Parallel and Distributed Bayesian Network Structure Learning

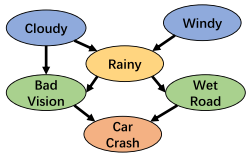
Jian Yang, Jiantong Jiang, Zeyi Wen, and Ajmal Mian

***Abstract*—Bayesian networks (BNs) are graphical models representing uncertainty in causal discovery, and have been widely used in medical diagnosis and gene analysis due to their effectiveness and good interpretability. However, mainstream BN structure learning methods are computationally expensive, as they must perform numerous conditional independence (CI) tests to decide the existence of edges. Some researchers attempt to accelerate the learning process by parallelism, but face issues including load unbalancing, costly dominant parallelism overhead. We propose a multi-thread method, namely Fast-BNS version 1 (Fast-BNS-v1 for short), on multi-core CPUs to enhance the efficiency of the BN structure learning. Fast-BNS-v1 incorporates a series of efficiency optimizations, including a dynamic work pool for better scheduling, grouping CI tests to avoid unnecessary operations, a cache-friendly data storage to improve memory efficiency, and on-the-fly conditioning sets generation to avoid extra memory consumption. To further boost learning performance, we develop a two-level parallel method Fast-BNS-v2 by integrating edge-level parallelism with multi-processes and CI-level parallelism with multi-threads. Fast-BNS-v2 is equipped with careful optimizations including dynamic work stealing for load balancing, SIMD edge list deletion for list updating, and effective communication policies for synchronization. Comprehensive experiments show that our Fast-BNS achieves 9 to 235 times speedup over the state-of-the-art multi-threaded method on a single machine. When running on multi-machines, it further reduces the execution time of the single-machine implementation by 80%.**

***Index Terms*—Bayesian networks, distributed machine learning systems, parallelism.**

I. Introduction

BAYESIAN networks (BNs) [1] are probabilistic graphical models that employ directed acyclic graphs (DAGs) to compactly represent a set of random variables and their conditional dependencies. The graphical nature of BNs makes them well-suited for representing knowledge with uncertainty and effective reasoning. With the recent growing demand for interpretable machine learning models, BNs have attracted much research attention since they are inherently interpretable models [2], [3].



(a)

Cloudy	Windy	Rainy	Bad Vision	Wet Road	Car Crash
Yes	No	Yes	Yes	Yes	Yes
No	Yes	Yes	No	Yes	No
Yes	Yes	Yes	Yes	Yes	Yes

(b)

Fig. 1. Example of a BN and its observed dataset: (a) the underlying BN structure and (b) the observed dataset.

One crucial task of training BNs is structure learning, which aims to learn DAGs that are well-matched to the observed data. Fig. 1 illustrates an example of a BN about traffic prediction with the dataset it is based on. Each column in the dataset represents a variable and opposites a node in the BN, while each row is a sample that records the observed values of these variables. Intuitively, given that in most cases, if *Cloudy* or *Windy* is **yes**, *Rainy* is also **yes**, we infer that *Cloudy* and *Windy* are highly likely reasons for *Rainy*. Therefore, during BN structure learning, we tend to discover two directed edges connecting "*Cloudy*" to "*Rainy*" and "*Windy*" to "*Rainy*". A similar idea can be used to conjecture edges between other variables.

However, the BN structure learned by this intuitive idea is too rough to be applied in real-world applications. Two common approaches for more precise BN structure learning include score-based approaches and constraint-based approaches. The score-based approaches use a scoring function to measure the fitness of DAGs to the data and find the highest score out of all the possible DAGs, which makes the number of possible DAGs super-exponential to the number of dimensions (i.e., variables) of the learning problems [4]. On the other hand, the constraint-based approaches perform many conditional independence (CI) tests to identify the conditional independence relations among the random variables and use these relations as constraints to construct BNs. This category of methods often runs in a polynomial time and has been commonly used in real-world applications [5].

A fundamental constraint-based algorithm is the PC (named after its authors Peter and Clark) algorithm [6] which starts from a complete undirected graph and removes edges in consecutive depths based on CI tests. PC-stable [7] solves the order-dependent issue in the original PC algorithm and reduces

Manuscript received 6 February 2023; revised 15 October 2023; accepted 19 October 2023. Date of publication 23 October 2023; date of current version 15 February 2024. This work was supported by the Guangzhou Municipal Science and Technology Project under Grant 2023A03J0143. The work of Ajmal Mian was supported in part by Australian Research Council Future Fellowship Award under Grant FT210100268 funded by the Australian Government. Recommended for acceptance by Y. Zhang. *(Corresponding author: Zeyi Wen.)*

Jian Yang and Zeyi Wen are with the Hong Kong University of Science and Technology (Guangzhou), (HKUST-GZ), Nansha, Guangzhou, Guangdong 511400, China, and also with the Hong Kong University of Science and Technology (HKUST), Clear Water Bay, Kowloon, Hong Kong (e-mail: jyang827@connect.hkust-gz.edu.cn; wenzeyi@ust.hk).

Jiantong Jiang and Ajmal Mian are with The University of Western Australia, Crawley, WA 6009, Australia (e-mail: jiantong.jiang@research.uwa.edu.au; ajmal.mian@uwa.edu.au).

Digital Object Identifier 10.1109/TPDS.2023.3326832

1045-9219 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

error than the original PC algorithm. The PC-stable algorithm has been widely used in various applications [5], [8] and is implemented in different mainstream BN packages such as bnlearn [9], pcalg [10] and tetrad [11]. Furthermore, most constraint-based methods are improved versions of the PC-stable algorithm or proceed along similar lines to the PC-stable algorithm.

However, the PC-stable algorithm suffers from long execution time when performing a large number of CI tests, especially for high-dimensional problems. Since algorithmic improvements for PC-stable are non-trivial [12], several research efforts have been devoted to the acceleration of the PC-stable algorithm on multi-core CPUs [13], [14], [15]. The most common way is to parallelize the processing of different network edges inside each depth, which is intuitive due to the order-independent property of the PC-stable algorithm. However, direct edge-level parallelism is load unbalanced because the workloads of CI tests for different edges are highly different. To overcome these problems, we propose Fast-BNS version 1 (Fast-BNS-v1 for short), a CI-level multi-thread parallel PC-stable implementation equipped with a dynamic work pool to contain the edges to be processed and their processing progresses about the CI tests. Fast-BNS-v1 exploits CI tests batch processing, cache-friendly storage, and on-the-fly data generation to accelerate PC-stable and save memory consumption. The related content has been published in a conference paper [16].

To further enhance the efficiency of BN structure learning, we extend Fast-BNS-v1 to multi-process and distributed settings, which results in Fast-BNS-v2. Since each process has independent memory to avoid racing conditions, multi-process parallelism tends to achieve higher CPU utilization [17], and meanwhile reduces time consumption by unleashing the computing power of multiple machines. Moreover, Fast-BNS-v2 exploits two-level parallelism which combines benefits from multi-thread and multi-process parallelism to enhance device utilization and boost algorithm performance. At different levels, we apply different parallel policies and granularity. More specifically, to reduce communication overhead and balance workload, higher levels of Fast-BNS-v2 adopt coarser subtask separation and lower levels use finer subtask separation. In Fast-BNS-v2, CI-level parallelism is achieved in the thread level, and edge-level parallelism is performed in the process level. It is worth pointing out that CI-level workload balancing is implemented by a dynamic work pool similar to Fast-BNS-v1, while edge-level workload scheduling exploits dynamic work stealing [18].

To summarize, this paper is an extension to our conference paper [16] with the following additional contributions.

- We extend our previous algorithm Fast-BNS-v1 and propose Fast-BNS-v2 to support multi-machines. Fast-BNS-v2 exploits two-level parallelism and leads to better resource utilization in multi-core and multi-machines, due to the independent memory space among processes. In Fast-BNS-v2, edge-level parallelism is implanted in multi-process parallelism while CI-level is applied in multi-thread parallelism.
- To further improve the efficiency of Fast-BNS-v2, we develop a series of novel techniques including load balancing by dynamic work stealing to boost CPU utilization, SIMD edge list deletion to maintain the active edges, and brief communication protocol to decrease data transfer cost.
- We conduct experiments to study the effectiveness of our proposed methods. Experimental results on a single machine show that the sequential versions of Fast-BNS-v1 and Fast-BNS-v2 outperform the existing work bnlearn [9] and tetrad [11] by up to 50 times. Compared with the multi-threaded implementation in bnlearn [13], Fast-BNS-v1 is 9 to 24 times faster, while Fast-BNS-v2 is 9 to 235 times faster. Finally, we show that Fast-BNS-v2 has good scalability to distributed environments, which further reduces the 80% execution time of the single-machine implementation.

II. Preliminaries

In this section, we provide the key terminologies and definitions related to Bayesian network structure learning, and then review the PC-stable algorithm.

A. Bayesian Networks

Bayesian networks (BNs) are a class of graphical models that represent a joint distribution P over a set of random variables $V = \{V_0, V_1, \ldots, V_{n-1}\}$ via a DAG. Typically, one variable corresponds to one feature in machine learning problems. We use $G = (V, \mathcal{E})$ to denote the DAG. Fig. 1(a) shows an example DAG denoted by G, where each node in V is associated with one variable and each edge in $\mathcal{E}$ represents conditional dependencies among the two variables. V_j is called a parent of V_i if there exists a directed edge from V_j to V_i in G, and we use $Pa(V_i)$ to denote the set of parent variables of V_i.

In a BN, each variable has its local probability distribution that describes the probabilities of possible values of this variable given its possible parent configurations. The joint probability of variables V in a BN can be decomposed into the product of the local probability distributions of each variable, and each local probability distribution depends only on a single variable V_i and its parents:

$$P(V_0, V_1, \ldots, V_{n-1}) = \prod_{i=0}^{n-1} P(V_i | Pa(V_i)),$$

where n is the number of variables, $P(V_0, V_1, \ldots, V_{n-1})$ is the joint probability and $P(V_i|Pa(V_i))$ is the conditional probability of variable V_i.

B. Conditional Independence Tests

Consider some random variables V_i, V_j and V_k in a BN, a CI test assertion of the form $I(V_i, V_j|\{V_k\})$ means V_i and V_j are independent given V_k. Let $\mathcal{D} = \{c_0, c_1, \ldots, c_{m-1}\}$ denote a Dataset of m complete samples, a CI test $I(V_i, V_j|\{V_k\})$ determines whether the corresponding hypothesis $I(V_i, V_j|\{V_k\})$ holds or not, based on statistics of D. For discrete variables, the most common statistic for testing $I(V_i, V_j|\{V_k\})$ is the G^2 test

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

statistic [6] defined as

$$G^2 = 2\sum_{x,y,z} N_{xyz} log \frac{N_{xyz}}{E_{xyz}},$$

where N_{xyz} is the number of samples in D that satisfies $V_i = x$, $V_j = y$ and $V_k = z$. The value of N_{xyz} can be obtained from the contingency table that shows the frequencies for all configurations of values. G^2 follows an asymptotic χ^2 distribution with $(|V_i| - 1)(|V_j| - 1)$, where $|\cdot|$ denotes the number of possible values of the variable. The p value of χ^2 distribution can be calculated according to the G^2 statistic and the final decision is made by comparing p value with the significance level α. If p value is greater than α, the independent hypothesis $I(V_i, V_j|\{V_k\})$ is accepted; otherwise, the hypothesis is rejected. E_{xyz} is the expected frequency which is defined as

$$E_{xyz} = \frac{N_{x+z}N_{+yz}}{N_{++z}},$$

where $N_{x+z} = \sum_y N_{xyz}$, $N_{+yz} = \sum_x N_{xyz}$, and $N_{++z} = \sum_{xy} N_{xyz}$, which represent the marginal frequencies.

C. The PC-Stable Algorithm

The PC-stable algorithm is a constraint-based method for BN structure learning from data. PC-stable consists of three steps. The first step is to determine the skeleton of the graph. The term skeleton means the underlying undirected graph of the learned network. This step is done by performing a large number of CI tests. The second step is to identify the v-structures in the skeleton. A v-structure is a triple (V_i, V_j, V_k) that can be denoted by $V_i \rightarrow V_k \leftarrow V_j$. In other words, nodes V_i and V_j have an outgoing edge to node V_k and are not connected by any edge in the graph. V-structure is a key component to distinguish different network structures. By identifying the v-structures in this step, some edges in the skeleton become directed edges. The third step is to set directions for as many of the remaining undirected edges as possible by applying a set of rules called Meek rules [19]. For example, we set the direction of the undirected edge $V_j - V_k$ into $V_j \rightarrow V_k$ whenever there is a directed edge $V_i \rightarrow V_j$ such that V_i and V_k are not adjacent; otherwise a new v-structure is created. In the three steps of the PC-stable algorithm, the first step is much more time-consuming [20], taking over 90% of the total execution time in many problems. In Section III, we elaborate the details of our proposed techniques to accelerating the first step.

The pseudo-code of the first step of the PC-stable algorithm is given in Algorithm 1. The general idea is to initialize G to a complete undirected graph over the node set V (Line 3), and remove some of the edges by performing a number of CI tests in consecutive depths (Lines 6 to 16). Specifically, at each depth d, the algorithm iteratively records the current adjacency sets of all the nodes, where $adj(G, V_i)$ denotes the adjacent nodes of V_i in G(Lines 5). This operation is used for choosing the conditioning set $\mathcal{S}$ later. Next, for every edge (V_i, V_j) in the graph G, a number of CI tests $I(V_i, V_j|\mathcal{S})$ are performed for different conditioning sets. The elements in the conditioning sets are chosen from $a(V_i)\backslash\{V_j\}$, and the size of each conditioning set $|\mathcal{S}|$ is equal to the current depth d (Lines 7 to 10). If there exists a conditioning set $\mathcal{S}$ where V_i is independent of V_j given $\mathcal{S}$, the edge (V_i, V_j) is removed from G, and $\mathcal{S}$ is stored in $SepSet(V_i, V_j)$ (Lines 11 to 13). $SepSet(V_i, V_j)$ denotes the separating set of V_i and V_j, which is used in the second step of the PC-stable algorithm to identify the v-structures. Since the second step is fast and is not the focus of our work, we omit the details of separating set. Once all edges are considered, d is incremented (Line 15) and the above procedure is repeated for the next depth. Depth d is used to control the size of the conditioning sets from small to large. This process continues until all pairs of adjacent nodes (V_i, V_j) in G satisfy $|a(V_i)\backslash\{V_j\}| < d$ as shown in Line 16.

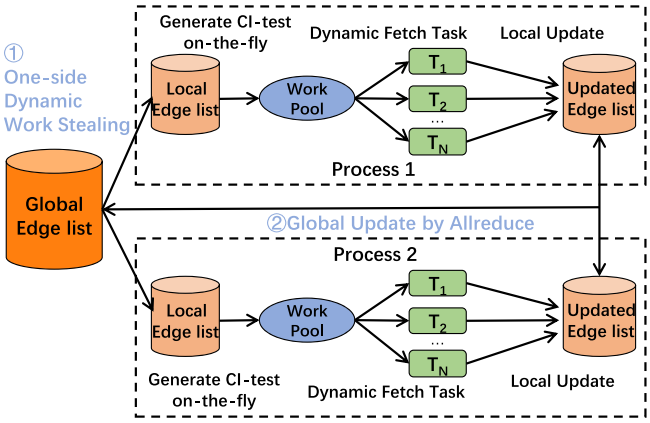


Fig. 2. The framework of Fast-BNS-v2 in 2 processes, including two large progresses: one-side dynamic work fetching and global update.

Algorithm 1: The 1st Step of PC-Stable Algorithm.

1 **Input:** Node set V
2 **Output:** Graph G, $SepSet$
3 Graph $G \leftarrow$ FormCompletedGraph$(V) = V \times V$
4 Depth $d \leftarrow 0$
5 Let $a(V_i)$ represent adjacency nodes of V_i
6 **repeat**
7 **for** any edge (V_i, V_j) in G **do**
8 **repeat**
9 Choose a new $\mathcal{S} \subseteq a(V_i)\backslash\{V_j\}$ with $|\mathcal{S}| = d$
10 Perform CI test $I(V_i, V_j|\mathcal{S})$
11 **if** hypothesis $I(V_i, V_j|\mathcal{S})$ holds **then**
12 Remove (V_i, V_j) from G
13 Store $\mathcal{S}$ in $SepSet(V_i, V_j)$
14 **until** (V_i, V_j) is removed or all $\mathcal{S}$ are considered
15 $d \leftarrow d + 1$
16 **until** all pairs of (V_i, V_j) in G satisfy $|a(V_i)\backslash\{V_j\}| < d$

III. Our Proposed Fast-BNS

This section presents the technical details of our proposed Fast-BNS, a parallel method for BN structure learning. The details of both versions of our proposed method (i.e., Fast-BNS-v1 and Fast-BNS-v2) are elaborated in the following.

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

A. Design Overview

Here, we provide the overview of our proposed Fast-BNS, an efficient parallel and distributed method for learning BN structure in parallel and distributed environment. Fig. 2 depicts the design of Fast-BNS at each depth d on two distributed processes. In the distributed multi-process level, a global edge list is maintained in the root process and the edges inside it are distributed to all the processes via a dynamic work-stealing strategy (cf. Section III-C). In the parallel multi-thread level, a dynamic work pool is maintained in each process to monitor the processing progresses of the distributed edges with regard to the CI tests (cf. Section III-B). The CI tests are then dynamically fetched and solved by the parallel threads. According to the results of the CI tests, the local edge list of each process is updated and merged to the global edge list.

Traditional parallelism can be divided into two types: data-level parallelism and task-level parallelism. Data-level parallel programming allocates different data blocks to workers executing the same operations, while task-level parallel programming mapping subtasks with different instructions to workers. In the PC-stable algorithm, each step depends on results from its previous step, which is unsuitable for conducting task-level parallelism. Most BN structure learning parallel methods utilize data-level parallelism to accelerate algorithm execution because the data is divisible.

From the implementation perspective, parallel techniques contain multi-thread and multi-process parallelism. In this work, we first develop Fast-BNS-v1 with multi-thread parallelism by evaluating many CI tests simultaneously. Although multi-thread parallelism is simple to implement, it only utilizes one process, which limits the total resources held by the learning process and limits the parallelism among threads. On the other hand, multi-process parallelism avoids race condition and increases overall computation ability, because each process occupies its own memory space and computing resources. To improve the efficiency of Fast-BNS-v1, we propose Fast-BNS-v2, which employs multi-processes and exploits edge-level parallelism with load balancing policies.

Moreover, Fast-BNS is equipped with a series of novel optimizations including i) grouping the CI tests to reduce unnecessary CI tests, ii) employing a cache-friendly data storage to improve the memory efficiency, iii) generating the conditioning sets of the CI tests on-the-fly and in parallel to avoid extra memory consumption, and iv) utilizing SIMD edge list deletion to reduce computational complexity. To better understand Fast-BNS, we provide theoretical analysis of its performance. In the following, we elaborate the technical details of Fast-BNS and derive theoretical results.

B. Multi-Thread Parallelism

In this subsection, we introduce the proposed method: Fast-BNS-v1, employing multi-thread parallelism. In each depth, threads perform CI tests from different edges in parallel. To achieve multi-thread parallelism, our key idea is to employ a dynamic work pool in shared memory for each depth implemented by a stack. The work pool contains the edges required to be processed and their processing progresses with respect to the CI tests. Therefore, each time a thread can fetch CI tests belonging to multiple edges from the work pool, find the next groups of CI tests of the edges through their processing progress, and execute them in parallel.

Creating and deleting threads usually takes less time, making it easy to maintain a batch of threads economically. Our method also can be recognized as a fork-join model [21]. At the beginning of each level, the main process forks some threads that read and write the same memory space to conduct CI tests in parallel. Then forked threads are joined into the main process and followed by the data merging operator. Fork-join model is commonly used in multi-thread parallelism because of the relatively low overhead for fork and join operation.

In particular, at the beginning of each depth, all the edges in the current graph G are pushed into the work pool with zero processing progress. Then, each time t edges are popped from the work pool and enumerate the CI tests from these edges. Each thread would then be responsible for processing a group of CI tests, where the number of CI tests is introduced as gs ($gs \geq 1$). When the gs CI tests are finished, decisions are made on whether the edge is required to be removed and whether the edge is required to be pushed into the work pool. Two decisions are made, including whether to accept the independence hypothesis of the group of CI tests and whether the edge is required to be pushed into the work pool. Specifically, the independence hypothesis of the group is accepted if any one of the CI tests in the group accepts its independence hypothesis; otherwise, the hypothesis is rejected. If the independence hypothesis of the group is accepted, or the edge has finished all its CI tests after processing this group, this means that the processing of the edge is complete, and thus the edge does not need to be pushed back to the work pool; otherwise, the edge would be pushed back to the work pool with its processing progress recorded as the last processed CI test. After that, t edges are popped from the work pool, and the next gs CI tests (according to the processing progress) of each edge are processed by t parallel threads again. This process is performed iteratively until the work pool is empty.

Intuitively, one can think of this process as multiple threads processing multiple CI tests on different edges in parallel, but a thread is not bounded to a fixed edge. When the processing of an edge is finished, the thread turns to process the CI tests of other edges immediately without waiting for other edges to be finished. This is due to the design of a dynamic work pool that monitors the processing progress of each edge. With the edge monitoring technique, the completed edges are terminated in time to reduce unnecessary CI tests. Moreover, we can better schedule the workloads among threads with the help of the design of the dynamic work pool. All the threads always process the CI tests that are required to be processed, and hence all the threads are active in the parallel region. As shown in Fig. 3, the CI tests in yellow are scheduled to thread 0 and the ones in blue are scheduled to thread 1. The CI tests of one edge are not necessarily processed by one thread.

The gs is a trade-off between the number of CI tests and memory accesses. In the parallel region, each thread processes

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

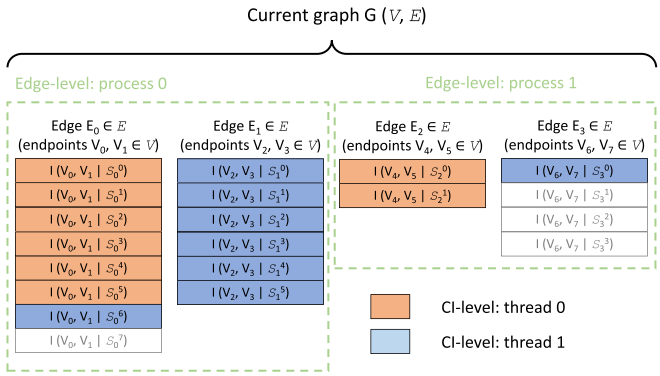


Fig. 3. Two different granularities of parallelism: edge-level parallelism and CI-level parallelism.

gs CI tests of the same edge $V_i - V_j$ each time and makes the decision according to the results. Hence, the CI tests in a group share the same form of $I(V_i, V_j|\mathcal{S}_n)$, $0 \leq n < gs$. Since V_i and V_j are common for the whole group, we propose to reuse them to reduce the memory accesses when traversing the Dataset. The reduced memory accesses increase as the increase of gs. However, more redundant CI tests are introduced at the same time, because all the CI tests in a group are required to be performed before making the final decision on whether the edge is required to be processed again. In a special case of $gs = 1$, no redundant CI tests are introduced. We carefully examine the effect of gs and observe that some small gs like 6 or 8 are good choices in practice.

It is worth noting that multi-thread parallelism is used when the depth $d \geq 1$. In depth $d = 0$, the conditioning set $S = \emptyset$ as the size of conditioning sets is equal to d (cf. Algorithm 1, Line 11). Specifically, for each edge (V_i, V_j) in G, only one CI test is required, which is $I(V_i, V_j|\emptyset)$ or simply a marginal independence test $I(V_i, V_j)$. In other words, we know in advance how many CI tests are required in depth zero, which is equal to $n(n-1)/2$, representing the number of edges in the complete undirected graph G over the node set V, where n represents the number of nodes. Consequently, the required computations for depth zero can be simplified. Therefore, the direct edge-level parallelism is applied to depth zero without the efficiency issue of load unbalancing.

C. Multi-Process Parallelism

In this subsection, we illustrate the proposed multi-process parallelism method Fast-BNS-v2, which distributes edge-level subtasks to processes and performs these subtasks in parallel. Generally, multi-thread parallelism is more suitable for finer subtasks, while multi-process parallelism is more suitable for coarser subtasks. Multi-process parallelism avoids race conditions and increases overall performance, because each process occupies its memory space and computing resources. To achieve multi-process parallelism, we must design inter-process communication techniques due to independent memory spaces. How to synchronize and exchange data and signals rapidly and precisely are critical steps in multi-process parallelism. Intuitively, we can allocate coarse subtasks to processes. Then we divide coarse subtasks into fine subtasks and assign threads to conduct these fine subtasks. To take advantages of processes and threads, we develop a two-level parallel PC-stable algorithm, applying CI test level parallelism with multi-threads and edge-level parallelism with multi-processes. For better utilization of multi-processes, we design two techniques to further boost the efficiency of Fast-BNS-v2: i) load balancing mechanism by dynamic work stealing to reduce CPU idle time, ii) one-side and collective communication policies to reduce data transfer overhead.

TABLE I
COMPARISON BETWEEN EDGE-LEVEL, SAMPLE-LEVEL, CI-LEVEL, AND TWO-LEVEL(CI-LEVEL + EDGE-LEVEL) PARALLELISMS

Granularity	Load balance	Reasonable workloads	Distributed setting
Edge-level	✗	✓	✓
CI-level	✓	✓	✗
Two-level	✓	✓	✓

1) Edge-Level Parallelism and CI-Level Parallelism: The most natural scheme to parallelize the PC-stable algorithm is to parallelize the processing of different edges inside each depth, which is a coarse-grained parallelism. The order-independent property of PC-stable makes it suitable for parallelizing at each depth. In other words, an edge deletion does not affect the processing of other edges at the same depth, and thus the processing of different edges can be done in parallel. In each depth d, it parallelizes the CI tests for each edge (i.e., the for-loop in Line 7 of Algorithm 1), dedicating $\frac{|\mathcal{E}_d|}{t}$ edges to each thread, where t represents the number of threads or processes and $|\mathcal{E}_d|$ represents the number of edges to be processed in depth d. Fig. 3 illustrate the execution progress of CI-level parallelism and edge-level parallelism. The example in the figure contains four edges, and thus in the case of using two processes, each process has two threads. Each process is responsible for processing two edges. Specifically, process 0 is dedicated to edges E_0 and E_1, while E_2 and E_3 are assigned to process 1.

Edge-level can allocate reasonable workloads to processes and run in multi-machines due to its small communication overhead. However, different edges have different numbers of CI tests, which needs to be carefully handled with load balancing policies. In comparison, CI-level parallelism is load balanced among threads in shared memory, but allocating a huge number of CI tests to multi-machines leads to huge communication overhead. In Fast-BNS-v2, we combine these two levels, applying edge-level parallel in multi-process and CI-level in multi-thread, to generate benefits from these two levels. Table I summarizes three parallelisms: edge-level, CI-level, and two-level (combination of edge-level and CI-level).

In a natural implementation of CI-level parallelism, we need to enumerate all the possible CI tests, totalling to $\binom{n}{r}$ CI tests. Then, the coordinator thread needs to distribute these CI tests to worker threads for evaluation. It requires a mere lift of the finger to appoint threads to conduct CI tests in shared memory space. Nevertheless, assigning these CI tests to processes produces notable communication overhead, because the volume of CI tests is huge. Inter-process communication, which is designed

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: Fast-BNS-v2.

1 **Input**: Node set V
2 **Output**: Skeleton Graph G
3 Graph $G \leftarrow$ FormCompletedGraph(V) $= V \times V$
4 Depth $d \leftarrow 0$
5 Need-remove tag $\mathcal{L}_{ij} \leftarrow False$ for each edge (V_i, V_j)
6 Let $a(V_i)$ represent adjacency nodes of V_i
7 **repeat**
8 // `multi-process parallelism`
9 **parallel for** *each process* r **do**
10 Edge sublist $\mathcal{T} \leftarrow$ FetchEdge(G)
11 **if** $\mathcal{T} \in \emptyset$ **then**
12 **break**
13 **repeat**
14 Sublist $\mathcal{T}' \leftarrow$ FetchCurrEdge($\mathcal{T}$)//$\mathcal{T}' \in \mathcal{T}$
15 CI test pool $\mathcal{C} \leftarrow$GenerateCItest($\mathcal{T}'$)
16 // `multi-thread parallelism`
17 **parallel** $\mathcal{C}_i = I(V_i, V_j|\mathcal{S})$ in $\mathcal{C}$ **do**
18 // $\mathcal{S}$ is speration set
19 **if** $I(V_i, V_j|\mathcal{S})$ holds **then**
20 $\mathcal{L}_{ij} \leftarrow True$
21 Skip CI tests from (V_i, V_j)
22 **until** $\mathcal{T}$ is empty
23 **allreduce** $\mathcal{L}$ by bool *or* operator
24 **SIMD-delete** (V_i, V_j) from G if $\mathcal{L}_{ij} = True$
25 Let $d = d + 1$
26 **until** all edges (V_i, V_j) in G satisfy $|a(V_i)\backslash\{V_j\}| < d$

to exchange data between distributed memory in multi-process parallelism, is powerless to CI-level data exchange. Therefore, multi-process parallelism is not suitable at the CI-level.

For multi-process parallelism, edge-level with larger granularity is more suitable in BN structure learning. Algorithm 2 shows the pipeline of our proposed Fast-BNS-v2. Similar to Algorithm 1, we form a completed graph and set the depth to 0 at first (Line 3 to 4). Here, we create a Boolean tag array to record edges that need to be removed (Line 5). We check edges depth by depth until we cannot find enough variables for separation sets (Line 7 to 26). For each process, it continually fetches new edge sublists until there are no more new sublists left (Line 9 to 22). After fetching edge sublist $\mathcal{T}$, we get a sublist $\mathcal{T}'$ from $\mathcal{T}$ and decompose these edges to CI tests (Line 14 to 15). Then we use multi-thread parallelism and CI-level parallelism to conduct CI tests, the same as the implementation in Fast-BNS-v1 (Line 17 to 21). If one CI test holds, we tag the edge by a need-remove label (Line 20) and skip CI tests from the same edge (Line 21). After finishing all CI tests at this depth, we apply a collective communication operation: allreduce, to synchronize the need-remove tag between processes (Line 23) and delete these edges by SIMD edge list deletion (Line 24).

2) Load Balancing Policies Among Processes: Before judging an edge, we cannot count how many CI tests should be done to recognize the existence of the edge. The number of each edge's CI tests varies: one in the best case and exponential to the number of neighbors in the worst case. Therefore, one-time allocation of work leads to load imbalancing, because the time needed by a process to test an edge is unpredictable. Our goal is to schedule processes to obtain approaching execution time and avoid idle time, improving the utilization of the machine. At first, we try choosing a proxy function to estimate the number of CI tests for each edge. Following the worst case $O(\binom{n}{r})$ for one edge, we allocate these edges to processes that have the same amount of total combinations. However, this one-time load balancing policy is inefficient, because of the inaccurate proxy. Inspired by the dynamic load balancing method–work stealing, we divide the edge list into rw_d sublists, w_d is the size of the sublists in depth d. Every process owns one sublist at the beginning. Once a process finishes checking a sublist, the unblocking scheduler provides the process with a new sublist until there are no more new sublists using one-side communication (mentioned in Section III-C2), while other processes are not interrupted by the scheduler from the execution. A naive way is to set each sublist to the same length. According to flow control theory, we can reduce communication overhead by setting head sublists longer than bottom sublists, because all the processes have the same starting time and run the tasks synchronously at the start. Thus, we reduce synchronization frequency here to decrease communication costs. When all the processes approach to the end, shorter sublists can occupy small process time intervals. However, there is also a lower bound for the length of sublists to avoid over-communication. To balance the communication overhead and computation overhead, we need to set the length of the sublists to a reasonable size. After finishing all sublists in this depth, we apply allreduce to collect results from different processes.

3) Process Communication Optimization: The communication cost for multi-process parallelism has significant impact on the overall efficiency. Therefore, we propose three techniques to reduce communication cost. First, after finishing PC-stable step 1, we take different policies to perform step 2 and step 3. We select only one process to execute in one machine under the multi-process setting. In the multi-machine setting, each machine executes these two steps and obtains results simultaneously to avoid communication for results transfer. Second, to shorten the time consumed by communication in one depth, we only maintain one integer variable–the start position of the unchecked edge list. Once a process completes its current subtask, it fetches the start position and updates the variable by adding the new sublist's length. This strategy is implemented by one-side communication [22] that is an unblocking communication, decoupling data exchange and task execution. We create a shared memory window in the main process, which is accessed for other processes to fetch and update. Fetching new sublists does not disturb the task running in the main process by one-side communication. A lock guarantees that only one process can access the window simultaneously to avoid racing problems. Third, after conducting CI tests in one depth, we synchronize results in different processes by the allreduce operation. Each process maintains a Boolean array to represent the existence of each edge and apply *and* operation with arrays from other processes to merge results of CI tests. To complete this step, naive implementation collects these arrays from processes first, then calculates results, and finally sends the result array to

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

each process. Allreduce efficiently combines communication and computation together and has strong scalability that is easy to extend to multiple machines. There are some fast allreduce implementations, such as ring-allreduce for small datasets and tree-allreduce for large datasets. Here we adopt tree-allreduce that is suitable for the large number of CI tests in our method.

D. Further Enhancing Fast-BNS

We find four issues in existing implementations: [13], [14], [15]. First, it is inefficient to distribute the CI tests of the edges with the same endpoints to different threads, as it may cause unnecessary CI tests. Second, the memory access pattern is irregular, because the required values of one CI test are not necessarily stored sequentially. Third, the number of CI tests is large, which requires much memory to store the indices of the conditioning sets for all the CI tests. Fourth, updating elements in the edge list following one-by-one single-point deletions has low-efficiency, because removing one element from the array costs linear time, leading to squared complexity in total. These issues degrade performance in the naive composition of our proposed method above. Our method requires careful design optimizations to enhance its efficiency. Here, we aim to tackle these four issues to further improve the overall efficiency of Fast-BNS.

1) Grouping CI Tests of Edges With the Same Endpoints: We view the edges with the same endpoints, such as edges $V_i - V_j$ and $V_j - V_i$, as the same edge in Fast-BNS-v1, instead of separating them as in the original PC-stable algorithm, because it is inefficient to separate the CI tests of two such edges. For instance, given the edge between V_i and V_j, we need to perform the CI tests conditioning on the variables in $adj(G, V_i)\backslash\{V_j\}$ and $adj(G, V_j)\backslash\{V_i\}$. However, if we first perform the CI tests conditioning on the variables in $adj(G, V_i)\backslash\{V_j\}$ and the edge between V_i and V_j is removed, then the CI tests conditioning on variables in $adj(G, V_j)\backslash\{V_i\}$ are unnecessary. Therefore, we solve this dependency by grouping the CI tests of the edges with the same endpoints together to reduce the number of CI tests to be performed, and thus improve the efficiency. If the CI tests between V_i and V_j accept the independence hypothesis when conditioning on $S \in adj(G, V_i)\backslash\{V_j\}$, Fast-BNS-v1 does not perform the CI tests conditioning on the variables in $adj(G, V_j)\backslash\{V_i\}$.

2) Using a Cache-Friendly Data Storage: A key step of the algorithm is to compute the contingency table. For example, to test $I(X, Y|\{Z_1, Z_2\})$, we need to traverse the whole Dataset and obtain the values of X, Y, Z_1 and Z_2 for all the samples. For a naive two-dimensional Dataset storage where each row represents one sample and each column represents one feature (i.e., one variable in BNs), we need to traverse all the rows and find four values for each row. Since X, Y, Z_1 and Z_2 are not necessarily stored next to each other, there are many random memory accesses and hence every memory access can be a cache miss. Therefore, we instead propose to transpose the data matrix, i.e., using each row to represent a feature and each column to represent a sample, which is cache-friendly data storage. For the previous example, after the first four memory accesses of the first column, the upcoming iterations access addresses that are right next to the previously fetched values in the cache. As a result, Fast-BNS-v1 only has four cache misses at the beginning and the rest can be served from four cache lines.

3) Generating Conditioning Sets On-the-Fly: In the PC-stable algorithm, processing an edge may require many CI tests, depending on the current depth d and the number of adjacent nodes of its endpoints. In a naive implementation, we must generate all the CI tests of an edge before processing the edge. This approach is inefficient because additional memory is required to store the indices of the conditioning sets of all the CI tests. Given an edge $V_i - V_j$, the selection of its conditioning sets $\mathcal{S} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{\binom{p}{q}-1}\}$ can be viewed as a combination problem of choosing q elements from $p = |a(V_i)\backslash\{V_j\}|$ elements at a time (cf. Algorithm 1, Line 11). Fast-BNS-v1 implements a combination function to generate $\mathcal{S}$ in lexicographical order [23]. Given p, q and r, the combination function of Fast-BNS-v2 is able to directly compute the vector $\mathcal{S}_r$ without computing the whole set $\mathcal{S}$. With the help of the combination function, all the indices of conditioning sets of the CI tests can be computed on-the-fly and also in parallel. Therefore, the work pool of Fast-BNS-v1 only contains the edges to be processed and their processing progress (i.e., r). No additional memory is required for storing the indices of the conditioning sets of the edges.

4) Edge List Deletion With SIMD: In our previous implementation, Fast-BNS-v1 [16], it deletes elements one by one. The deletion of one element on the edge list requires linear complexity. Thus, updating the whole edge list has a $O(n^2)$ time complexity, which is time consuming for the overall algorithm. The edge list updating at the end of each level is a Boolean mask operation, which selects elements from the old edge list to build a new one, referring to a Boolean array with the same length as the old edge list. Based on this observation, Fast-BNS-v2 creates a new edge list and then copies the required elements to it. With the memory allocation in $O(n)$ and copy operation in $O(n)$, the overall complexity is reduced to $O(n)$, tremendously boosting the edge list deletion efficiency. Experimental results show that the execution time of the operation approaches $O(1)$.

E. Performance Analysis

Here, we analyze the theoretical speedups provided by our optimizations discussed earlier in this Section. The optimizations to be analyzed include: i) using CI-level parallelism with the design of the dynamic work pool; ii) grouping the CI tests of the edges with the same endpoints; iii) using a cache-friendly data storage. We also discuss the communication overhead in this section.

1) CI-Level Parallelism and Edge-Level Parallelism: In the depth d of the graph G, there are $|\mathcal{E}_d|$ edges to be processed. Each edge E_i, with two endpoints ep_i^1 and ep_i^2, has a number of CI tests. The number of adjacent nodes of the two endpoints, denoted by $a_i^1 = |adj(G, ep_i^1)|$ and $a_i^2 = |adj(G, ep_i^2)|$, as well as the depth d and the results of the CI tests, determines the number of the CI tests. Specifically, each edge leads to at most

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

$\binom{a_i^1}{d} + \binom{a_i^2}{d}$ CI tests, while if one CI test accepts the independence assumption during the processing of one edge, the process of the edge is terminated in advance (i.e., the remaining CI tests become unnecessary).

For the case of t threads running in parallel, the edge-level parallelism assigns $\frac{|\mathcal{E}_d|}{t}$ edges to each thread. Ideally, the $\frac{|\mathcal{E}_d|}{t}$ edges assigned to each of the threads have the same number of CI tests to be processed. However, in practice, there is load unbalanced issue in most cases. For example, $\frac{|\mathcal{E}_d|}{t}$ out of the $|\mathcal{E}_d|$ edges process all the $\binom{a_i^1}{d} + \binom{a_i^2}{d}$ CI tests required for each edge E_i, while the other $\frac{(t-1)|\mathcal{E}_d|}{t}$ edges only process one CI test as they accept the independence assumption when handling their first CI test. In the worst case, the $\frac{|\mathcal{E}_d|}{t}$ edges that process all the required CI tests are assigned to the same thread p. In that case, the performance of the edge-level parallelism can be severely affected by this unbalanced workload, since all the threads must wait for the completion of the slowest thread p. Suppose that the time for each CI test is T_{CI}, then the required time for the edge-level parallelism under t threads is

$$T_1 = T_{CI} \sum_{i=1}^{\frac{|\mathcal{E}_d|}{t}} \left(\binom{a_i^1}{d} + \binom{a_i^2}{d} \right). \tag{1}$$

However, the proposed CI-level parallelism evenly distributes all the CI tests to each thread with the help of the dynamic work pool, and hence the required time is

$$T_2 = \frac{T_{CI}}{t} \left(\sum_{i=1}^{\frac{|\mathcal{E}_d|}{t}} \left(\binom{a_i^1}{d} + \binom{a_i^2}{d} \right) + \frac{(t-1)|\mathcal{E}_d|}{t} \right). \tag{2}$$

Therefore, the speedup provided by the CI-level parallelism with the design of the dynamic work pool is $S_{CI} = \frac{T_1}{T_2}$.

In our Fast-BNS-v2, we develop a two-level parallelism (multi-thread and multi-process levels). In the edge-level, we equip Fast-BNS-v2 with dynamic work stealing mechanism to balance the workload among processes, which is similar to the effect of dynamic work pool as in the multi-thread parallelism. Hence, the required time is

$$T_3 = T_{CI} \left(\frac{1}{t} \left(\sum_{i=1}^{|\mathcal{E}_d|} \left(\binom{a_i^1}{d} + \binom{a_i^2}{d} \right) + \text{argmax}_i \left(\binom{a_i^1}{d} + \binom{a_i^2}{d} \right) \right) \right). \tag{3}$$

The largest time lag among processes is the edge with the longest execution time and this edge is at the last position of the queue.

2) Grouping CI Tests of Edges With the Same Endpoints: This optimization provides speedup by reducing unnecessary CI tests. Consider the case of depth d that has $|\mathcal{E}_d|$ edges to be processed, for the edge between V_i and V_j, since edges $V_i - V_j$ and $V_j - V_i$ are viewed separately in the original PC-stable algorithm, we need to perform the CI tests considering two sets, i.e., $adj(G, V_i)\backslash\{V_j\}$ and $adj(G, V_j)\backslash\{V_i\}$. Therefore, we need to consider $2|\mathcal{E}_d|$ sets in total for the $|\mathcal{E}_d|$ edges in depth d. However, by grouping the CI tests of the edges $V_i - V_j$ and $V_j - V_i$, if the CI tests accept the independence hypothesis when considering the set $adj(G, V_i)\backslash\{V_j\}$, Fast-BNS-v2 does not consider the set $adj(G, V_j)\backslash\{V_i\}$. Suppose ρ_d is the ratio of edge deletion for depth d. Then, this optimization reduces the CI tests by $\rho_d|\mathcal{E}_d|$ unnecessary sets. That is, only $2|\mathcal{E}_d| - \rho_d|\mathcal{E}_d|$ sets need to be considered. Therefore, if we ignore the difference in the number of CI tests for different sets, the speedup brought by grouping CI tests is

$$S_{grouping} = \frac{2|\mathcal{E}_d|}{2|\mathcal{E}_d| - \rho_d|\mathcal{E}_d|} = \frac{2}{2 - \rho_d}.$$

3) Using a Cache-Friendly Data Storage: This optimization provides speedup by reducing the ratio of cache misses. The memory accesses of PC-stable mainly come from the accesses to the Dataset when computing the contingency table. For the CI test $I(X, Y|\{Z_1, \ldots, Z_d\})$ in depth d, we need to access the values of X, Y, Z_1,..., Z_d of the m samples in the Dataset, where each value is 4 bytes in memory. Suppose that the cache line size is B bytes. First we consider the access to the $\frac{B}{4}$ samples. Regarding the cache-unfriendly data storage, since X, Y, Z_1,..., Z_d are not necessarily stored next to each other, every memory access can be a cache miss. Therefore, the required time of accessing the values of $\frac{B}{4}$ samples for the cache-unfriendly data storage is

$$T_4 = T_{DRAM}(d+2)\frac{B}{4},$$

where T_{DRAM} represents the access time of main memory (caused by the cache misses). However, for the cache-friendly data storage, it only has $(d+2)$ cache misses for the access of the first sample, and the rest accesses of the $(\frac{B}{4} - 1)$ samples can be served from the $(d+2)$ cache lines since they access addresses that are next to the previously fetched values in the cache. Therefore, the required time of accessing the values of $\frac{B}{4}$ samples for the cache-friendly data storage is

$$T_5 = T_{DRAM}(d+2) + T_{cache}(d+2)\left(\frac{B}{4} - 1\right),$$

where T_{cache} is the cache access time. Since m is often much greater than B, the access time to the whole Dataset is a multiple of the access time to the $\frac{B}{4}$ samples. Therefore, the speedup provided by the cache-friendly data storage is $S_{cache} = \frac{T_4}{T_5}$.

4) Overall Speedup: To conclude, the performance improvement of Fast-BNS can be computed as $S = S_{CI} \cdot S_{grouping} \cdot S_{cache}$. For example, let us consider the case where the number of threads $t = 4$ and the depth $d = 2$. Suppose that there are $|\mathcal{E}_d| = 1200$ edges at the beginning of depth 2 and 480 edges at the end, and hence, the edge deletion ratio $\rho_d = 0.6$. Suppose each edge has the same number of adjacent nodes, which is the mean degree of the graph (we assume the mean degree is 10). Hence, every a_i^1 and a_i^2 in (1) and (2) can be replaced by the mean degree 10. Moreover, the cache line size B is often 64 bytes. The cache access time T_{cache} is typically less than the access time of main memory T_{DRAM} by a factor of 5 to 10, and we assume $\frac{T_{DRAM}}{T_{cache}} = 8$. Therefore, we can calculate the ideal speedup provided by Fast-BNS under these circumstances: $S_{CI} = 3.87$, $S_{grouping} = 1.43$, $S_{cache} = 5.57$,

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

and hence the speedup $S = 30.8$. However, this theoretical analysis only provides a general speedup of Fast-BNS, the situation in the experiments is often more complicated than the ideal case. For example, the values of $|\mathcal{E}_d|$, ρ_d, a_i^1 and a_i^2 all depend on the specific problem to be solved, and they are usually unknown beforehand.

5) Communication Analysis: In this part, we discuss the communication overhead of the Fast-BNS. At the beginning of each depth, each process fetches its sublists by one-sided communication. The total number of sublists is rw_d, so this one-sided communication costs rw_d bytes. After all the processes finish the CI tests within each depth, allreduce is performed to synchronize the results of CI tests from different processes. The size of edges in depth d is $|\mathcal{E}_d|$, so our method transfers $(r+1)|\mathcal{E}_d|$ bytes in this depth. Then we conclude that the total communication amount is $\sum_d (r+1)|\mathcal{E}_d| + rw_d$ bytes.

F. Generalization of Our Method

Our proposed method can easily be extended to other machine learning models. For example, in Convolutional Neural Network (CNN), we can apply multi-process parallelism between batches and multi-thread parallelism between samples in one batch. In addition, load balancing techniques with communication optimization are suitable for multi-process paradigms, such as multi-GPU training. Grouping samples or improving cache efficiency benefit tasks which have huge datasets. Generating medium results on-the-fly helps reduce memory overhead in other tasks. SIMD array modification can accelerate tensor operations in deep learning methods.

IV. Experiments

We conduct extensive experiments to demonstrate the performance of our method and compare the results to state-of-the-art methods.

A. Experiment Setting

We implemented our Bayesian network structure learning methods in C++ with OpenMPI and OpenMP libraries and compared their performance to existing methods. All the single-machine experiments were conducted on a Linux machine with two 28-core 2.6 GHz Intel Xeon Gold 6348 CPUs and 768 GB main memory. For the multi-machine setting, we employ 8 machines with above environment. Specifically, we compared the sequential implementations of Fast-BNS-v2 and Fast-BNS-v1 with three different open-source packages including bnlearn [9], pcalg [10] and tetrad [11]. We also compared Fast-BNS-v2 with the recent multi-threaded implementations in bnlearn [13] and parallel-PC [15]. Bnlearn, pcalg and parallel-PC are all R packages, while tetrad is implemented in Java. Experiments were also conducted to compare Fast-BNS-v1 and Fast-BNS-v2 to show the enhancement in this work, in comparison with our previous work [16]. There are other parallel works for PC-stable, such as the work [14], however, that algorithm is not open-source. Moreover, their experimental results show that its parallel implementation achieves lower speedup over its sequential implementation compared with the speedup of the parallel implementation of Fast-BNS-v2 over its sequential counterpart. Therefore, we did not compare Fast-BNS-v2 with it.

TABLE II
BNs From Which Datasets Used are Generated

Dataset	#nodes	#edges	#samples	max depth
Alarm [24]	37	46	15000	4
Insurance [25]	27	52	15000	7
Hepar2 [26]	70	123	15000	9
Munin1 [27]	186	273	5000	6
Diabetes [28]	413	602	5000	5
Link [29]	724	1125	5000	12
Munin2 [27]	1003	1244	5000	6
Munin3 [27]	1041	1306	5000	6

Datasets used in our experiments were obtained from eight benchmark BNs of different sizes listed in Table II, where the last four Datasets are large-scale BNs. These networks represent problems from different fields and have been widely used for comparative purposes in the literature of BN structure learning. We obtained 5,000 samples of data with no missing values from each network. Besides, more Datasets were obtained for the first four networks with 10,000 and 15,000 samples to test the impact of different sample sizes. We used G^2 test statistic to perform the CI tests while setting the significance level α to 0.05 in all experiments. The accuracy of Fast-BNS-v2 and Fast-BNS-v1 is exactly the same as the other PC-stable algorithm implementations because the two methods are accelerated implementations of the same PC-stable algorithm. Hence, we omit reporting the results on accuracy comparison.

B. Overall Comparison

In the overall evaluation of Fast-BNS-v2, we compared the execution time of both sequential and parallel implementations of Fast-BNS-v2 with the existing implementations on the eight Datasets with 5000 samples. Specifically, we compared the sequential version of Fast-BNS-v2 with the PC-stable implementations in bnlearn [9], pcalg [10] and tetrad [11] packages; we also compared the parallel version of Fast-BNS-v2 with the multi-threaded implementation in bnlearn [13] and parallel-PC [15]. The gs of Fast-BNS-v2 was set to 1 for all the experiments here. For comparing the parallel implementations, we varied the number of MPI processes t from 1 to 32 and chose the one with the shortest execution time. We terminated the experiment if the execution time exceeded 48 hours with no results obtained.

The experimental results are summarized in Table III. As seen from the "Speedup" columns of the table, the sequential implementation of our proposed Fast-BNS-v2 often achieves two to three orders of magnitude speedup over tetrad and pcalg, and can be 1.07 to 82 times faster than the sequential version of bnlearn. The speedups of Fast-BNS-v2 are mainly due to the careful optimizations, including SIMD edge list deletion, grouping CI tests of the edges with the same endpoints, using cache-friendly data storage and generating conditioning sets on-the-fly. These general optimizations can be applied to both sequential and parallel implementations.

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

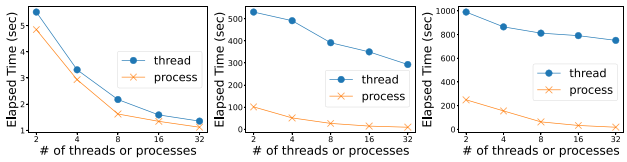
TABLE III
EXECUTION TIME COMPARISON OF FAST-BNS-V2 WITH OTHER IMPLEMENTATIONS UNDER BOTH SEQUENTIAL AND PARALLEL SETTING

Dataset	Sequential implementation							Parallel implementation				
	Execution time (sec)				Speedup			Execution time (sec)			Speedup	
	bnlearn	tetrad	pcalg	ours	bnlearn	tetrad	pcalg	bnlearn	parallel-PC	ours	bnlearn	parallel-PC
Alarm	0.42	5.38	53.8	0.231	1.8	23	230	0.42	15.4	0.019	24.5	890
Insurance	0.38	13	71.9	0.350	1.07	37	203	0.34	25.4	0.038	9.2	687
Hepar2	4.03	37.7	209	1.75	2.29	21	76	2.82	158	0.19	15.2	852
Munin1	111	770	2160	18.7	6.0	41.4	116	16.5	162	1.21	13.6	134
Diabetes	113k	$>$ 2 days		14.6k	7.7	$>$ 7.4		7640	54k	743	10.0	72.7
Link	$>$ 2 days			30.1k	$>$ 2.7			49.4k	$>$ 2 days	862	58.7	$>$ 199
Munin2	27.9k	$>$ 2 days		340	82	$>$ 507		2734	$>$ 2 days	9.66	283	$>$ 17800
Munin3	38.7k	$>$ 2 days		1027	37.7	$>$ 168		3621	$>$ 2 days	15.4	235	$>$ 11200

Speedup of Fast-BNS-v2 over each compared method is also reported. "Ours" represents "Fast-BNS-v2".

TABLE IV
COMPARISON OF FAST-BNS-V2 AND FAST-BNS-V1

Dataset	Seq. version		Par. version		
	Exec time (s)		Exec time (s)		speedup
	FBNSv1	FBNSv2	FBNSv1	FBNSv2	
Alarm	0.233	0.231	0.017	0.019	1.0
Insurance	0.353	0.350	0.037	0.038	1.0
Hepar2	1.759	1.745	0.19	0.19	1.0
Munin1	18.9	18.7	1.78	1.21	1.47
Diabetes	14.7k	14.6k	1203	743	1.62
Link	30.2k	30.1k	4349	862	5.04
Munin2	386.9	340.2	293	9.66	30.3
Munin3	1197	1027	751	15.42	48.7



(a) Munin1 (b) Munin2 (c) Munin3

Fig. 4. Comparison of multi-thread parallelism (Fast-BNS-v1) and multi-process parallelism (Fast-BNS-v2) (Strong scaling).

When comparing the parallel implementations, Fast-BNS-v2 is often much faster than parallel-PC, and can run 9.2 to 283 times faster than the parallel bnlearn. It is worth noting that for some small Datasets, such as *Alarm* and *Insurance*, bnlearn failed to get improvements by the multi-threaded techniques, and thus the same results were obtained for its sequential and parallel implementations. Another observation is that Fast-BNS-v2 always achieves its shortest execution time when $t = 32$. Moreover, the execution time of the sequential version of Fast-BNS-v2 can be reduced by more than 85% using multi-processes. The experiment on the *Link* Dataset is the task taking the longest time to complete. This task ran more than 2 days using the existing sequential implementations bnlearn, tetrad, pcalg and the parallel implementation parallel-PC, while the execution time is significantly reduced to about 14 minutes in Fast-BNS-v2.

C. Fast-BNS -V2 vs Fast-BNS-V1

To get a better understanding of the improvement of Fast-BNS-v2, we compare it with Fast-BNS-v1. Table IV shows the results. In the sequential version comparison, Fast-BNS-v2 brings up to 16% improvement, because of its SIMD edge list deletion. We observed that Fast-BNS-v1 took about 170 seconds to modify the edge list array, while Fast-BNS-v2 spent only 0.02 s on this part with the SIMD edge list deletion. In the parallel version, the boost of Fast-BNS-v2 is insignificant in tiny Datasets due to the overhead of multi-process initialization. When the size of Datasets increases, such as Diabetes, Link, Munin2, and Munin3, Fast-BNS-v2 shortens the execution time by 5 to 48 times over Fast-BNS-v1, because of our multi-process parallelism and enhancing techniques including SIMD edge list deletion, load balancing methods, and communication policies.

To further discover how Fast-BNS-v2 runs faster than Fast-BNS-v1, we contrast multi-thread parallelism and multi-process parallelism. In the multi-process version, we set each process to have only one thread, and ran it on single machine. Fig. 4 shows Fast-BNS-v2 with multi-process parallelism reducing the execution time (Strong scaling), which is more noticeable in larger Datasets. Two versions show similar execution times in Munin1, while the multi-process version is significantly faster than the multi-thread version in Munin2 and Munin3. Another observation from Fig. 4 is that increasing the number of threads is useful when the number is less than 8, while 32 threads' performance is similar to 16 threads'. The reason is that multi-thread maintains a dynamic thread pool in one process, which contains threads that share the process's resources. In comparison, each process holds its resources and no need to disturb other processes. Therefore, multi-process parallelism can reduce training time ideally due to the isolated running environments. For example, the 4 processes' execution time is about half of the 2 processes'.

To further investigate why Fast-BNS-v2 is faster, we used perf Linux profiler to obtain the detailed measurements for Fast-BNS-v1 and Fast-BNS-v2. The results on *Munin2* and *Munin3* are shown in Table V. We can observe that both the parallel and sequential versions of Fast-BNS-v2 increase CPU utilization and FLOPS more than Fast-BNS-v1. This is because multi-process parallelism enhances resource utilization, while multi-thread parallelism in one process has an upper bound for CPU utilization posed by the OS.

We also measure the elapsed time under different numbers of samples in Fig. 5 (Weak scaling) in three datasets. In Munin1, the elapsed time grows linearly, while the growth of time between 15000 and 10000 is large than that between 10000 and 5000 in Munin2 and Munin3. Larger datasets lead to more elapsed time for loading data and counting terms for CI tests, which is not significant in smaller datasets.

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

TABLE V
DETAILED COMPARISON OF THE PARALLEL AND SEQUENTIAL VERSIONS OF FAST-BNS-V2 AND FAST-BNS-V1

Munin2	L1-cache accesses	L1-cache misses (rate)	LL-cache accesses	LL-cache misses (rate)	FLOPS	CPU utilization
Fast-BNS-v2-par	1.9×10^{12}	9.5×10^{10} (5.02%)	4.3×10^{9}	8.0×10^{8} (19.0%)	9.9×10^{8}	29.252
Fast-BNS-v1-par	9.9×10^{11}	1.9×10^{10} (19.1%)	1.1×10^{10}	1.8×10^{9} (16.4%)	4.1×10^{8}	11.1
Fast-BNS-v2-seq	9.9×10^{11}	1.9×10^{10} (19.1%)	5.8×10^{9}	2.3×10^{8} (4.01%)	5.2×10^{7}	1
Fast-BNS-v1-seq	9.9×10^{11}	1.9×10^{10} (19.1%)	6.8×10^{9}	3.4×10^{8} (5.11%)	5.1×10^{7}	1
Munin3	L1-cache accesses	L1-cache misses (rate)	LL-cache accesses	LL-cache misses (rate)	FLOPS	CPU utilization
Fast-BNS-v2-par	3.5×10^{12}	1.3×10^{11} (3.71%)	8.7×10^{9}	1.2×10^{9} (13.8%)	1.6×10^{9}	29.6
Fast-BNS-v1-par	2.6×10^{12}	5.8×10^{10} (2.22%)	2.9×10^{10}	1.4×10^{10} (24.1%)	7.2×10^{8}	13.2
Fast-BNS-v2-seq	2.3×10^{12}	4.1×10^{10} (1.78%)	1.5×10^{10}	6.0×10^{8} (3.97%)	7.0×10^{7}	1
Fast-BNS-v1-seq	2.6×10^{12}	5.8×10^{10} (2.22%)	2.8×10^{10}	7.0×10^{9} (25.4%)	6.5×10^{7}	1

"-Seq" and "-par" represent sequential and parallel implementation, respectively.

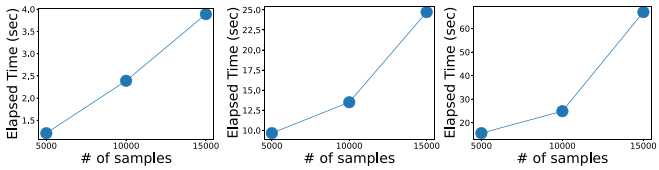


Fig. 5. Elapsed time for different numbers of samples (Weak scaling).

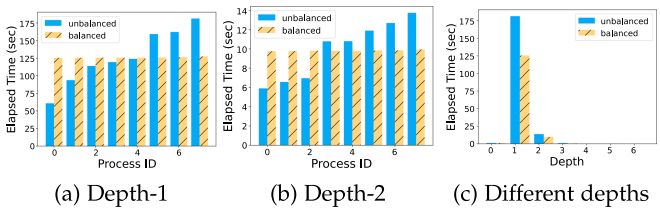


Fig. 6. Fast-BNS-v2 without and with load balancing.

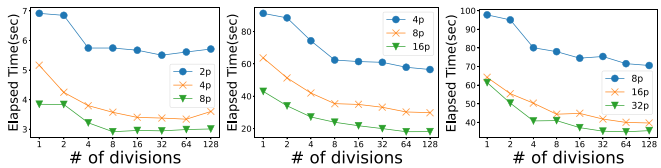


Fig. 7. Elapsed time of different numbers of divisions for load balancing.

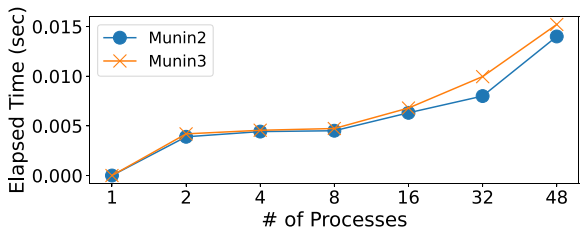


Fig. 8. Communication time for different # of processes.

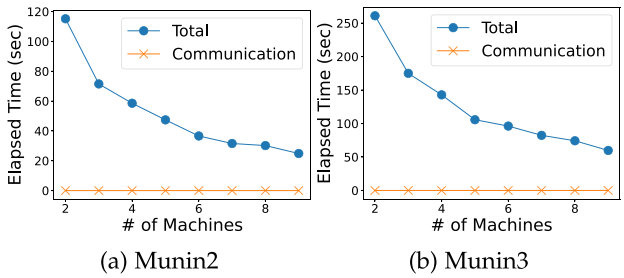


Fig. 9. Communication cost in distributed systems.

D. Load Balancing and Communication of Processes

In this set of experiments, we study the effectiveness of our proposed load balancing policies, where we allocated equal edge amounts to each process and used Munin3 as an example. The results of Fig. 6(a) and (b) show the depth-1 and depth-2 execution time in the Munin3 Dataset, where depth-1 and depth-2 have one and two variables for each separation set, respectively. As we can see from the results, Fast-BNS-v2 without load balancing optimizations leads to load unbalancing. In depth-1, the fastest process takes 60 seconds while the slowest costs 180 seconds, due to the different workload of each edge. Processes need to synchronize at the end of each level, leading to idle problems. The execution time of processes converges to about 125 seconds, after being equipped with the load balancing policies (discussed in Section III-C). In depth-2, our method reduces 30% execution time of the slowest process and improves the utilization. Fig. 6(c) shows the time distribution for the balanced method and the unbalanced method in different depths. Depth-1 and depth-2 occupy the most elapsed time in the two methods, while the time of other depths is negligible.

Our load balancing policies have a hyperparameter l which is the number of sublists in dynamic work stealing. As the number of sublists may affect the overall efficiency, here we varied l to better understand its effect. Fig. 7 illustrate the execution time with different l. For the Munin1 Dataset, 4 or 8 sublists are enough, while 16 or 32 are more suitable for Munin2 and Munin3. As we can see from the figure, when l increases to 128, higher communication costs decreases the execution efficiency.

Our proposed multi-process parallelism can naturally run on a distributed environment. Here, we study the scalability of Fast-BNS-v2, and analyze its communication cost. We measured the communication cost in one-machine and multi-machine settings. In the one-machine setting, we observe that the communication cost is shorter than 15 milliseconds as shown in Fig. 8, which shows the time in Muini2 and Munini3. The results illustrate that the task is computation intensive and communication costs a small amount of time in total execution progress. In the multi-machine settings, we ran Fast-BNS-v2 in 2 to 9 machines, where each machine has only one process, and inter-process communications are inter-machine communication. Fig. 9 shows the results. The execution time is reduced as the number of

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

machines increases while the communication cost remains low. We observe that the speedup does not improve linearly with the growth of processes. This is mainly because: imperfect balanced workload among the nine machines (even though the imbalance is not larger than one edge), the overhead of setting up the computation environment for each process, and not all the steps being parallelized in our solution.

V. Related Work

Bayesian networks (BNs) are powerful models for representation learning and reasoning under uncertainty in artificial intelligence. BNs have recently attracted much attention within the research and industry communities. A crucial aspect is to learn the dependency graph of a BN from data, which is called *structure learning*. In this paper, we categorize the related work on BN structure learning into two groups: score-based approaches and constraint-based approaches.

Score-based approaches [30], [31], [32], [33] seek the best DAG according to scoring functions that measure the fitness of BN structures to the observed data. Widely adopted scores include BDeu, BIC, and MDL. However, the number of possible DAGs is super-exponential to the number of variables [4]. Hence, many score-based approaches employ heuristics, like greedy search or simulated annealing, in an attempt to reduce the search space. Such approaches can easily get trapped in local optima [12]. The optimization techniques in this paper focus on the constraint-based approaches which tend to scale better to high-dimensional data.

Constraint-based approaches [6], [7], [34] perform structure learning using a series of statistical tests, such as Chi-square test, G^2 test and mutual information test, to learn the conditional independence relationships among the variables in the model. The DAG is then built according to these relations as constraints. In this work, we focus on accelerating PC-stable algorithm for learning BN structures. Indeed, many common constraint-based algorithms are similar to or variants of the PC or PC-stable algorithm [6], [7], such as GES [35] combining PC and greedy search, MMHC [36] adopting benefits from PC and GES, IAMB [36] search Markov Blanket from local to global by CI tests, and Hiton-PC [37] with heuristic search for fast training. Unlike score-based approaches, it is generally non-trivial to perform algorithmic improvements for constraint-based approaches using general-purpose optimization theory.

There are some well-known open-source BN libraries which contain the implementation of the PC-stable algorithm, such as bnlearn [9], pcalg [10] and tetrad [11]. Meanwhile, since the recent parallel computing platforms, such as multi-core CPUs and GPUs, have emerged to efficiently address various computational machine learning problems [38], [39], [40], there are several research works that focus on the acceleration of the PC-stable algorithm using parallel techniques on CPUs [13], [14], [15]. The key idea is to parallelize the processing of different edges inside each depth, which is an intuitive idea due to the order-independent property of the PC-stable algorithm. However, the edge-level parallelism is load unbalanced, because the workload of the conditional independence tests for different edges is highly skewed. This paper improves the efficiency of the PC-stable algorithm using CI-level parallelism to boost the applications of the BN structure learning. Parallel techniques are also applied to Markov Blanket methods [41]. Nowadays, GPU parallel techniques show a large potential for BN learning acceleration. gpuPC [42] and cuPC [20] propose GPU-based PC and PC-stable accelerations. However, GPU struggles to handle divergence, such as an if-statement, which is an essential part of CI tests. In addition, GPU is hard to arrange irregular subtask sizes in GPU and balance the workload among processes or threads, leading to difficulties for BN learning implementation.

TABLE VI
Comparison Between CPU and GPU Parallelism for BN Learning

Device	Branch prediction	Flexibility	Many threads	Parallel granularity
CPU	✓	✓	✗	Two-level
GPU	✗	✗	✓	CI-level

We compare CPU and GPU features for BN learning in Table VI. CPUs equipped with Branch prediction can better handle the computation of CI tests. CPUs also have better flexibility and universality, accommodating sequential, multi-thread, multi-process, and multi-machine executions. In comparison, GPUs exhibit certain limitations in multi-process or multi-machine execution due to their weaker support for process isolation and the inherent complexity of data transfer and coordination mechanisms between multiple GPUs. Meanwhile, the number of threads or processors in GPUs is more than that in CPUs, which is suitable for simpler instructions applied to different data. Consequently, CPU parallelism demonstrates distinct advantages for complex parallel computing scenarios like hybrid edge-level and CI-level parallelism, while GPU parallelism only supports CI-level or finer-grained level parallelism.

VI. Conclusion

Bayesian network structure learning has been a popular topic for causal discovery and reliable machine learning. This paper proposes a novel parallel and distributed method (named Fast-BNS) to accelerate the PC-stable algorithm, one of the most commonly used constraint-based structure learning algorithms. Past methods only focus on multi-thread parallelism, while Fast-BNS exploits multi-process and multi-thread parallelism, supporting multi-machines. In the data aspect, CI-level is overfine for multi-process parallelism, but edge-level parallelism suffers from the work-unbalanced issues. In Fast-BNS, we developed load balancing policies to overcome the work-unbalanced problem, while making use of CI-level parallelism with multi-threads to further enhance the performance. Our experimental results have shown that Fast-BNS can be 9 to 235 times faster than the state-of-the-art implementations. When running in multiple machines, it saves 80% time of the one-machine implementation with a moderately low communication overhead.

Acknowledgment

Professor Ajmal Mian is the recipient of an Australian Research Council Future Fellowship Award (project number FT210100268) funded by the Australian Government.

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] S. K. Andersen, "Judea pearl, probabilistic reasoning in intelligent systems: Networks of plausible inference," *Artif. Intell.*, vol. 48, no. 1, pp. 117–124, 1991, doi: 10.1016/0004-3702(91)90084-W.

[2] D. Gunning and D. Aha, "Darpa's explainable artificial intelligence (XAI) program," *AI Mag.*, vol. 40, no. 2, pp. 44–58, 2019.

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

[4] R. W. Robinson, "Counting unlabeled acyclic digraphs," in *Combinatorial Mathematics V*. Berlin, Germany: Springer, 1977, pp. 28–43.

[5] X. Zhang et al., "Inferring gene regulatory networks from gene expression data by path consistency algorithm based on conditional mutual information," *Bioinformatics*, vol. 28, no. 1, pp. 98–104, 2012.

[6] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman, *Causation, Prediction, and Search*. Cambridge, MA, USA: MIT Press, 2000.

[7] D. Colombo et al., "Order-independent constraint-based causal structure learning," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3741–3782, 2014.

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

[9] M. Scutari, "Learning Bayesian networks with the bnlearn R package," 2009, *arXiv:0908.3817*.

[10] M. Kalisch, M. Mächler, D. Colombo, M. H. Maathuis, and P. Bühlmann, "Causal inference using graphical models with the R package pcalg," *J. Stat. Softw.*, vol. 47, no. 11, pp. 1–26, 2012.

[11] J. D. Ramsey et al., "TETRAD–A toolbox for causal discovery," in *Proc. Int. Workshop Climate Informat.*, 2018. [Online]. Available: https://cmu-phil.github.io/tetrad/manual/

[12] M. Scutari, C. E. Graafland, and J. M. Gutiérrez, "Who learns better Bayesian network structures: Accuracy and speed of structure learning algorithms," *Int. J. Approx. Reasoning*, vol. 115, pp. 235–253, 2019.

[13] M. Scutari, "Bayesian network constraint-based structure learning algorithms: Parallel and optimised implementations in the bnlearn R package," 2014, *arXiv:1406.7648*.

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

[15] T. D. Le, T. Hoang, J. Li, L. Liu, H. Liu, and S. Hu, "A fast PC algorithm for high dimensional causal discovery with multi-core PCS," *Trans. Comput. Biol. Bioinf.*, vol. 16, no. 5, pp. 1483–1495, 2016.

[16] J. Jiang, Z. Wen, and A. Mian, "Fast parallel Bayesian network structure learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 617–627.

[17] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A hardware/software Approach*. Houston, TX, USA: Gulf, 1999.

[18] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proc. IEEE Conf. High Perform. Comput. Netw.*, 2009, pp. 1–11.

[19] C. Meek, "Causal inference and causal explanation with background knowledge," 2013, *arXiv:1302.4972*.

[20] B. Zarebavani, F. Jafarinejad, M. Hashemi, and S. Salehkaleybar, "cuPC: CUDA-based parallel PC algorithm for causal structure learning on GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 530–542, Mar. 2020.

[21] L. Nyman and M. Laakso, "Notes on the history of fork and join," *IEEE Ann. Hist. Comput.*, vol. 38, no. 3, pp. 84–87, Jul.-Sep. 2016.

[22] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur, "High performance MPI-2 one-sided communication over infiniband," in *Proc. IEEE Int. Symp. Cluster Comput. Grid*, 2004, pp. 531–538.

[23] B. P. Buckles and M. Lybanon, "Algorithm 515: Generation of a vector from the lexicographical index [G6]," *ACM Trans. Math. Softw.*, vol. 3, no. 2, pp. 180–182, 1977.

[24] I. A. Beinlich, H. J. Suermondt, R. M. Chavez, and G. F. Cooper, "The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks," in *Proc. Artif. Intell. Med.*, 1989, pp. 247–256.

[25] J. Binder, D. Koller, S. Russell, and K. Kanazawa, "Adaptive probabilistic networks with hidden variables," *Mach. Learn.*, vol. 29, no. 2, pp. 213–244, 1997.

[26] A. Onisko, "Probabilistic causal models in medicine: Application to diagnosis of liver disorders," in *Ph D. dissertation Inst. Biocybern. Biomed. Eng.*, Warsaw, Poland: Polish Academy Sci., 2003.

[27] S. Andreassen et al., "Computer-aided electromyography and expert systems," 1989.

[28] S. Andreassen, R. Hovorka, J. J. Benn, K. G. Olesen, and E. R. Carson, "A model-based approach to insulin adjustment," in *Proc. 3rd Conf. Artif. Intell. Med.*, vol. 44, M. Stefanelli, A. Hasman, M. Fieschi, & J. L. Talmon, Eds., Maastricht, The Netherlands, Jun. 24-27, 1991, pp. 239–248, doi: 10.1007/978-3-642-48650-0_19.

[29] C. S. Jensen and A. Kong, "Blocking Gibbs sampling for linkage analysis in large pedigrees with many loops," *Amer. J. Hum. Genet.*, vol. 65, no. 3, pp. 885–901, 1999.

[30] S. Acid and L. M. de Campos, "Searching for Bayesian network structures in the space of restricted acyclic partially directed graphs," *J. Artif. Intell. Res.*, vol. 18, pp. 445–490, 2003.

[31] J. Tian, "A branch-and-bound algorithm for mdl learning Bayesian networks," 2013, *arXiv:1301.3897*.

[32] J. W. Myers, K. B. Laskey, and T. S. Levitt, "Learning Bayesian networks from incomplete data with stochastic search algorithms," 2013, *arXiv:1301.6726*.

[33] T. Gao and D. Wei, "Parallel Bayesian network structure learning," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1685–1694.

[34] D. Colombo, M. H. Maathuis, M. Kalisch, and T. S. Richardson, "Learning high-dimensional directed acyclic graphs with latent and selection variables," *Ann. Statist.*, vol. 40, pp. 294–321, 2012.

[35] D. M. Chickering, "Optimal structure identification with greedy search," *J. Mach. Learn. Res.*, vol. 3, pp. 507–554, 2002.

[36] I. Tsamardinos, L. E. Brown, and C. F. Aliferis, "The max-min hill-climbing Bayesian network structure learning algorithm," *Mach. Learn.*, vol. 65, pp. 31–78, 2006.

[37] C. F. Aliferis, I. Tsamardinos, and A. Statnikov, "HITON: A novel Markov blanket algorithm for optimal variable selection," in *Proc. AMIA Annu. Symp.*, 2003, Art. no. 21.

[38] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs," *J. Mach. Learn. Res.*, vol. 19, no. 1, pp. 797–801, 2018.

[39] J. Jiang, Z. Wen, Z. ke Wang, B. He, and J. Chen, "Parallel and distributed structured SVM training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 5, pp. 1084–1096, May 2022.

[40] J. Jiang, Z. Wen, A. Mansoor, and A. Mian, "Fast parallel exact inference on Bayesian networks: Poster," 2022, *arXiv:2212.04241*.

[41] A. Srivastava, S. P. Chockalingam, and S. Aluru, "A parallel framework for constraint-based Bayesian network learning via Markov blanket discovery," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–15.

[42] C. Hagedorn and J. Huegle, "GPU-accelerated constraint-based causal structure learning for discrete data," in *Proc. SIAM Int. Conf. Data Mining*, 2021, pp. 37–45.

Jian Yang received the BEng degree in software engineering from the South China University of Technology, in 2022. He is currently working toward the PhD degree with the Hong Kong University of Science and Technology, Guangzhou (HKUST-GZ). His research interests include machine learning systems and graph mining.

Jiantong Jiang received the master's degree from Northeastern University, in China. She is currently working toward the PhD degree with the University of Western Australia. Before commencing her PhD study, she was a research assistant at School of Software Engineering, Zhejiang University, China. Her research interests include high-performance computing and automatic machine learning system.

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.

Zeyi Wen is an assistant professor with the Hong Kong University of Science and Technology, Guangzhou (HKUST-GZ). Before joining HKUST-GZ, he was a lecturer at The University of Western Australia, a research fellow in National University of Singapore and The University of Melbourne, after receiving his PhD degree from The University of Melbourne. Zeyi's areas of research include high-performance computing, machine learning systems and data mining.

Ajmal Mian is a professor of computer science with the University of Western Australia. He is the recipient of three prestigious national level fellowships from the Australian Research Council (ARC) including the recent Future Fellowship award 2022. He is a fellow of the International Association for Pattern Recognition, Distinguished Speaker of the Association for Computing Machinery and President of the Australian Pattern Recognition Society. He received the West Australian Early Career Scientist of the Year Award 2012 and the HBF Mid-Career Scientist of the Year Award 2022. He has secured research funding from the ARC, the National Health and Medical Research Council of Australia, US Department of Defence DARPA, and the Australian Department of Defence. He has published more than 260 scientific papers. He is a senior editor for IEEE TNNLS and associate editor IEEE TIP and the Pattern Recognition journal. His research interests include computer vision, artificial intelligence, deep learning, 3D point cloud analysis, facial recognition, human action measurement and video analysis.

Authorized licensed use limited to: University of Western Australia. Downloaded on February 28,2024 at 12:55:42 UTC from IEEE Xplore. Restrictions apply.