Fast Parallel Bayesian Network Structure Learning

Jiantong Jiang, Zeyi Wen* and Ajmal Mian
Department of Computer Science and Software Engineering, The University of Western Australia
jiantong.jiang@research.uwa.edu.au, zeyi.wen@uwa.edu.au, ajmal.mian@uwa.edu.au

Abstract—**Bayesian networks (BNs) are a widely used graphical model in machine learning for representing knowledge with uncertainty. The mainstream BN structure learning methods require performing a large number of conditional independence (CI) tests. The learning process is very time-consuming, especially for high-dimensional problems, which hinders the adoption of BNs to more applications. Existing works attempt to accelerate the learning process with parallelism, but face issues including load unbalancing, costly atomic operations and dominant parallel overhead. In this paper, we propose a fast solution named Fast-BNS on multi-core CPUs to enhance the efficiency of the BN structure learning. Fast-BNS is powered by a series of efficiency optimizations including (i) designing a dynamic work pool to monitor the processing of edges and to better schedule the workloads among threads, (ii) grouping the CI tests of the edges with the same endpoints to reduce the number of unnecessary CI tests, (iii) using a cache-friendly data storage to improve the memory efficiency, and (iv) generating the conditioning sets on-the-fly to avoid extra memory consumption. A comprehensive experimental study shows that the sequential version of Fast-BNS is up to 50 times faster than its counterpart, and the parallel version of Fast-BNS achieves 4.8 to 24.5 times speedup over the state-of-the-art multi-threaded solution. Moreover, Fast-BNS has a good scalability to the network size as well as sample size.**

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 dependency. The graphical nature of BNs makes them well-suited for representing knowledge with uncertainty and efficient reasoning. They have been successfully applied in a wide range of real-world applications [2]–[5]. With recent growing demand for interpretable machine learning models, BNs have attracted much research attention since they are inherently interpretable models [6], [7].

One crucial task of training BNs is structure learning, which aims to learn DAGs that are well matched the observed data. There are two common approaches for BN structure learning from data: 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 [8]. On the other hand, the constraint-based approaches perform a number of 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 is commonly used in high-dimensional problems [9].

A fundamental constraint-based algorithm is the PC (named after its authors Peter and Clark) algorithm [10] which starts from a complete undirected graph and removes edges in consecutive depths based on CI tests. PC-stable [11] solves the order-dependent issue in the original PC algorithm and produces less error. The PC-stable algorithm has been widely used in various applications [12], [13] and is implemented in different mainstream BN packages such as bnlearn [14], pcalg [15] and tetrad [16]. Furthermore, most constraint-based methods are improved versions of the PC-stable algorithm or proceed along similar lines of the PC-stable algorithm.

However, a key barrier that hinders the wider usage of the PC-stable algorithm is its long execution time for performing a large number of CI tests, especially for high-dimensional data sets. It is non-trivial to perform algorithmic improvements for the PC-stable algorithm [17]. Several research works have been conducted on the acceleration of the PC-stable algorithm on multi-core CPUs exploiting parallelization techniques [18]–[20]. The most common way is to parallelize the processing of different edges of the network inside each depth, which is an intuitive idea due to the order-independent property of the PC-stable algorithm. However, the direct edge-level parallelism is load unbalanced, because the workloads of CI tests for different edges is highly different. Another approach to parallelize the algorithm is by processing multiple samples inside each CI test, which is a finer granularity of parallelism. However, this approach requires many atomic operations and has a large parallel overhead, which decreases the efficiency.

To address the issues of workload balancing, atomic operations and large parallel overhead, we propose Fast-BNS, a fast BN structure learning solution. Fast-BNS is equipped with a dynamic work pool to contain the edges to be processed and their processing progresses with regard to the CI tests. The work pool is able to monitor the processing progresses of edges, terminating the completed edges in time to avoid unnecessary CI tests. Moreover, with the work pool, we can better schedule the work among threads to maintain workload balancing. Each thread always processes a group of CI tests that are required to be processed, and hence all the threads are active during this process. As the CI tests are independent and each CI test requires a reasonable amount of computation, Fast-BNS can be performed without atomic operations and can amortize the parallel overhead. To further enhance Fast-BNS, we develop a series of efficiency optimizations including (i) grouping the CI tests of the edges with the same endpoints

* Zeyi Wen is the corresponding author.

to reduce the number of CI tests, (ii) using a cache-friendly data storage to improve memory efficiency and (iii) generating the conditioning sets of the CI tests on-the-fly and in parallel to avoid extra memory consumption. To summarize, we make the following major contributions in this paper.

- We propose Fast-BNS to accelerate the BN structure learning on multi-core CPUs. Fast-BNS employs a dynamic work pool to monitor the processing of edges and schedule the workloads among threads, which solves the efficiency issues of load unbalancing. The granularity of parallelism used in Fast-BNS avoids atomic operations and amortizes the parallel overhead.
- We develop a series of novel techniques to further improve the efficiency of Fast-BNS. First, we propose to group the CI tests of the edges with the same endpoints together to reduce the number of unnecessary CI tests. Second, we employ a cache-friendly data storage to improve the memory efficiency. Lastly, we compute the conditioning sets on-the-fly and in parallel to reduce memory consumption.
- We conduct experiments to study the effectiveness of our proposed techniques. Experimental results show that the sequential version of Fast-BNS outperforms the existing work bnlearn [14] and tetrad [16] by up to 50 times. When compared with the multi-threaded implementation in bnlearn [18], the parallel version of Fast-BNS is 4.8 to 24.5 times faster. Finally, we show that Fast-BNS has good scalability to the network size and sample size.

II. 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 of using BNs 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 [21]–[27] seek the best directed acyclic graph (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 [8]. Hence, many score-based approaches employ heuristic methods, like greedy search or simulated annealing, in an attempt to reduce the search space. Such approaches can easily get trapped in local optima [17]. The optimization techniques in this paper focus on the constraint-based approaches which tend to scale better to high-dimensional data.

Constraint-based approaches [10], [11], [28]–[32] 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. Most of the constraint-based algorithms proceed along similar lines as the work of the PC-stable algorithm [10], [11]. Unlike score-based approaches, it is generally non-trivial to perform algorithmic improvements for constraint-based approaches using general-purpose optimization theory. This paper mainly focuses on improving the efficiency of the PC-stable algorithm using parallel techniques.

There are some well-known open-source BN libraries which contain the implementation of the PC-stable algorithm, such as bnlearn [14], pcalg [15] and tetrad [16]. Meanwhile, since the recent parallel computing platforms, such as multi-core CPUs and GPUs, have emerged to efficiently address various computational machine learning problems [33], [34], there are several research works that focus on the acceleration of the PC-stable algorithm using parallel techniques [18]–[20]. 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.

III. 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 $\mathcal{V} = \{V_0, V_1, ..., V_{n-1}\}$ via a directed acyclic graph (DAG). Typically, one variable corresponds to one feature in the machine learning problems. We use $G = (\mathcal{V}, \mathcal{E})$ to denote the DAG. In a DAG G, each node in $\mathcal{V}$ 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 distributionthat describes the probabilities of possible values of this variable given its possible parent configurations. The joint probability of variables $\mathcal{V}$ 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:

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

where n is the number of variables, $P(V_0, V_1, ..., V_{n-1})$ 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 $\mathcal{D} = \{c_0, c_1, ..., c_{m-1}\}$ denote a data set 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 statistic [10] 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 satisfy $V_i = x$, $V_j = y$ 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 [35]. 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 [36], taking more than 90% of the total execution time in many problems. In Section IV, 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 the graph G to a complete undirected graph over the node set $\mathcal{V}$ (Line 3), and remove some of the edges by performing a number of CI tests in consecutive depths (Lines 5 to 20). Specifically, at each depth d, the algorithm iteratively records

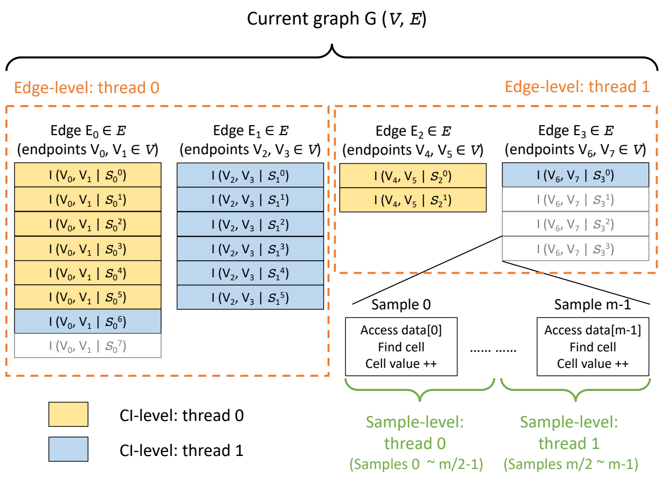
Algorithm 1: The first step of the PC-stable algorithm.

```
1  Input: Node set $\mathcal{V}$
2  Output: Graph $G$, $SepSet$
3  Form the complete undirected graph $G$ over $\mathcal{V}$
4  Let depth $d = 0$
5  repeat
6      for any node $V_i$ in $G$ do
7          Let $a(V_i) = adj(G, V_i)$
8      end for
9      for any edge $(V_i, V_j)$ in $G$ do
10         repeat
11             Choose a new $\mathcal{S} \subseteq a(V_i)\backslash\{V_j\}$ with $|\mathcal{S}| = d$
12             Perform CI test $I(V_i, V_j|\mathcal{S})$
13             if hypothesis $I(V_i, V_j|\mathcal{S})$ holds then
14                 Remove $(V_i, V_j)$ from $G$
15                 Store $\mathcal{S}$ in $SepSet(V_i, V_j)$
16             end if
17         until $(V_i, V_j)$ is removed or all $\mathcal{S}$ are considered
18     end for
19     Let $d = d + 1$
20 until all pairs of $(V_i, V_j)$ in $G$ satisfy $|a(V_i)\backslash\{V_j\}| < d$
```

the current adjacency sets of all the nodes (Lines 6 to 8), where $adj(G, V_i)$ denotes the adjacent nodes of V_i in G. 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 9 to 12). 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 13 to 15). $SepSet(V_i, V_j)$ 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 19) 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 20.

IV. Parallel BN Structure Learning

This section elaborates the technical details of our proposed Fast-BNS. First, we identify two granularities of parallelism, including edge-level parallelism and sample-level parallelism. We find some shortcomings of accelerating the PC-stable algorithm using the above two granularities of parallelism, including (i) load unbalancing between threads, (ii) many atomic operations and (iii) high parallel overhead. To remedy the shortcomings, we propose Fast-BNS which exploits a CI-level parallelism, where multiple groups of CI tests from different edges are performed in parallel. Fast-BNS takes



Current graph G ($\mathcal{V}$, $\mathcal{E}$)
Edge-level: thread 0
Edge-level: thread 1
Edge $E_0 \in \mathcal{E}$ (endpoints $V_0, V_1 \in \mathcal{V}$)
Edge $E_1 \in \mathcal{E}$ (endpoints $V_2, V_3 \in \mathcal{V}$)
Edge $E_2 \in \mathcal{E}$ (endpoints $V_4, V_5 \in \mathcal{V}$)
Edge $E_3 \in \mathcal{E}$ (endpoints $V_6, V_7 \in \mathcal{V}$)
$I(V_0, V_1 \mid \mathcal{S}_0^0)$
$I(V_0, V_1 \mid \mathcal{S}_0^1)$
$I(V_0, V_1 \mid \mathcal{S}_0^2)$
$I(V_0, V_1 \mid \mathcal{S}_0^3)$
$I(V_0, V_1 \mid \mathcal{S}_0^4)$
$I(V_0, V_1 \mid \mathcal{S}_0^5)$
$I(V_0, V_1 \mid \mathcal{S}_0^6)$
$I(V_0, V_1 \mid \mathcal{S}_0^7)$
$I(V_2, V_3 \mid \mathcal{S}_1^0)$
$I(V_2, V_3 \mid \mathcal{S}_1^1)$
$I(V_2, V_3 \mid \mathcal{S}_1^2)$
$I(V_2, V_3 \mid \mathcal{S}_1^3)$
$I(V_2, V_3 \mid \mathcal{S}_1^4)$
$I(V_2, V_3 \mid \mathcal{S}_1^5)$
$I(V_4, V_5 \mid \mathcal{S}_2^0)$
$I(V_4, V_5 \mid \mathcal{S}_2^1)$
$I(V_6, V_7 \mid \mathcal{S}_3^0)$
$I(V_6, V_7 \mid \mathcal{S}_3^1)$
$I(V_6, V_7 \mid \mathcal{S}_3^2)$
$I(V_6, V_7 \mid \mathcal{S}_3^3)$
Sample 0
Sample m-1
Access data[0] Find cell Cell value ++
....
Access data[m-1] Find cell Cell value ++
Sample-level: thread 0 (Samples 0 ~ m/2-1)
Sample-level: thread 1 (Samples m/2 ~ m-1)
CI-level: thread 0
CI-level: thread 1

Fig. 1: Three different granularities of parallelism: edge-level parallelism, sample-level parallelism and CI-level parallelism.

advantage of a dynamic work pool that contains the edges required to be processed and their processing progress to ensure load balancing among threads. The CI-level parallelism also avoids atomic operations and leads to a reasonable amount of workloads to amortize the overhead of parallel computing. To further enhance the efficiency of Fast-BNS, we propose 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, and (iii) generating the conditioning sets of the CI tests on-the-fly and in parallel to avoid extra memory consumption.

A. Edge-Level and Sample-Level Parallelism

Here, we first describe two schemes using two granularities of parallelism: edge-level parallelism and sample-level parallelism. Then, we discuss the limitations of these schemes.

Edge-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. In each depth d, it parallelizes the for-loop in Line 9 of Algorithm 1, dedicating $\frac{|\mathcal{E}_d|}{t}$ edges to each thread, where t represents the number of threads and $|\mathcal{E}_d|$ represents the number of edges to be processed in the depth d. A simple example is shown in Figure 1. The current graph contains four edges, and thus for the case of $t = 2$, each of the two threads is responsible for processing two edges. Specifically, thread 0 is dedicated to edges E_0 and E_1, while E_2 and E_3 are assigned to thread 1. This is an intuitive idea because the order-independent property of PC-stable makes it suitable for parallelizing at each depth. In other words, an edge deletion does not effect the processing of other edges at the same depth, and thus parallelization can be applied.

Sample-Level Parallelism. Another scheme is to parallelize among samples inside each CI test, which is a fine-grained parallelism. Since there are a large number of CI tests in the PC-stable algorithm and they take most of the execution time, the key reason behind this scheme is to optimize each of the CI tests. To explain the sample-level parallelism, we decompose the processing of CI tests into three steps: (i) generating the contingency table, (ii) computing the marginal tables, and (iii) computing G^2 statistics and determining the dependency hypothesis. Among them, the most important step is to generate the contingency table. In this step, the contingency table is generated by traversing the whole data set. Specifically, for the CI test $I(V_i, V_j | \{V_k\})$, it accesses all samples in the data set and finds one cell in the contingency table for each sample by getting the values of V_i, V_j and V_k of the sample, and finally increments one to this cell. Therefore, the sample-level parallelism is to parallelize the traversing of the whole data set, dedicating $\frac{m}{t}$ samples to each thread, where m represents the number of samples in the data set. For the example in Figure 1, for the case of $t = 2$, each thread would process $\frac{m}{2}$ samples for every CI test.

Limitations of Edge-Level Parallelism. The coarse-grained edge-level parallelism is load unbalanced, because the workloads of CI tests for different edges is highly different due to the following reasons.

- Firstly, different nodes in the network may have different number of adjacent nodes. For example, if node V_i has two adjacent nodes while V_k has ten adjacent nodes, then there is $\binom{2}{2} = 1$ conditioning set when processing edge $V_i - V_j$ in the depth $d = 2$, while there are $\binom{10}{2} = 45$ possible conditioning sets for edge $V_k - V_j$ in the same depth. We can see from this example that the number of possible conditioning sets can be quite different. As shown in Figure 1, edges E_0 and E_1 assigned to thread 0 have 14 possible CI tests in total, while E_2 and E_3 assigned to thread 1 have only 6 possible CI tests.
- Secondly, when a hypothesis $I(V_i, V_j | \mathcal{S})$ holds, the edge between V_i and V_j is removed in advance. Then the CI tests conditioning on the remaining conditioning sets are unnecessary. Let us take the edge E_3 in Figure 1 as an example. If V_6 and V_7 are conditionally independent given the first conditioning set tested (i.e., $\mathcal{S}_3^0$), the edge E_3 is removed in advance and the three CI tests of V_6 and V_7 left are unnecessary to perform, as there is no edge between V_6 and V_7 now. Such unnecessary CI tests are marked in gray in Figure 1. This example shows that we do not know in advance as to how many CI tests are required for each edge, since any edge has the possibility of being removed before finishing its CI tests conditioning on all its possible conditioning sets.

Limitations of Sample-Level Parallelism: The sample-level parallelism is fine-grained, but fails to get good speedups due to the following two main reasons.

- Firstly, since each sample contributes to the generation of the contingency table, sample-level parallelism may result in a race condition when updating the cells in the contingency table. Therefore, we need many atomic operations to guarantee the correctness of the execution. However, atomic operations are expensive. An alternative way to perform the contingency table generation is to create a local contingency table for each thread. For a specific CI test, each thread would then be responsible for updating its local table according to its own data. Then the local tables from all threads are combined to generate the final contingency table. However, this alternative approach requires much more memory and introduces non-negligible costs of frequent synchronization and communication among threads.

- Secondly, although generating the contingency table is the most time-consuming step, it is mainly due to the large number of such operations, while the workload of each operation is relatively small. We need a larger amount of workload for each thread in order to amortize the overhead of parallel computing (e.g., thread creation).

B. CI-Level Parallelism

To overcome the limitations of the edge-level and sample-level parallelism, our proposed Fast-BNS employs a CI-level parallelism, which has a parallelism granularity between edge-level and sample-level. In each depth, CI tests from different edges are performed in parallel. To achieve CI-level parallelism, our key idea is to employ a dynamic work pool 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 we can fetch multiple edges required to be processed from the work pool, find the next groups of CI tests of the edges through their processing progresses, and execute them in parallel.

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 assigned to the threads. Each thread would then be responsible for processing a group of CI tests of the assigned edge, where the number of CI tests is introduced as gs ($gs \geq 1$). When the gs CI tests are finished, 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, it means that the processing of the edge is completed, 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 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 the 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 Figure 1, 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.

TABLE I: Comparison between edge-level parallelism, sample-level parallelism and the proposed CI-level parallelism.

Granularity of parallelism	Load balance	Avoid atomic operations	Reasonable workloads
Edge-level parallelism	✗	✓	✓
Sample-level parallelism	✓	✗	✗
CI-level parallelism	✓	✓	✓

Compared with the edge-level parallelism, the proposed CI-level parallelism has a finer granularity. The workloads of CI tests can be evenly distributed to each thread with the design of the dynamic work pool, which solves the issue of load unbalancing. On the other hand, CI-level parallelism is coarser than the sample-level parallelism. Since one CI test is not distributed to multiple threads, each thread holds a complete contingency table and the race condition does not occur. Therefore, the CI-level parallelism avoids expensive atomic operations and also has a reasonable amount of workloads to amortize the parallel overhead. Table I summarizes the key differences between edge-level parallelism, sample-level parallelism and the CI-level parallelism.

The gs in the CI-level parallelism is a trade-off between the number of CI tests and memory accesses. In the parallel region, each thread processes 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 data set. 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 the CI-level 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 $\mathcal{V}$, 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. Further Enhancing Fast-BNS

We find three issues in existing implementations. 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. In this section, we aim to tackle these three issues to further improve the overall efficiency.

Grouping CI tests of the 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, 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 does not perform the CI tests conditioning on the variables in $adj(G, V_j)\backslash\{V_i\}$.

Using a cache-friendly data storage: As discussed in Section IV-A, 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 data set and obtain the values of X, Y, Z_1 and Z_2 for all the samples. For a naive two-dimensional data set 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 one feature and each column to represent one sample, which is a cache-friendly data storage for the data. 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 only has four cache misses at the beginning and the rest can be served from four cache lines.

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 $\mathcal{S} = \{\mathcal{S}_0, \mathcal{S}_1, ..., \mathcal{S}_{\binom{p}{q}-1}\}$ 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 implements a combination function to generate $\mathcal{S}$ in lexicographical order [37]. Given p, q and r, the combination function of Fast-BNS 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 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.

D. Performance Analysis

As we have discussed in Sections IV-A and IV-C, we have optimizations for Fast-BNS. Here, we analyze the theoretical speedups provided by these optimizations. 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. Moreover, we also use the strategy to generate conditioning sets on-the-fly. However, conditioning set generation mainly aims to reduce memory consumption, and hence we omit the speedup it provides in this section.

1) Using CI-level parallelism with the design of the dynamic work pool: 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 $\binom{a_i^1}{d} + \binom{a_i^2}{d}$ 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 practical, 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 have to wait for the completion of the slowest thread p. In the other words, 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

$$S_{CI} = \frac{T_1}{T_2}$$

2) Grouping CI tests of the edges with the same endpoints: This optimization provides the 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 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 the speedup by reducing the ratio of cache misses. The memory accesses of PC-stable mainly come from the accesses to the data set when computing the contingency table. For the CI test $I(X, Y | \{Z_1, ..., Z_d\})$ in depth d, we need to access the values of X, Y, Z_1, ..., Z_d of the m samples in the data set, where each of the values is 4 bytes in memory. Suppose that the cache line size is B bytes. Firstly 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

$$T_3 = T_{DRAM}(d+2)\frac{B}{4}$$

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

$$T_4 = T_{DRAM}(d+2) + T_{cache}(d+2)(\frac{B}{4} - 1)$$

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

$$S_{cache} = \frac{T_3}{T_4}$$

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, and we assume that the mean degree is 10. Hence, every a_i^1 and a_i^2 in Equations (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: $S_{CI} = 3.87$, $S_{grouping} = 1.43$, $S_{cache} = 5.57$, 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.

V. Experimental Evaluation

In this section, we conducted experiments on to evaluate the performance of our proposed techniques and compared the results to existing methods.

A. Experimental Setup

We implemented Fast-BNS using OpenMP in C++ for Bayesian Network structure learning and compared its performance to the existing methods. Specifically, we compared Fast-BNS with sequential implementations of the PC-stable algorithm in three different open-source packages including bnlearn [14], pcalg [15] and tetrad [16]. We also compared Fast-BNS with the recent multi-threaded implementations in bnlearn [18] and parallel-PC [20]. Bnlearn, pcalg and parallel-PC are all R packages, while tetrad is implemented in Java. There are other parallel work for PC-stable, such as the work [19]. However, the algorithm in [19] is not open-source,

TABLE II: BNs from which data sets used are generated.

Data set	# of nodes	# of edges	max # of samples
Alarm [38]	37	46	15000
Insurance [39]	27	52	15000
Hepar2 [40]	70	123	15000
Munin1 [41]	186	273	15000
Diabetes [42]	413	602	5000
Link [43]	724	1125	5000
Munin2 [41]	1003	1244	5000
Munin3 [41]	1041	1306	5000

and its 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 over its sequential counterpart. Therefore, we did not compare Fast-BNS with [19]. All the experiments were conducted on a Linux machine with two 26-core 2GHz Intel Xeon Platinum 8167M CPUs and 768GB main memory.

Data sets used in our experiments were obtained from eight benchmark BNs of different sizes listed in Table II, where the last four data sets 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 of the networks. Besides, more data sets are 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 is exactly the same as the other PC-stable algorithm implementations, because Fast-BNS is an accelerated implementation 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, we compared the execution time of both sequential and parallel implementations of Fast-BNS with the existing implementations on the eight data sets with 5000 samples. Specifically, we compared the sequential version of Fast-BNS (i.e., Fast-BNS-seq) with the PC-stable implementations in bnlearn (i.e., bnlearn-seq) [14], pcalg [15] and tetrad [16] packages; we also compared the parallel version of Fast-BNS (i.e., Fast-BNS-par) with the multi-threaded implementation in bnlearn (i.e., bnlearn-par) [18] and and parallel-PC [20]. The gs of Fast-BNS was set to 1 for all the experiments here. For comparing the parallel implementations, we varied the number of OpenMP threads 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 can be seen from the "Speedup" columns of the table, the sequential implementation of our proposed Fast-BNS often achieves two to three orders of magnitude speedup over tetrad and pcalg, and can be 1.4 to 7.2 times faster than the sequential version of bnlearn. The speedups of Fast-BNS are mainly due to the careful optimizations, including of grouping CI tests of the edges with the same endpoints, using a cache-friendly data storage and generating conditioning sets on-the-fly, as we discussed in Section IV-C. These general optimizations can be applied to both sequential and parallel implementations. When comparing the parallel implementations, Fast-BNS-par is often much faster than parallel-PC, and can run 4.8 to 24.5 times faster than bnlearn-par. It is worth noting that for some small data sets, such as *Alarm* and *Insurance*, bnlearn failed to get improvements by the multi-threaded techniques, and thus the same results were reported for its sequential and parallel implementations. Another observation is that Fast-BNS always achieves its shortest execution time when $t = 32$. Moreover, the execution time of the sequential version of Fast-BNS can be reduced by more than 85% by using the parallel computing techniques. The experiment on the *Link* data set 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 1.2 hours using the proposed Fast-BNS.

To further investigate why Fast-BNS is faster, we used perf Linux profiler to obtain the detailed measurements for Fast-BNS-par, Fast-BNS-seq and bnlearn-par. The results on *Hepars* and *Munin1* are shown in Table IV. We can observe that the parallel version of Fast-BNS increases CPU utilization and FLOPS. Moreover, compared with bnlearn, both the sequential and parallel implementations of Fast-BNS have fewer accesses to the L1 cache and last level (LL) cache, and significantly decreases the rate of cache misses.

C. Studies on Different Granularities

Compared with the schemes of the edge-level and sample-level parallelism, the CI-level parallelism employed in Fast-BNS solves the efficiency issues of load unbalancing, atomic operations and large parallel overhead (cf. Sections IV-A and IV-B). To investigate the performance of different parallelism granularities, we implemented another two parallel versions using the schemes of edge-level and sample-level parallelism, and compared it with Fast-BNS that employs the CI-level parallelism. All these parallel versions are based on the optimized sequential version of Fast-BNS.

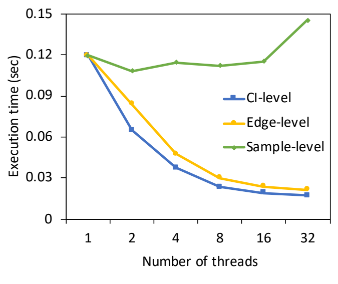
Figure 2 illustrates the execution time of the three schemes using different granularities of parallelism with different number of threads. We observe that the CI-level parallelism always leads to the shortest execution time under different number of threads, indicating the effectiveness of the optimizations used in the CI-level parallelism. Overall, the sample-level parallelism is the worst due to the efficiency issues of many expensive atomic operations and large parallelization overhead. Moreover, the execution time of the edge-level parallelism can be reduced by more than 20% using the CI-level parallelism which solves its load unbalancing issue. Learning larger-scale BNs may encounter more issues of load unbalancing, and Fast-BNS can take more advantage of the load balancing optimization in the CI-level parallelism. On *Diabetes* and *Link*, the improvement is over 3 times.

TABLE III: Execution time comparison of Fast-BNS with other implementations under both sequential and parallel setting. Speedup of Fast-BNS over each compared method is also reported.

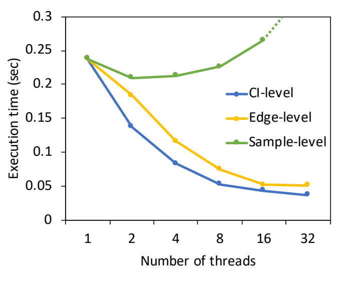
Data set	Sequential implementation							Parallel implementation				
	Execution time (sec)				Speedup			Execution time (sec)			Speedup	
	bnlearn	tetrad	pcalg	Fast-BNS	bnlearn	tetrad	pcalg	bnlearn	parallel-PC	Fast-BNS	bnlearn	parallel-PC
Alarm	0.42	5.38	53.8	0.12	3.5	45.1	450	0.42	15.4	0.017	24.5	890
Insurance	0.34	13.08	71.9	0.24	1.4	55	302	0.34	25.4	0.037	9.2	687
Hepar2	4.03	37.7	208.9	1.57	2.8	24	133	2.82	158	0.19	15.2	852
Munin1	111	770	2160	15.5	7.2	49.8	140	16.5	162	1.78	9.3	91.3
Diabetes	113k	$>$ 2 days		23.3k	4.9	$>$ 7.4		7640	54k	1203	6.4	44.9
Link	$>$ 2 days			62.9k	$>$ 2.7			49.4k	$>$ 2 days	4349	11.4	$>$ 39.7
Munin2	27.9k	$>$ 2 days		3496	8.0	$>$ 49.4		2734	$>$ 2 days	293	9.3	$>$ 590
Munin3	38.7k	$>$ 2 days		8081	4.8	$>$ 21.4		3621	$>$ 2 days	751	4.8	$>$ 230

TABLE IV: Detailed comparison of the parallel and sequential versions of Fast-BNS with the parallel version of bnlearn. "-seq" and "-par" represent sequential and parallel implementation, respectively.

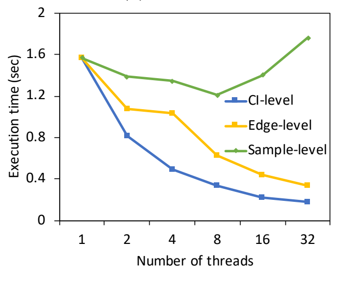
Hepar2	L1-cache accesses	L1-cache misses (rate)	LL-cache accesses	LL-cache misses (rate)	FLOPS	CPU utilization
Fast-BNS-par	4.5×10^9	7.9×10^7 (1.78%)	1.6×10^6	8.1×10^4 (5.1%)	1.4×10^9	12.7
Fast-BNS-seq	4.1×10^9	7.2×10^7 (1.73%)	2.5×10^5	1.5×10^4 (6.0%)	2.3×10^8	1
bnlearn-par	1.5×10^{10}	4.7×10^8 (3.17%)	4.2×10^7	1.7×10^7 (39.9%)	7.0×10^7	3.7
Munin1	**L1-cache accesses**	**L1-cache misses (rate)**	**LL-cache accesses**	**LL-cache misses (rate)**	**FLOPS**	**CPU utilization**
Fast-BNS-par	3.8×10^{10}	8.7×10^8 (2.28%)	8.9×10^6	1.8×10^5 (2.03%)	2.3×10^9	13.2
Fast-BNS-seq	3.8×10^{10}	8.8×10^8 (2.28%)	8.6×10^6	9.3×10^4 (1.08%)	2.7×10^8	1
bnlearn-par	1.0×10^{11}	3.0×10^9 (2.92%)	1.7×10^8	8.2×10^7 (47.1%)	2.4×10^8	8.7



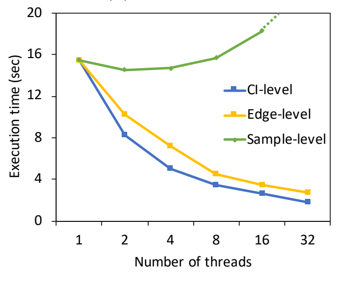
(a) Alarm



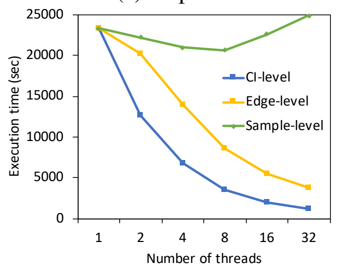
(b) Insurance



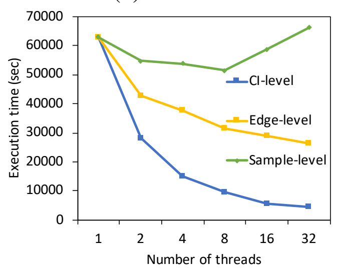
(c) Hepar2



(d) Munin1



(e) Diabetes



(f) Link

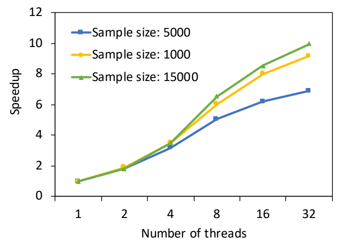
Fig. 2: Execution time of parallel implementations using three different granularities of parallelism: CI-level parallelism, edge-level parallelism and sample-level parallelism.

D. Sensitivity Studies

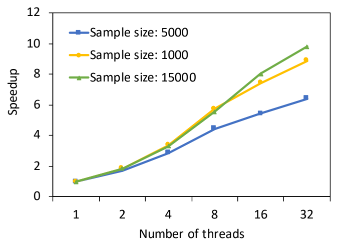
To better understand Fast-BNS, we performed sensitivity studies on three key parameters: sample size, group size and network size. A series of experiments were carried out by changing the sample size and group size. Experiments were also conducted on data sets with different network sizes.

Varying sample size: We conducted experiments on *Alarm*, *Insurance*, *Hepar2* and *Munin1* networks to investigate the scalability of the proposed Fast-BNS to the sample size. We used different data sets of 5,000, 10,000 and 15,000 samples, and compared the execution time for the different sample sizes. Figure 3 shows the speedups of the parallel implementation of Fast-BNS over the sequential implementation of Fast-BNS under different sample sizes. We can observe a smooth improvement in speedups for all the sample sizes, indicating good scalability of the proposed techniques to the sample size. A large sample size often gets a slightly higher speedup because in this case, each CI test has a larger amount of workload which can better amortize the parallel overhead.

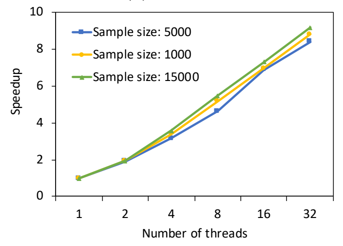
Varying group size: We examined the effect of gs of Fast-BNS on the number of CI tests and execution time. We tested on *Alarm*, *Insurance*, *Hepar2* and *Munin1* networks with 10,000 samples, which are also used in the previous experiments. The results are shown in Figure 4. The bars in the figure illustrate the execution time under different group sizes for the four problems, and the lines illustrate the proportion of the CI tests increased by the group size compared to the case of $gs = 1$. We can observe that the number of CI tests increases with the increase in group size. Therefore, although a larger group size can reduce more memory accesses, the final execution time may increase for the larger group sizes. The gs is a trade-off between the number of CI tests and memory accesses. Our observation is that the proportion of



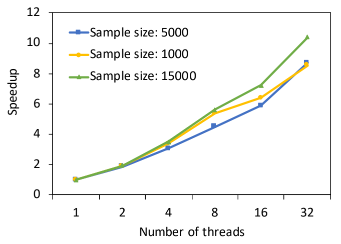
(a) Alarm



(b) Insurance

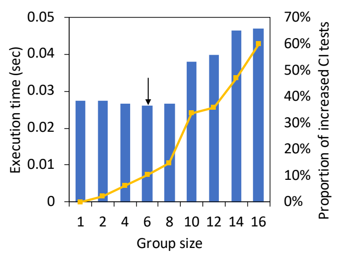


(c) Hepar2

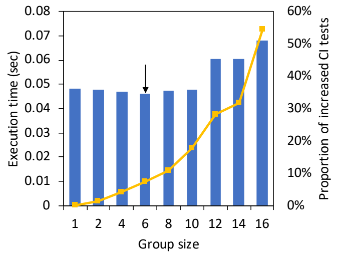


(d) Munin1

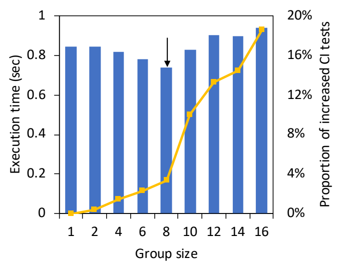
Fig. 3: Studies on speedup of Fast-BNS-par over Fast-BNS-seq under different sample sizes.



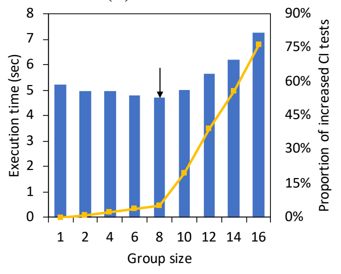
(a) Alarm



(b) Insurance



(c) Hepar2



(d) Munin1

Fig. 4: Effect of group sizes on execution times and the number of increased CI tests.

the increased CI tests is not too high (e.g., less than 10%) when the group size is no more than 8, while the number of CI tests often increases more rapidly when the group size is greater than 8. For example, the proportion of the increased CI tests is about 5% for *Munin1* when $gs = 8$, while the proportion increases to about 20% when $gs = 10$. As a result, the shortest execution time is often achieved when $gs \leq 8$ and it depends on the specific problem. The downward arrows in Figure 4 mark the gs that achieves the shortest execution time. We can observe that *Alarm* and *Insurance* achieve the shortest execution time when $gs = 6$, and *Hepar2* and *Munin1* achieve the shortest execution time when $gs = 8$. Note that the experimental results of Fast-BNS in Section V-B are with $gs = 1$, and thus the execution time can be further reduced by about 10% with careful parameter tuning.

Different network sizes: Figure 5 shows the speedups of the parallel implementation of Fast-BNS over its sequential implementation on the data sets with 5000 samples. The six BNs tested in the experiments are of different network sizes as shown in Table II. We can observe that Fast-BNS can achieve high speedups for large-scale networks, indicating good scalability of the proposed techniques to large networks. For example, Fast-BNS achieves 19.3 times speedup on the *Diabetes* network, which contains 413 nodes and 602 edges. For small-scale networks, the speedups of parallel implementation is relatively smaller, because they already require short execution time for structure learning (e.g., less than 1 second for *Alarm* and *Insurance*) and the parallelization overhead of these small-scale networks accounts for a large proportion. Therefore, with the help of the proposed general optimizations discussed in Section IV-C, our sequential implementation is sufficient for such small-scale networks with a relatively small number of nodes and edges.

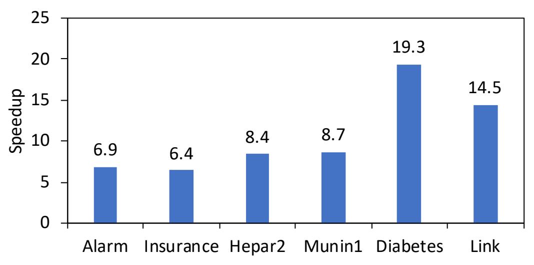


Fig. 5: Studies on speedup of Fast-BNS-par over Fast-BNS-seq under different network sizes.

VI. Conclusion

In this paper, we have proposed a parallel PC-stable algorithm namely Fast-BNS for learning Bayesian Network (BN) structure. The challenges of developing a fast solution for BN structure learning include addressing load unbalancing issues, reducing atomic operations and amortizing parallel overhead. To tackle these challenges, Fast-BNS exploits the CI-level parallelism, which avoids the expensive atomic operations and has suitable amount of workloads to amortize the parallel overhead. A dynamic work pool is designed to monitor the processing progress of edges and to schedule the workloads of threads, so as to balance the workload among threads. Fast-BNS also leverages a series of techniques to reduce the unnecessary CI tests and improve the memory efficiency. We have conducted extensive experiments to test the effectiveness of Fast-BNS. Experimental results have shown that the sequential version of Fast-BNS is up to 50 times faster than the existing solutions. When compared with the parallel counterpart, the parallel version of Fast-BNS is 9.2 to 24.5 times faster. Furthermore, Fast-BNS has demonstrated good scalability to network size and sample size.

Acknowledgment

Professor Ajmal Mian is the recipient of an Australian Research Council Future Fellowship Award (project number FT210100268) funded by the Australian Government. This research is also supported by Oracle for Research, Australia.

References

[1] J. Pearl, "Probabilistic reasoning in intelligent systems: networks of plausible inference," 1988.

[2] E. Kyrimi, S. McLachlan, K. Dube, M. R. Neves, A. Fahmi, and N. Fenton, "A comprehensive scoping review of Bayesian networks in healthcare: Past, present and future," *Artificial Intelligence in Medicine*, p. 102108, 2021.

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

[4] B. G. Marcot and T. D. Penman, "Advances in Bayesian network modelling: Integration of modelling technologies," *Environmental modelling & Software*, vol. 111, pp. 386–393, 2019.

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

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

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

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

[9] M. Kalisch and P. Bühlman, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm." *Journal of Machine Learning Research*, vol. 8, no. 3, 2007.

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

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

[12] X. Zhang, X.-M. Zhao, K. He, L. Lu, Y. Cao, J. Liu, J.-K. Hao, Z.-P. Liu, and L. Chen, "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.

[13] 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.

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

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

[16] J. D. Ramsey, K. Zhang, M. Glymour, R. S. Romero, B. Huang, I. Ebert-Uphoff, S. Samarasinghe, E. A. Barnes, and C. Glymour, "Tetrad—a toolbox for causal discovery," in *International Workshop on Climate Informatics*, 2018.

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

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

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

[20] 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," *Transactions on Computational Biology and Bioinformatics*, vol. 16, no. 5, pp. 1483–1495, 2016.

[21] D. M. Chickering, "Optimal structure identification with greedy search," *Journal of Machine Learning Research*, vol. 3, no. Nov, pp. 507–554, 2002.

[22] D. Chickering, D. Geiger, and D. Heckerman, "Learning Bayesian networks: Search methods and experimental results," in *Proceedings of Fifth Conference on Artificial Intelligence and Statistics*, 1995, pp. 112–128.

[23] S. Acid and L. M. de Campos, "Searching for Bayesian network structures in the space of restricted acyclic partially directed graphs," *Journal of Artificial Intelligence Research*, vol. 18, pp. 445–490, 2003.

[24] P. Larranaga, M. Poza, Y. Yurramendi, R. H. Murga, and C. M. H. Kuijpers, "Structure learning of Bayesian networks by genetic algorithms: A performance analysis of control parameters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 9, pp. 912–926, 1996.

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

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

[27] R. Blanco, I. Inza, and P. Larranaga, "Learning Bayesian networks in the space of structures by estimation of distribution algorithms," *International Journal of Intelligent Systems*, vol. 18, no. 2, pp. 205–220, 2003.

[28] D. Colombo, M. H. Maathuis, M. Kalisch, and T. S. Richardson, "Learning high-dimensional directed acyclic graphs with latent and selection variables," *The Annals of Statistics*, pp. 294–321, 2012.

[29] T. S. Richardson, "A discovery algorithm for directed cyclic graphs," *arXiv preprint arXiv:1302.3599*, 2013.

[30] N. Harris and M. Drton, "Pc algorithm for nonparanormal graphical models." *Journal of Machine Learning Research*, vol. 14, no. 11, 2013.

[31] D. Margaritis, "Learning Bayesian network model structure from data," Carnegie-Mellon Univ Pittsburgh Pa School of Computer Science, Tech. Rep., 2003.

[32] S. Yaramakala and D. Margaritis, "Speculative markov blanket discovery for optimal feature selection," in *International Conference on Data Mining*. IEEE, 2005, pp. 4–pp.

[33] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "Thundersvm: A fast SVM library on GPUs and CPUs," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 797–801, 2018.

[34] J. Jiang, Z. Wen, Z. ke Wang, B. He, and J. Chen, "Parallel and distributed structured SVM training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 1084–1096, 2022.

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

[36] B. Zarebavani, F. Jafarinejad, M. Hashemi, and S. Salehkaleybar, "cupc: Cuda-based parallel PC algorithm for causal structure learning on gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 530–542, 2019.

[37] B. P. Buckles and M. Lybanon, "Algorithm 515: Generation of a vector from the lexicographical index [g6]," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 2, pp. 180–182, 1977.

[38] 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 *AIME 89*. Springer, 1989, pp. 247–256.

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

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

[41] S. Andreassen, F. Jensen, S. Andersen, B. Falck, U. Kjrul, M. Woldbye, A. Srensen, A. Rosenfalck, and F. Jensen, "Computer-aided electromyography and expert systems," 1989.

[42] S. Andreassen, R. Hovorka, J. Benn, K. G. Olesen, and E. R. Carson, "A model-based approach to insulin adjustment," in *AIME 91*. Springer, 1991, pp. 239–248.

[43] C. S. Jensen and A. Kong, "Blocking gibbs sampling for linkage analysis in large pedigrees with many loops," *The American Journal of Human Genetics*, vol. 65, no. 3, pp. 885–901, 1999.