPIM10] Kojima, Kaname, Perrier, Eric, Imoto, Seiya, and Miyano, Satoru: *Optimal search on clustered structural constraint for learning Bayesian Network structure.* The Journal of Machine Learning Research, 11:285–310, 2010.

[Kra09] Kragt, Marit E: *A beginners guide to Bayesian Network modelling for integrated catchment management.* Landscape Logic, 2009.

[KS04] Koivisto, Mikko and Sood, Kismat: *Exact Bayesian structure discovery in Bayesian Networks.* Journal of Machine Learning Research, 5:549–573, 2004.

[LB94] Lam, Wai and Bacchus, Fahiem: *Learning Bayesian belief networks: An approach based on the mdl principle.* Computational intelligence, 10(3):269–293, 1994.

[LKMY96] Larranaga, Pedro, Kuijpers, Cindy MH, Murga, Roberto H, and Yurramendi, Yosu: *Learning Bayesian Network structures by searching for the best ordering with genetic algorithms.* Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, 26(4):487–493, 1996.

[LMY12] Liu, Zhifa, Malone, Brandon, and Yuan, Changhe: *Empirical evaluation of scoring functions for Bayesian Network model selection.* BMC bioinformatics, 13(Suppl 15):S14, 2012.

[LS10] Leiserson, Charles E and Schardl, Tao B: *A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers).* In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 303–314. ACM, 2010.

[LSS05] Luo, Jiebo, Savakis, Andreas E, and Singhal, Amit: *A Bayesian Network-based framework for semantic image understanding.* Pattern Recognition, 38(6):919–934, 2005.

[MAPV96] Madigan, David, Andersson, Steen A, Perlman, Michael D, and Volinsky, Chris T: *Bayesian model averaging and model selection for markov equivalence classes of acyclic digraphs.* Communications in Statistics–Theory and Methods, 25(11):2493–2519, 1996.

[ML98] Moore, Andrew and Lee, Mary Soon: *Cached sufficient statistics for efficient machine learning with large datasets.* J. Artif. Int. Res., 8(1):67–91, March 1998, ISSN 1076-9757.

[MLL99] Myers, James W, Laskey, Kathryn Blackmond, and Levitt, Tod: *Learning Bayesian Networks from incomplete data with stochastic search algorithms.* In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 476–485. Morgan Kaufmann Publishers Inc., 1999.

[MY12] Malone, Brandon and Yuan, Changhe: *A parallel, anytime, bounded error algorithm for exact Bayesian Network structure learning.* In *Proceedings of the Sixth European Workshop on Probabilistic Graphical Models (PGM-12)*, 2012.

[N+04] Neapolitan, Richard E *et al.*: *Learning Bayesian Networks.* Prentice Hall Series in Artificial Intelligence. Pearson Prentice Hall, 2004, ISBN 9780130125347.

[Nef02] Nefian, Ara V: *Embedded Bayesian Networks for face recognition.* In *Multimedia and Expo, 2002. ICME'02. Proceedings. 2002 IEEE International Conference on*, volume 2, pages 133–136. IEEE, 2002.

[OIM04] Ott, Sascha, Imoto, Seiya, and Miyano, Satoru: *Finding optimal models for small gene networks.* In *Pacific Symposium on Biocomputing*, pages 557–567, 2004.

[OM03] Ott, Sascha and Miyano, Satoru: *Finding optimal gene networks using biological constraints.* GENOME INFORMATICS SERIES, pages 124–133, 2003.

[Pea88] Pearl, Judea: *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann, 1988.

[PIM08] Perrier, Eric, Imoto, Seiya, and Miyano, Satoru: *Finding optimal Bayesian Network given a super-structure.* Journal of Machine Learning Research, 9(2):2251–2286, 2008.

[PK13] Parviainen, Pekka and Koivisto, Mikko: *Finding optimal Bayesian Networks using precedence constraints.* Journal of Machine Learning Research, 14:1387–1415, 2013.

[Sch78] Schwarz, Gideon: *Estimating the dimension of a model.* The annals of statistics, 6(2):461–464, 1978.

[SKM07] Silander, Tomi, Kontkanen, Petri, and Myllymaki, Petri: *On sensitivity of the MAP Bayesian Network structure to the*

equivalent sample size parameter. In *UAI'07*, pages 360–367, 2007.

[SM05] Singh, Ajit P and Moore, Andrew W: *Finding Optimal Bayesian Networks by Dynamic Programming*. Technical report, Carnegie Mellon University, School of Computer Science, 2005.

[SM06] Silander, Tomi and Myllymäki, Petri: *A simple approach for finding the globally optimal Bayesian Network structure*. In *Proceedings of the 22th Conference on Uncertainty in Artificial Intelligence*, pages 445–452, 2006.

[SRKM08] Silander, Tomi, Roos, Teemu, Kontkanen, Petri, and Myllymäki, Petri: *Factorized normalized maximum likelihood criterion for learning Bayesian Network structures*. 2008.

[TBA06] Tsamardinos, Ioannis, Brown, Laura E, and Aliferis, Constantin F: *The max-min hill-climbing Bayesian Network structure learning algorithm*. Machine learning, 65(1):31–78, 2006.

[Tey05] Teyssier, Marc: *Ordering-based search: A simple and effective algorithm for learning Bayesian Networks*. In *UAI'05*, 2005.

[TIM11] Tamada, Yoshinori, Imoto, Seiya, and Miyano, Satoru: *Parallel algorithm for learning optimal Bayesian Network structure*. Journal of Machine Learning Research, 12:2437–2459, 2011.

[Uus07] Uusitalo, Laura: *Advantages and challenges of Bayesian Networks in environmental modelling*. Ecological modelling, 203(3):312–318, 2007.

[YC02] Yang, Shulin and Chang, Kuo Chu: *Comparison of score metrics for Bayesian Network learning*. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, 32(3):419–428, 2002.

[YM13] Yuan, Changhe and Malone, Brandon: *Learning optimal Bayesian Networks: A shortest path perspective*. Journal of Artificial Intelligence Research, 48:23–65, 2013.

[ZR98] Zweig, Geoffrey and Russell, Stuart: *Speech recognition with dynamic Bayesian Networks*. 1998.