cuPC: CUDA-based Parallel PC Algorithm for Causal Structure Learning on GPU

Behrooz Zarebavani, Foad Jafarinejad, Matin Hashemi, and Saber Salehkaleybar

Abstract—The main goal in many fields in the empirical sciences is to discover causal relationships among a set of variables from observational data. PC algorithm is one of the promising solutions to learn underlying causal structure by performing a number of conditional independence tests. In this paper, we propose a novel GPU-based parallel algorithm, called cuPC, to execute an order-independent version of PC. The proposed solution has two variants, cuPC-E and cuPC-S, which parallelize PC in two different ways for multivariate normal distribution. Experimental results show the scalability of the proposed algorithms with respect to the number of variables, the number of samples, and different graph densities. For instance, in one of the most challenging datasets, the runtime is reduced from more than 11 hours to about 4 seconds. On average, cuPC-E and cuPC-S achieve 500 X and 1300 X speedup, respectively, compared to serial implementation on CPU. The source code of cuPC is available online [1].

Index Terms—Bayesian Networks, Causal Discovery, CUDA, GPU, Machine Learning, Parallel Processing, PC Algorithm.

1 Introduction

Learning causal structures is one of the main problems in empirical sciences. For instance, we need to understand the impact of a medical treatment on a disease or recover causal relations between genes in gene regulatory networks (GRN) [2]. By discovering such causal relations, one will be able to predict the impact of different actions. Causal relations can be inferred by controlled randomized experiments. However, in many cases, it is not possible to perform the required experiments due to technical or ethical reasons. In such cases, causal relations need to be learned merely from observational data [3], [4].

Causal Bayesian network is one of the models which has been widely considered to explain the data-generating mechanism. In this model, causal relations among variables are represented by a directed acyclic graph (DAG) where there is a direct edge from variable V_i to variable V_j if V_i is a direct cause of V_j. The task of causal structure learning is to learn all DAGs that are compatible with the observed data. Under some assumptions [4], the underlying true causal structure is in the set of recovered DAGs if the number of observed data samples goes to infinity. Two common approaches for learning causal structures are score-based and constraint-based approaches. In the score-based approach, in order to find a set of DAGs that best explains dependency relations among the variables, a score function is evaluated, which might become an NP-hard problem [5].

In the constraint-based approach, such DAGs are found by performing a number of conditional independence (CI) tests. Sprites and Glymour [4] proposed a promising solution, called PC algorithm. For ground-truth graphs with bounded degrees, PC algorithm does not require to perform high-order conditional independence tests, and thus, runs in polynomial time. PC algorithm has become a common tool for causal explorations and is available in different graphical model learning packages such as pcalg [6], bnlearn [7], and TETRAD [8]. Moreover, it has been widely applied in different applications such as learning the causal structure of GRNs from gene expression data [9], [10]. Furthermore, a number of causal structure learning algorithms, for instance, FCI and its variants such as RFCI [4], [11], and CCD algorithm [12], use PC algorithm as a subroutine.

PC algorithm starts from a complete undirected graph and removes the edges in consecutive levels based on carefully selected conditional independence tests. However, performing these number of tests might take a few days on a single machine in some gene expression data such as DREAM5-Insilico dataset [13]. Furthermore, the order of performing conditional independence tests may affect the final result. Parallel implementations of PC algorithm on multi-core CPUs have been proposed in [14], [15]. In [16], Colombo and Maathuis proposed a variant of PC algorithm called PC-stable which is order-independent and produces less error compared with the original PC algorithm. The key property of PC-stable is that removing an edge in a level has no effect on performing conditional independence tests of other edges in that level. This order-independent property makes PC-stable suitable for executing on multi-core machines. In [17], Le et al. proposed a parallel implementation of PC-stable algorithm on multi-core CPUs, called Parallel-PC, which reduces the runtime by an order of magnitude. For instance, it takes a couple of hours to process DREAM5-Insilico dataset. In case of using GPU hardware, there was an attempt for parallelization of the PC-stable algorithm in [18]. However, only a small part (only level zero and level one) of the PC-stable algorithm is parallelized in this method, and thus, it cannot be used as a complete solution in many datasets which require more than two levels. In

The authors are with Learning and Intelligent Systems Laboratory, Department of Electrical Engineering, Sharif University of Technology, Tehran, Iran. E-mails: behrooz.zare@ee.sharif.edu, foad.jafarinejad@ee.sharif.edu, matin@sharif.edu (corresponding author), saleh@sharif.edu.

fact, their approach cannot be generalized to level two and beyond.

In this paper, we propose a GPU-based parallel algorithm, called "cuPC", for learning causal structures based on PC-stable. We assume that there is no missing observations for any variable, and data has multivariate normal distribution. In order to execute PC-stable, one needs to perform conditional independence tests to evaluate whether two variables V_i and V_j are independent given another set of variables S. The proposed algorithm has two variants, called "cuPC-E" and "cuPC-S", which employ the following ideas.

I) cuPC-E employs two degrees of parallelism at the same time. First is performing tests for multiple edges in parallel and second, is parallelizing the tests which are performed for a given edge. Although abundant parallelism is available, parallelizing all such tests does not yield the highest performance because it incurs different overheads and also results in many unnecessary tests. Instead, cuPC-E judiciously strikes a balance between the two degrees of parallelism in order to efficiently utilize the parallel computing capabilities of GPU and avoid launching unnecessary tests at the same time. In addition, cuPC-E employs two configuration parameters which can be adjusted to tune the performance and achieve high speedup in both sparse and dense graphs.

II) A conditional set S might be common in tests of many pairs of variables. cuPC-S takes advantage of this property and reuses the results of computations in one of such tests in the others. This sharing can be performed in different ways. For instance, sharing all redundant computations in processing the entire graph might first seem more beneficial, but it has non-justifiable overheads. Hence, cuPC-S employs a carefully-designed local sharing strategy in order to avoid different overheads and achieve significant speedup.

III) cuPC-E and cuPC-S parallel algorithms avoid storing the indices of variables in set S. Instead, a combination function is employed to compute the indices on-the-fly and also in parallel. *IV)* The causal structure is represented by an adjacency matrix which is compacted before starting the computations in every level. The compacted format is judiciously selected to assign and execute parallel threads more efficiently, and also, improves cache performance. *V)* GPU shared memory is used in order to improve performance. *VI)* Where applicable, threads are terminated early in order to avoid performing unnecessary computations. For instance, edge removals are monitored in parallel, and when an edge is removed in another thread or another block, the rest of the tests on that edge are skipped.

Experiments on multiple datasets show the scalability of the proposed parallel algorithms with respect to the number of variables, the number of samples, and different graph densities. For instance, in one of the most challenging datasets, cuPC-S can reduce the runtime of PC-stable from more than 11 hours to about 4 seconds. On average, cuPC-E and cuPC-S achieve about 500 X and 1300 X speedup, respectively, compared to serial implementation on CPU.

The rest of this paper is organized as follows. In Section 2, we review some preliminaries on causal Bayesian networks and description of PC-stable. In Section 3, we present the two variants of cuPC algorithm, cuPC-E and cuPC-S. Furthermore, we elaborate details of our contributions in Section 4. We conduct experiments to evaluate the performance and scalability of the proposed solution in Section 5 and conclude our results in Section 6.

2 Preliminaries

2.1 Bayesian Networks

Consider a set of **random variables** $\mathcal{V} = \{V_1, V_2, \dots, V_n\}$. Given $X, Y, Z \subseteq \mathcal{V}$, a **conditional independence (CI) assertion** of the form $X \perp\!\!\!\perp Y|Z$ means X and Y are independent given Z. A **CI test** of the form $I(X, Y|Z)$ is a test procedure based on observed data samples from X, Y and Z which determines whether the corresponding CI assertion $X \perp\!\!\!\perp Y|Z$ holds or not. Section 4.3 describes how to perform CI tests from observed-data samples.

Graphical model G is a graph which encodes a **joint distribution** P over the random variables in $\mathcal{V}$. The reason behind the development of a graphical model is that the explicit representation of the joint distribution becomes infeasible as the number of variables grows. Furthermore, under some assumptions on the data generating model, one can interpret causal relations among the variables from these graphs [19].

Bayesian Networks (BN) are a class of graphical models that represent a factorization of P over $\mathcal{V}$ by a directed acyclic graph (DAG) $G = (\mathcal{V}, \mathcal{E})$ as

$$P(V_1, V_2, \dots, V_n) = \prod_{i=1}^{n} P(V_i|par(V_i)), \qquad (1)$$

where $\mathcal{E}$ is the set of edges, and $par(V_i)$ denotes parents of V_i in G. Moreover, the graph G encodes conditional independence between the random variables in $\mathcal{V}$ by some notion of separation in graphs.

A **Causal Bayesian Network (CBN)** is a BN where each directed edge represents a cause-effect relationship from the parent to its child. For the exact definition of CBN, please refer to [20], Section 1.3. A CBN satisfies causal Markov condition, i.e., given $par(V_i)$, variable V_i is independent of any variable V_j that there is no directed path from V_i to V_j. Let $\mathcal{I}(P)$ be the set of all CI assertions that holds in P. Under causal Markov condition and faithful assumptions [4], all CI assertions in $\mathcal{I}(P)$ are encoded in the true causal graph G [21].

2.2 CPDAG

In a directed graph G, we say that three variables $V_i, V_k, V_j \in \mathcal{V}$ form a **v-structure** at V_k if variables V_i and V_j have an outgoing edge to variable V_k while they are not connected by any edge in G. This is denoted by $V_i \rightarrow V_k \leftarrow V_j$. The **skeleton** of a directed graph G is an undirected graph that contains edges of G without considering their orientations.

For a given joint distribution P, there might be different DAGs that can represent $\mathcal{I}(P)$. The set of all such DAGs is called **Markov equivalence class** [22]. It can be shown that two DAGs are in the same Markov equivalence class if they have the same **skeleton** and the same set of **v-structures** [23]. A Markov equivalence class can be represented uniquely by a mixed graph called **completed partial DAG (CPDAG)**. In particular, there is a directed edge in CPDAG from V_i to V_j if this edge exists with the same direction in all DAGs in the Markov equivalent class. There is an undirected edge between V_i and V_j in CPDAG if there exist two DAGs in the Markov equivalence class which have an edge between V_i and V_j but with different orientations.

2.3 Causal Structure Learning

Causal structure learning, our focus in this paper, is the problem of finding a CPDAG which best describes dependency relations in a given data that is sampled from the random variables in $\mathcal{V}$. In the literature, two main approaches have been proposed for causal structure learning [24]: constraint-based approach and score-based approach.

In the constraint-based approach, CI tests are utilized to recover the CPDAG. Examples include PC [4], Rank PC [25], PC-stable [16], IC [26], and FCI [4]. In the score-based approach, a score function indicates how well each DAG explains dependency relations in the data. Then, a CPDAG with the highest score is obtained by searching over Markov equivalence classes. Examples include Chow-Liu [27] and GES [28] algorithms. There are other methods such as LiNGAM [29], [30], and BACKSHIFT [31] which do not belong to any of the above two categories because their underlying assumptions are more restricted or their settings are different.

Choosing between the types of algorithms depends on the characteristics of the data [32]. For instance, Scutari et al. [33] concluded that constraint-based algorithms are more accurate than score-based algorithms for small sample sizes and that they are as accurate as hybrid algorithms. PC algorithm, as one of the main constraint-based algorithms, has become a common tool for causal explorations and is available in different graphical model learning packages [6], [7], [8]. In addition, a number of causal structure learning algorithms utilize PC algorithm as a subroutine [4], [11], [12].

2.4 PC-stable Algorithm

In the constraint-based approach, a naive solution to check whether there is an edge between two variables V_i and V_j in the CPDAG is to perform all CI tests of the form $I(V_i, V_j|S)$ where $S \subseteq \mathcal{V}\backslash\{V_i, V_j\}$. This solution is computationally infeasible for large number of variables due to exponentially growing number of CI tests.

Unlike the naive solution, the PC algorithm is computationally efficient for sparse graphs with up to thousands number of variables and is commonly used in high-dimensional settings [34]. Here we describe PC-stable algorithm which is a variation of PC with less estimation errors [16].

Algorithm 1 The first step in PC-stable algorithm.

```
Input: $\mathcal{V}$
Output: $G$, $SepSet$
1:  $G$ = fully connected graph
2:  $SepSet = \emptyset$
3:  $\ell = 0$
4:  repeat
5:      Copy $G$ into $G'$
6:      for any edge $(V_i, V_j)$ in $G$ do
7:          repeat
8:              Choose a new $S \subseteq adj(V_i, G')\backslash\{V_j\}$ with $|S| = \ell$
9:              Perform $I(V_i, V_j|S)$
10:             if $V_i \perp\!\!\!\perp V_j|S$ then
11:                 Remove $(V_i, V_j)$ from $G$
12:                 Store $S$ in $SepSet$
13:             end if
14:         until $(V_i, V_j)$ is removed or all sets $S$ are considered
15:     end for
16:     $\ell = \ell + 1$
17: until ( max degree $-1 \geq \ell$ )
```

PC-stable algorithm consists of two main steps: In the first step, the skeleton is determined by performing a number of carefully selected CI tests. In the second step, the set of v-structures are extracted and as many of the undirected edges as possible are oriented by applying a set of rules called Meek rules [35]. The second step is fairly fast. The first step is computationally intensive [15] and forms our focus in this paper. For instance, in ground truth graphs with a bound Δ on the maximum degree, the time complexity of PC-stable algorithm is in the order of $O(n^{\Delta})$. Sections 3 and 4 present our proposed solution for acceleration of this step on GPU. Details of the first step are described in the following.

See Algorithm 1. First, G is initiated with a fully connected undirected graph over set $\mathcal{V}$ (line 1). Next, the extra edges are removed from G by performing a number of CI tests. The tests are performed by levels. In each level ℓ, first a copy of G is stored in G' (line 5). Next, for every edge (V_i, V_j) in graph G, a CI test $I(V_i, V_j|S)$ is performed for any $S \subseteq adj(V_i, G')\backslash\{V_j\}$ such that $|S| = \ell$ (lines $6-9$), where $adj(V_i, G')$ denotes the neighbors of V_i in G' (see Section 4.3 for the details of performing a CI test). If there exists a set S where V_i is independent of V_j given S (line 10), edge (V_i, V_j) is removed from G, and S is stored in $SepSet$ (lines $11-12$). Once all the edges are considered, ℓ is incremented (line 16) and the above procedure is repeated. The algorithm continues as long as the maximum degree of the graph is large enough (line 17). The second step in PC-stable is to use $SepSet$ to find v-structures and orient the edges of graph G.

Fig. 1 illustrates execution of the first step on a small graph. In level $\ell = 0$, six CI tests are performed, one for every edge in the fully connected graph. Assuming that the result of the fourth CI test is true, we have $V_1 \perp\!\!\!\perp V_2$, and hence, edge (V_1, V_2) is removed. In level $\ell = 1$, 12 CI tests are performed and edges (V_1, V_3) and (V_2, V_3) are removed.

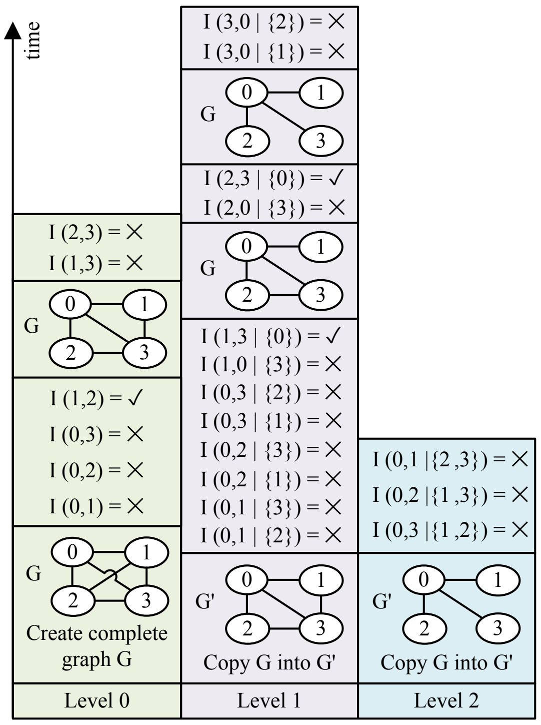


Fig. 1. An example of execution of PC-stable algorithm. For better readability, we use the term i instead of V_i. For instance, $I(0,1)$ actually means $I(V_0, V_1)$. This is done in Fig. 3 and Fig. 4 as well.

Note that by selecting the conditional sets S from G' but removing edges from G, the algorithm finally reaches the same graph regardless of the edge selection order. In other words, during the execution of the algorithm in a level, S only depends on G'. Since performing CI tests in each level is independent of the edge selection order, making an error in one of the CI tests does not have any impact on other CI tests in that level.

3 cuPC: CUDA-Accelerated PC Algorithm

This section presents our proposed solution for acceleration of the computationally-intensive portion of PC-stable (lines $5-15$ in Algorithm 1) on GPU using CUDA parallel programming API. We assume that there is no missing observations for any variable, and data has multivariate normal distribution. The overall view of the proposed method is shown in Algorithm 2. The main loop on ℓ which iterates through the levels still exists in the proposed solution, but the internal computations of every level are accelerated on GPU. In specific, since the computations of level zero can be simplified, a separate parallel algorithm is employed for this level (line 7 in Algorithm 2). For every level $\ell \geq 1$, first G is copied into G' (line 9), and then, the required computations are performed (line 10). Note that we work on adjacency matrix of graph G denoted as A_G. In order to increase the efficiency of the proposed parallel algorithms, A'_G is a compacted version of A_G. The details are discussed in the following.

A short background on CUDA is presented in Section 3.1. Acceleration of level $\ell = 0$ is discussed in Section 3.2. For levels $\ell \geq 1$, two different parallel algorithms called cuPC-E and cuPC-S are proposed. cuPC-E and the compact procedure are discussed in Section 3.3. cuPC-S is discussed in Section 3.4. Further details on some parts of the proposed solution are discussed later in Section 4.

Algorithm 2 Overall view of the proposed solution. Lines 7, 9, and 10 are executed in parallel on GPU.

```
Input: $\mathcal{V}$
Output: $G$, $SepSet$
 1: $G$ = fully connected graph
 2: $SepSet = \emptyset$
 3: $\ell = 0$
 4: $A_G$ = adjacency matrix of graph $G$
 5: repeat
 6:     if ($\ell == 0$) then
 7:         GPU: execute level zero
 8:     else
 9:         GPU: compact $A_G$ into $A'_G$
10:         GPU: execute level $\ell$
11:     end if
12:     $\ell = \ell + 1$
13: until ( max degree $-1 \geq \ell$ )
```

3.1 CUDA

CUDA is a parallel programming API for Nvidia GPUs. GPU is a massively parallel processor with hundreds to thousands of cores. CUDA follows a hierarchical programming model. At the top level, computationally intensive functions are specified by the programmer as CUDA **kernels**. A kernel is specified as a sequential function for a single **thread**. The kernel is then launched for parallel execution on the GPU by specifying the number of concurrent threads.

Threads are grouped into **blocks**. A kernel consists of a number of blocks, and every block consists of a number of threads. Every block has access to a small, on-chip and low-latency memory, called **shared memory**. The shared memory of a block is accessible to all threads within that block, but not to any thread from other blocks[1].

In order to identify blocks within a kernel, and also, threads within a block, a set of indices are used in the CUDA API, for instance, $blockIdx.y$ and $blockIdx.x$ as the block index in dimension y and dimension x within a 2D kernel, and $threadIdx.y$ and $threadIdx.x$ as the thread index in dimensions y and x within a 2D block. For brevity, we denote these four indices as by, bx, ty and tx, respectively.

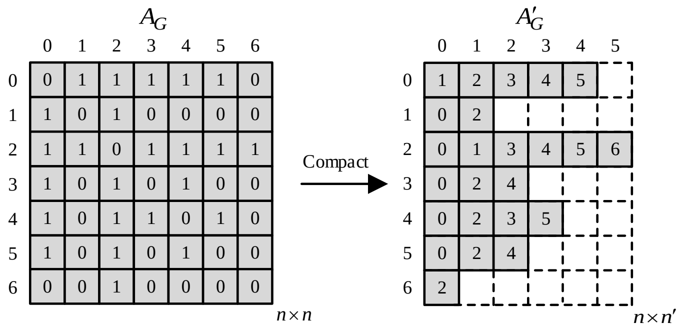
3.2 Level $\ell = 0$

Size of conditional sets S is equal to ℓ (Algorithm 1, line 8). As a result, in level zero, $S = \emptyset$, and therefore, the required computations can be simplified. In specific, for

1. This article is presented based on CUDA programming framework. However, the presented ideas and parallel algorithms can readily be ported to OpenCL programming framework for other GPU vendors as well. In specific, block, thread and shared memory in CUDA programming framework correspond to work-group, work-item, and local memory in OpenCL programming framework.

Algorithm 3 Acceleration of level $\ell = 0$. See Section 3.2.

```
Input: $A_G$
Output: $A_G$
# of blocks: $n/32 \times n/32$
# of threads / block: $32 \times 32$
1: $i = by \times 32 + ty$
2: $j = bx \times 32 + tx$
3: if $(i < j)$ then
4:     Perform $I(V_i, V_j)$
5:     if $(V_i \perp\!\!\!\perp V_j)$ then
6:         $A_G[i, j] = A_G[j, i] = 0$
7:     end if
8: end if
```



$$A_G = \begin{array}{c|ccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 3 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 4 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 5 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array} \qquad A'_G = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 & \\ 1 & 0 & 2 & & & & \\ 2 & 0 & 1 & 3 & 4 & 5 & 6 \\ 3 & 0 & 2 & 4 & & & \\ 4 & 0 & 2 & 3 & 5 & & \\ 5 & 0 & 2 & 4 & & & \\ 6 & 2 & & & & & \end{array}$$

Fig. 2. A'_G is formed by compacting A_G.

every edge (V_i, V_j) in G, only one CI test is required, which is $I(V_i, V_j|\emptyset)$ or simply $I(V_i, V_j)$. In addition, copying G into G' is not required.

All the required CI tests $I(V_i, V_j)$ are performed in parallel as shown in Algorithm 3. Since the input graph in level zero is a fully connected undirected graph, a total of $n(n-1)/2$ tests are required, i.e., one for every edge. Every test $I(V_i, V_j)$ is assigned to a separate thread and n^2 threads are launched. Threads are grouped in a 2D kernel of $n/32 \times n/32$ blocks. Every block has 32×32 threads. Indices i and j are calculated in lines $1-2$. Here, $0 \leq by, bx < n/32$ and $0 \leq ty, tx < 32$.

Lines $4-7$ are executed in only $n(n-1)/2$ threads. In line 4, the CI test $I(V_i, V_j)$ is performed. In lines $5-7$, the edge (V_i, V_j) is removed from graph G if $V_i \perp\!\!\!\perp V_j$. The term A_G denotes the adjacency matrix of graph G. Edge (V_i, V_j) is removed from G by setting $A_G[i, j] = A_G[j, i] = 0$.

3.3 Level $\ell \geq 1$: Parallel Algorithm cuPC-E

Two different parallel algorithms (kernels) are proposed for acceleration of every level $\ell \geq 1$. This section describes the first algorithm, called cuPC-E. See Algorithm 4.

Compact: cuPC-E takes ℓ, A_G and A'_G as input. As shown in line 9 in Algorithm 2, A'_G is formed by compacting adjacency matrix A_G into a sparse representation. Fig. 2 illustrates a small example. An element with value j in i-th row in A'_G denotes existence of edge (V_i, V_j) in A_G. A'_G has n rows. Row i has n'_i elements, i.e., edges. Let $n' = \max_{0 \leq i < n} n'_i$. Note that A'_G can be implemented in different formats such as linked lists as in adjacency list representations [36].

Algorithm 4 Acceleration of level $\ell \geq 1$ with parallel algorithm cuPC-E. See Section 3.3 and Fig. 3.

```
Input: $A_G$, $A'_G$, $\ell$
Output: $A_G$, $SepSet$
# of blocks: $n \times n'/\beta$
# of threads / block: $\gamma \times \beta$
1: $i = by$
2: $n'_i$ = size of row $i$ in $A'_G$
3: Copy the entire row $i$ from matrix $A'_G$ into vector $A'_{sh}$
   in shared memory
4: $p = bx \times \beta + tx$
5: $j = A'_{sh}[p]$
6: for $(t = ty;\ t < \binom{n'_i - 1}{\ell};\ t = t + \gamma)$ do
7:     if $(A_G[i, j] == 1)$ then
8:         $P_{1\times\ell} = Comb(n'_i - 1, \ell, t, p)$
9:         $S_{1\times\ell} = A'_{sh}[P]$
10:        Perform $I(V_i, V_j|S)$
11:        if $(V_i \perp\!\!\!\perp V_j|S)$ then
12:            $A_G[i, j] = A_G[j, i] = 0$
13:            Store $S$ in $SepSet$
14:        end if
15:    end if
16: end for
```

However, since linked lists are not efficient for parallel execution, A'_G is implemented as a matrix with n rows and $n' + 1$ columns. The element at the last column of each row i stores n'_i. The $Compact$ procedure is executed in parallel by employing another parallel algorithm called $scan$ [37], [38]. Details are removed for brevity.

Blocks and Threads: cuPC-E kernel consists of $n \times n'/\beta$ blocks. See Fig. 3(a). Every block performs the required CI tests for β edges, i.e., β consecutive elements from one row in A'_G. In Fig. 3(a), there are 7×2 blocks. Block $(2, 1)$, which is marked with green color, works on $\beta = 3$ edges, namely, (V_2, V_4), (V_2, V_5), and (V_2, V_6). See Fig. 3(b). The CI tests for each one of the β edges are split among γ threads. Hence, every block consists of $\gamma \times \beta$ threads. In Fig. 3(d), there are 2×3 threads in block $(2, 1)$. Thread $(1, 1)$ in this block, which is marked with purple color, works on half of the CI tests for edge (V_2, V_5). Thread $(0, 1)$ works on the other half.

Shared Memory: The threads in block (by, bx) frequently access different elements in row by in A'_G. Therefore, in order to speedup the memory accesses, the entire row is copied into the block's shared memory, i.e., into vector A'_{sh}. See Fig. 3(c).

Index Calculations: Let (V_i, V_j) denote the target edges in block (by, bx). For all threads within block (by, bx), i is equal to by. See line 1 in Algorithm 4. For thread (ty, tx) in this block, j is equal to the tx-th element in the green portion of the corresponding row, i.e., element $bx \times \beta + tx$ in A'_{sh}. See lines $4-5$ in Algorithm 4, and also, Fig. 3(c).

Combinations: Consider all CI tests $I(V_i, V_j|S)$ for edge (V_i, V_j). Set S is formed by selecting ℓ elements from row i in A'_G or equivalently from A'_{sh}. Since element j in this row should not be selected, there will remain $n'_i - 1$ elements to choose from. Therefore, there are a total of $\binom{n'_i - 1}{\ell}$ possible

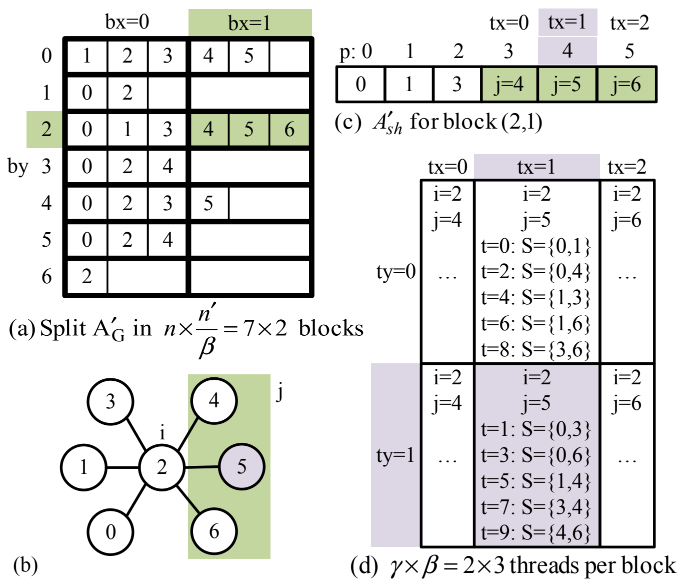


Fig. 3. Blocks and threads in cuPC-E parallel algorithm. In this example, $n = 7$, $n' = 6$, $\beta = 3$, $\gamma = 2$ and $\ell = 2$. Block $(2, 1)$ is marked with green color, and thread $(1, 1)$ in this block is marked with purple color. For threads $(0, 1)$ and $(1, 1)$ in block $(2, 1)$, S is selected from set $\{0, 1, 3, 4, 6\}$.

combinations for set S. CI tests of an edge (V_i, V_j) are split among γ threads. In the example of Fig. 3(d), $i = 2$ and $j = 5$. Hence, S is selected from $\{0, 1, 3, 4, 6\}$. There are $\binom{5}{2} = 10$ possible combinations for S. Each of the $\gamma = 2$ threads sequentially perform $10/2 = 5$ of these tests. See lines $6 - 10$ in Algorithm 4, and also, Fig. 3(d). P is an array of pointers that point to the selected elements. For instance, when $t = 9$ (the last combination), we have $P = \{3, 5\}$ and $S = \{V_4, V_6\}$. The $Comb$ function returns t-th combination in parallel, while skipping the unwanted combinations that include p, i.e., the pointer to j. Different parallel threads call this function with different values of t. The internal details of the $Comb$ function are discussed later in Section 4.2.

Edge Removal: In lines $10 - 14$, one CI test $I(V_i, V_j|S)$ is performed, and if $V_i \perp\!\!\!\perp V_j|S$, the edge (V_i, V_j) is removed from A_G. As mentioned before in Algorithm 1, the conditional sets S are selected from G' but the edges are removed from G.

Key Features: Important features of cuPC-E parallel algorithm are discussed in the following. *I)* cuPC-E offers two degrees of parallelism, in specific, processing all the edges in parallel, and for every edge, performing the CI tests in parallel. Although abundant parallelism is available, parallelizing all such tests does not yield the highest performance. cuPC-E does not fully parallelize all CI tests for an edge. The number of CI tests for edge (V_i, V_j) is equal to $\binom{n'_i - 1}{\ell}$, while these CI tests are performed by only γ parallel threads. In the example of Fig. 3(d), for edge (V_2, V_5), 10 tests are performed by 2 parallel threads. When one of these γ threads removes the target edge, we no longer need to perform the rest of the CI tests for that edge. The if statement in line 7 in Algorithm 4 blocks these unnecessary tests. $\gamma = 1$ avoids all the unnecessary tests but is sequential, and $\gamma = \binom{n'_i - 1}{\ell}$ is fully parallel but does not avoid any of the unnecessary tests. Parallel algorithm cuPC-E, therefore, strikes a balance by judiciously employing partial parallelism of the CI tests.

II) Edge removals are monitored in parallel in order to avoid unnecessary tests. In specific, when edge (V_i, V_j) is removed by another block, i.e., by a block with $by = j$, the same if statement in line 7 in Algorithm 4 blocks the unnecessary tests.

III) All indices required for fetching sets S are calculated on-the-fly and also in parallel based on a combination function (Section 4.2), and hence, cuPC-E does not use extra memory for storing the indices.

IV) Processing the compacted version of the adjacency matrix removes unnecessary checks for zero elements of A_G, reduces total number of combinations for set S, and also leads to better cache performance. The compacted format is judiciously selected to match the proposed method.

V) Use of shared memory for the rows of A'_G increases the performance. Note that every block has only one copy of its corresponding row but processes β edges. Storing the correlation matrix C or the set of combinations in shared memory is not beneficial.

3.4 Level $\ell \geq 1$: Parallel Algorithm cuPC-S

Every CI test $I(V_i, V_j|S)$ includes computing pseudo-inverse of a matrix M_2. See Sections 4.3 and 4.4 for the details. Pseudo-inverse computations are time consuming. cuPC-S employs the following idea in order to accelerate the process. The matrix M_2, which requires inversion, depends only on set S, and not V_i or V_j. See Equation 4. Therefore, by assigning the CI tests that depend on the same set S to a single thread, it is possible to avoid multiple calculations of the same pseudo-inverse by sharing it among the CI tests. See Algorithm 5.

Blocks and Threads: cuPC-S kernel consists of $n \times \delta$ blocks. For a given row i in A'_G, there exist $\binom{n'_i}{\ell}$ possible sets S of size ℓ. Processing of these sets are split among δ blocks, each containing θ threads. Each one of these $\delta \times \theta$ threads, therefore, is responsible for processing $\binom{n'_i}{\ell}/(\delta \times \theta)$ sets S.

Fig. 4 illustrates a small example. Row 2 contains $n'_2 = 6$ elements. Therefore, there are $\binom{6}{2} = 15$ possible sets S for this row, which are split among $\delta = 2$ blocks, each containing $\theta = 4$ threads. See Fig. 4(b). Block $(2, 1)$ is marked with green color, and thread 0 within this block is marked with purple color. This thread works on two sets S, in specific, $S = \{V_0, V_6\}$ and $S = \{V_4, V_5\}$.

Index Calculations: Lines $1 - 3$ in Algorithm 5 are similar to cuPC-E. Since every thread that is assigned to row $i = by$ in cuPC-S is responsible for processing $\binom{n'_i}{\ell}/(\delta \times \theta)$ sets S, the for loop in line 4 iterates $\binom{n'_i}{\ell}/(\delta \times \theta)$ times. In Fig. 4(b), it iterates twice, for instance, we have $t = 1 \times 4 + 0 = 4$ and $t = 4 + 8 = 12$ in thread 0 in block $(2, 1)$.

In every iteration, one set S is selected based on the value of t. This is done using the $Comb$ function. See lines $5 - 6$ in Algorithm 5. The selected set S is used to perform a number of CI tests $I(V_i, V_j|S)$. Since matrix M_2 depends only on S, and not V_i or V_j, we compute this matrix and its pseudo-inverse once and use the results in all these CI tests. See lines $7 - 8$.

Algorithm 5 Acceleration of level $\ell \geq 1$ with parallel algorithm cuPC-S. See Section 3.4 and Fig. 4.

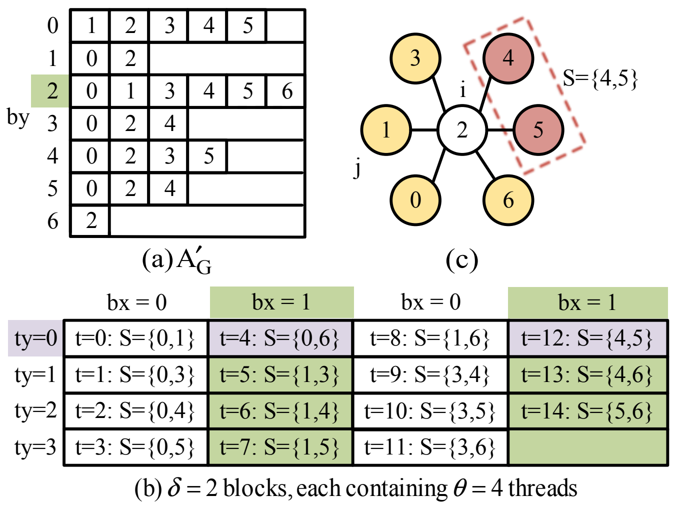
```
Input: $A_G$, $A'_G$, $\ell$
Output: $A_G$, $SepSet$
# of blocks: $n \times \delta$
# of threads / block: $\theta \times 1$
1: $i = by$
2: $n'_i$ = size of row $i$ in $A'_G$
3: Copy the entire row $i$ from matrix $A'_G$ into vector $A'_{sh}$ in shared memory
4: for ($t = bx \times \theta + ty$; $t < \binom{n'_i}{\ell}$; $t = t + \theta \times \delta$) do
5:     $P_{1\times\ell} = Comb(n', \ell, t)$
6:     $S_{1\times\ell} = A'_{sh}[P]$
7:     Form matrix $M_2$ based on set $S$ (Section 4.3)
8:     $M_2^{-1}$ = Pseudo-inverse of $M_2$ (Section 4.4)
9:     for $p = 0$ to $n'_i$ do
10:        $j = A'_{sh}[p]$
11:        if ($j \notin S$) then
12:            if ($A_G[i,j] == 1$) then
13:                Perform $I(V_i, V_j|S)$
14:                if ($V_i \perp\!\!\!\perp V_j|S$) then
15:                    $A_G[i,j] = A_G[j,i] = 0$
16:                    Store $S$ in $SepSet$
17:                end if
18:            end if
19:        end if
20:    end for
21: end for
```

In the target CI tests $I(V_i, V_j|S)$, $i = by$ and different values of j are determined in lines $9-11$ by iterating through all adjacent nodes of V_i and selecting the ones which are not in S. As an example, consider thread 0 in block $(2,1)$ in Fig. 4. This thread has two loop iterations: $t = 4$ and $t = 12$. In the second iteration ($t = 12$), we have $S = \{V_4, V_5\}$ which is marked with red color in the figure. As a result, V_j's are the other adjacent nodes of V_i, namely, V_0, V_1, V_3 and finally V_6. They are marked with orange color. See Fig. 4(c). Hence, in the second iteration ($t = 12$) in thread 0 in block $(2,1)$, the following CI tests are performed: $I(V_2, V_0|\{V_4, V_5\})$, $I(V_2, V_1|\{V_4, V_5\})$, $I(V_2, V_3|\{V_4, V_5\})$, and $I(V_2, V_6|\{V_4, V_5\})$.

Edge Removal: Lines $12-18$ in Algorithm 5 are similar to cuPC-E, except that line 13 executes faster because part of performing a CI test is to compute pseudo-inverse M_2^{-1} which is already computed in line 8.

Key Features: Similar to cuPC-E parallel algorithm, cuPC-S *I)* works on A'_G which is the compacted version of the adjacency matrix, *II)* employs shared memory, *III)* skips unnecessary CI tests via the if statement in line 12 in Algorithm 5, and *IV)* employs a parallel combination function to compute the indices of sets S. *V)* More importantly, sharing one pseudo-inverse among multiple CI tests brings a large saving.

VI) In the CUDA framework, every 32 threads within a block form a warp. Therefore, in order to maximize GPU



	bx = 0	bx = 1	bx = 0	bx = 1
ty=0	t=0: S={0,1}	t=4: S={0,6}	t=8: S={1,6}	t=12: S={4,5}
ty=1	t=1: S={0,3}	t=5: S={1,3}	t=9: S={3,4}	t=13: S={4,6}
ty=2	t=2: S={0,4}	t=6: S={1,4}	t=10: S={3,5}	t=14: S={5,6}
ty=3	t=3: S={0,5}	t=7: S={1,5}	t=11: S={3,6}	

(b) $\delta = 2$ blocks, each containing $\theta = 4$ threads

Fig. 4. Blocks and threads in cuPC-S parallel algorithm. In this example, $n = 7$, $n' = 6$, $\delta = 2$, $\theta = 4$ and $\ell = 2$. Block $(2,1)$ is marked with green color, and thread 0 in this block is marked with purple color. In the second loop iteration in this thread, we have $t = 12$, and hence, $S = \{4,5\}$ (red color). Therefore, j is equal to 0, 1, 3, and finally 6 (orange color).

utilization, the number of threads within a block, i.e., θ, should be a multiple of 32. However, $\binom{n'_i}{\ell}$ might not be divisible by $\delta \times \theta$. cuPC-S employs the following idea in order to resolve this issue. Blocks do not process all their assigned sets S in parallel. Instead, they iterate multiple times and in every iteration, process θ sets S, where θ is a multiple of 32. As a result, only the last iteration may not contain a multiple of 32 active threads.

VII) There are many CI tests $I(V_i, V_j|S)$ that share the same set S. For instance, in Fig. 4, $S = \{V_4, V_5\}$ can be shared among CI tests in not only row 2 but also row 0 because both of these rows have elements 4 and 5, i.e., because both V_0 and V_2 are connected to V_4 and V_5. See Fig. 4(a). cuPC-S only shares a set S and its corresponding pseudo-inverse M_2^{-1} locally. In other words, a set S is shared only among the CI tests $I(V_i, V_j|S)$ with the same value i, i.e., among the CI tests of edges which are connected to the same V_i. This is in contrast to sharing a set S globally, i.e., among all CI tests from the entire graph. While global sharing may yield more savings, it requires searching the entire graph. The amount of extra saving is not large enough to justify the additional cost of global search. Section 5.5 demonstrates this point through an experiment.

4 Further Details of cuPC

4.1 Early Termination

So far we have discussed only one of the early termination strategies employed in cuPC, in specific, the if statements in line 7 in Algorithm 4, and line 12 in Algorithm 5. There are other cases where further processing is no longer required, and threads may terminate early in order to save time. Such cases are listed in the following. For brevity, their corresponding if statements are not shown in Algorithm 4 and Algorithm 5. *I)* If the number of adjacent nodes of V_i

is less than $\ell + 1$, i.e., $n'_i < \ell + 1$, then all threads in the corresponding blocks are terminated because we need at least one adjacent node V_j plus ℓ other adjacent nodes for set S. *II)* In block (by, bx) in cuPC-E, if $bx \times \beta \geq n'_i$, all threads terminate. This is because n'_i, i.e., the number of edges to be processed in row $i = by$, is too small to require the processing power of this block. *III)* Similarly, in block (by, bx) in cuPC-S, if $bx \times \theta \geq \binom{n'_i}{\ell}$, all threads terminate. This is because $\binom{n'_i}{\ell}$, i.e., the number of sets S in row $i = by$, is too small.

4.2 Computing Sets of Combination in Parallel

The $Comb$ function employed in Algorithm 4 and Algorithm 5 is discussed in this section. Let $O = \{O_0, O_1, O_2, \cdots, O_{\binom{n}{\ell}-1}\}$ be the set of all possible combinations of choosing ℓ elements from set $\{1, 2, 3, \cdots, n\}$ in lexicographical order. For instance, when $n = 3$ and $\ell = 2$, we have $O_0 = [1, 2]$, $O_1 = [1, 3]$, and $O_2 = [2, 3]$. Given n, ℓ and t, the algorithm in [39] directly computes vector O_t without requiring to compute the entire set O. Thus, by utilizing this algorithm in every thread, every O_t is derived separately.

There are ℓ elements in O_t. Let $O_t = [O_t[0], \ldots, O_t[\ell-1]]$ and $O_t[-1] = 0$. According to [39], the following statement holds true:

$$t = \sum_{c=0}^{\ell-1} \sum_{k=O_t[c-1]+1}^{O_t[c]-1} \binom{n-k}{\ell-(c+1)} \tag{2}$$

Based on the above equation, Algorithm 6 iteratively computes O_t. The algorithm has ℓ iterations. In iteration c, $O_t[c]$ is computed. The value of Sum must be less than or equal to, and also, as close as possible to the value of t.

Once all the ℓ elements in O_t are computed in Algorithm 6, the following minor modifications are performed in order to use the results in cuPC-E and cuPC-S parallel algorithms. In cuPC-S, since all indices start from zero (and not one), all elements in O_t, i.e., the output of Algorithm 6, are decremented by 1. In cuPC-E, in addition to the above modification, we also need to skip all the combinations which include p, i.e., the index of j. Hence, we set the input of the $Comb$ function to $n'_i - 1$ instead of n'_i, and also, increment all the values which are larger than or equal to p by 1.

4.3 CI Tests

In practice, the CI tests need to be performed based on data samples observed from the random variables. In particular, for multivariate normal distribution, CI test $I(V_i, V_j|S)$ can be performed based on partial correlations. Let $\rho(V_i, V_j|S)$ be the partial correlation between V_i and V_j given S. Then, we have $V_i \perp\!\!\!\perp V_j|S$ if and only if $\rho(V_i, V_j|S)$ is zero. The exact procedure is described below: Let $C_{n\times n}$ be the correlation matrix among the n random variables in the set $\mathcal{V}$, and $C[V_i, V_j]$ be (i, j)-th entry in matrix C. We define a $1 \times \ell$ vector $C(V_i, S)$ as

$$C(V_i, S) := \Big[C[V_i, S[1]], C[V_i, S[2]], \ldots C[V_i, S[\ell]]\Big]_{1\times\ell}, \tag{3}$$

Algorithm 6 Combination function.

```
Input: $n$, $\ell$, $t$, $p$
Output: $O_t$
1: $Sum = 0$
2: $O_t[-1] = 0$
3: for $c = 0$ to $\ell - 1$ do
4:     $O_t[c] = O_t[c-1]$
5:     while $Sum \leq t$ do
6:         $O_t[c] = O_t[c] + 1$
7:         $Sum = Sum + \binom{n-O_t[c]}{\ell-(c+1)}$
8:     end while
9:     $Sum = Sum - \binom{n-O_t[c]}{\ell-(c+1)}$
10: end for
```

where $S[k]$ is the k-th element in the set S. In order to compute $\rho(V_i, V_j|S)$, we first extract M_0, M_1, and M_2 matrices from the correlation matrix as the following:

$$M_0 = \begin{bmatrix} C[V_i, V_i] & C[V_i, V_j] \\ C[V_j, V_i] & C[V_j, V_j] \end{bmatrix}_{2\times 2}, \quad M_1 = \begin{bmatrix} C(V_i, S) \\ C(V_j, S) \end{bmatrix}_{2\times\ell},$$
$$M_2 = \begin{bmatrix} C(S[1], S) \\ C(S[2], S) \\ \vdots \\ C(S[\ell], S) \end{bmatrix}_{\ell\times\ell}. \tag{4}$$

Next, we obtain matrix $H = M_0 - M_1 \times M_2^{-1} \times M_1^T$. Note that M_2 might be ill-conditioned, and hence, M_2^{-1} needs to be computed using a pseudo-inverse algorithm (Section 4.4). Once H which is a 2×2 matrix is computed, an estimation of $\rho(V_i, V_j|S)$ is computed as the following:

$$\hat{\rho}(V_i, V_j|S) = \frac{H[1,2]}{\sqrt{H[1,1] \times H[2,2]}}. \tag{5}$$

In order to test whether the value of $\hat{\rho}(V_i, V_j|S)$ implies $V_i \perp\!\!\!\perp V_j|S$, we compute Fisher's z-transform [34] as

$$Z(\hat{\rho}(V_i, V_j|S)) = \left|\frac{1}{2} \times \ln\left(\frac{1 + \hat{\rho}(V_i, V_j|S)}{1 - \hat{\rho}(V_i, V_j|S)}\right)\right|, \tag{6}$$

and compare it with the following threshold:

$$\tau = \frac{\Phi^{-1}(1 - \frac{\alpha}{2})}{\sqrt{m - |S| - 3}}, \tag{7}$$

where m, α and Φ are the size of data samples for every random variable, the significance level for testing partial correlations, and CDF of standard normal distribution, respectively. If $Z(\hat{\rho}(V_i, V_j|S)) \leq \tau$, we imply that $V_i \perp\!\!\!\perp V_j|S$. Note that in level zero, the above procedure is reduced to comparing $Z(C[V_i, V_j])$ with the threshold τ.

We can conclude that a CI test $I(V_i, V_j|S)$ can be performed based on observational data, in specific, based on the threshold τ and the correlation matrix $C_{n\times n}$ among the n random variables.

Algorithm 7 Pseudo-inverse method.

```
Input: $M_2$
Output: $M_2^{-1}$
1: $L =$ Cholesky Factorization $(M_2^T \times M_2)$
2: $R = (L^T \times L)^{-1}$
3: $M_2^{-1} = L \times R \times R \times L^T \times M_2^T$
```

4.4 Pseudo-Inverse

As mentioned above, a pseudo-inverse algorithm is needed in order to compute M_2^{-1}. We employ Moore-Penrose [40] method as shown in Algorithm 7. The pseudo-inverse is computed based on two matrices L and R. Matrix L is computed as the full rank Cholesky factorization of matrix $M_2^T \times M_2$. Matrix R is computed as the inverse (the usual inverse) of $L^T \times L$.

5 Experimental Evaluation

5.1 Source Code

cuPC is implemented in the `C` language in the CUDA framework. Our parallel implementation is wrapped in a function in the `R` language with the exact same interface as the original PC-stable function in pcalg [6]. Thus, cuPC is consistent with standard casual learning `R` packages and can be easily integrated in pcalg. The source code of cuPC is available online [1].

5.2 Experiment Setup

We experimentally evaluate cuPC along with the following related previous works. Two different serial implementations of PC-stable [16] algorithm are available as part of the pcalg [41] package. The original one (called "Stable" in pcalg) is implemented in `R` language, and the recent one (called "Stable.fast") is in `C` language. A multi-threaded method, called "Parallel-PC" [17], is implemented in `R` language and is available here [42]. In addition, Stable.fast (i.e., the `C` implementation in pcalg) supports multi-threaded execution mode as well.

We employ a machine with an Intel Xeon CPU with 8 cores running at 2.5 GHz. Serial methods (Stable and Stable.fast) are executed on a single core, and multi-threaded methods (Parallel-PC and Stable.fast) are executed on all the 8 cores. The CUDA kernels in cuPC are executed on Nvidia GTX 1080 GPU which is hosted on the same machine, and the other procedures in cuPC are executed sequentially on a single core. We employ Ubuntu OS 16.04, gcc version 5.4, and CUDA version 9.2.

Six gene expression datasets are employed as our benchmarks [10], [13], [43]. These are the same benchmarks used in [17]. Table 1 shows the number of random variables and the number of samples in every dataset.

The accuracy of the proposed method is exactly the same as the one of PC-stable which was evaluated extensively in [16] in terms of True Discovery Rate (TDR) and Structural Hamming Distance (SHD). This is because cuPC is GPU-accelerated implementation of the same PC-stable algorithm.

TABLE 1
Benchmark datasets.

Dataset	# of variables (n)	# of samples (m)
NCI-60	1190	47
MCC	1380	88
BR-51	1592	50
S.cerevisiae	5361	63
S.aureus	2810	160
DREAM5-Insilico	1643	850

5.3 Performance Comparison

Comparing Serial, Multicore, and GPU:

The speedup gained by multicore and GPU implementations over serial implementations are compared in Table 2. In specific, the last column in Table 2 compares three average speedup ratios. The details are discussed below.

The first two rows in Table 2 report runtime of Stable and Parallel-PC. It is noteworthy to mention that Parallel-PC has two modes. In every benchmark, both modes are executed and the smaller runtime is reported. Runtime of Stable ranges from 11 minutes in `NCI-60` to about 3 days in `DREAM5-Insilico`. Parallel-PC takes about 11 hours in `DREAM5-Insilico`, which is 6.7 X faster than Stable. On average, Parallel-PC on eight cores is about 5.6 X faster than Stable.

The third row in Table 2 reports runtime of Stable.fast on a single core. The runtime ranges from 74 seconds in `NCI-60` to more than 11 hours in `DREAM5-Insilico`. The multi-threaded mode in Stable.fast is not yet optimized at the time of this writing. With full optimizations, the multi-threaded mode may reach linear speedup gain on multicore systems. In other words, the speedup gain on eight cores compared to serial execution may reach up to 8 X.

The fourth and fifth rows in Table 2 report runtime of cuPC-E and cuPC-S, respectively. Note that the time it takes to transfer data to and from GPU is counted as well. Runtime of cuPC-E ranges from 440 milliseconds to about 48 seconds. On average, the speedup ratio of cuPC-E over the serial execution in `C` language, i.e., Stable.fast, is 525 X. Runtime of cuPC-S ranges from 390 milliseconds to 4.76 seconds. On average, the speedup ratio of cuPC-S over serial execution is 1296 X.

Comparing cuPC with Baseline Methods:

Fig. 5 compares cuPC with the following two baseline GPU-parallel algorithms. The first algorithm is formed by basically porting parallel-PC [17] from its original multi-threaded CPU implementation to GPU. In specific, in every level ℓ, all rows i of the adjacency matrix are processed in parallel in separate blocks. In block i, all edges (V_i, V_j) are processed in parallel. All the CI tests for an edge (V_i, V_j) are performed sequentially in the corresponding thread. We also apply the same ideas in cuPC, namely, using the same compacted form of the adjacency matrix, using the same shared memory allocations, and using the same early termination strategies.

The second baseline algorithm is formed as the following. In every level ℓ, all elements ij of the adjacency matrix,

TABLE 2
Comparing serial, multicore, and GPU implementations. The first five rows show the runtime values, which are denoted as T1 to T5. The last three rows show speedup ratios, which are calculated as T1/T2, T3/T4 and T3/T5. The last column compares the geometric mean of speedup ratios.

			NCI-60	MCC	BR-51	S.cerevisiae	S.aureus	DREAM5-Insilico	
Runtime (sec.)	Stable (R, Single Core)	T1	646	3,522	3,118	10,568	11,324	265,360 (~3 days)	
	Parallel-PC (R, 8 Cores)	T2	102	570	549	2,847	1,920	39,880 (~11 hours)	
	Stable.fast (C, Single Core)	T3	74	510	491	5,567	3,359	41,668	
	cuPC-E	T4	0.44	0.85	1.15	7.99	4.21	48.08	
	cuPC-S	T5	0.39	0.44	0.56	4.76	1.64	4.09	
Speedup Ratio	Parallel-PC	T1/T2	6.3	6.2	5.7	3.7	5.9	6.7	Mean = 5.6
	cuPC-E	T3/T4	171	600	425	697	799	867	Mean = 525
	cuPC-S	T3/T5	193	1,157	868	1,170	2,052	10,178	Mean = 1,296

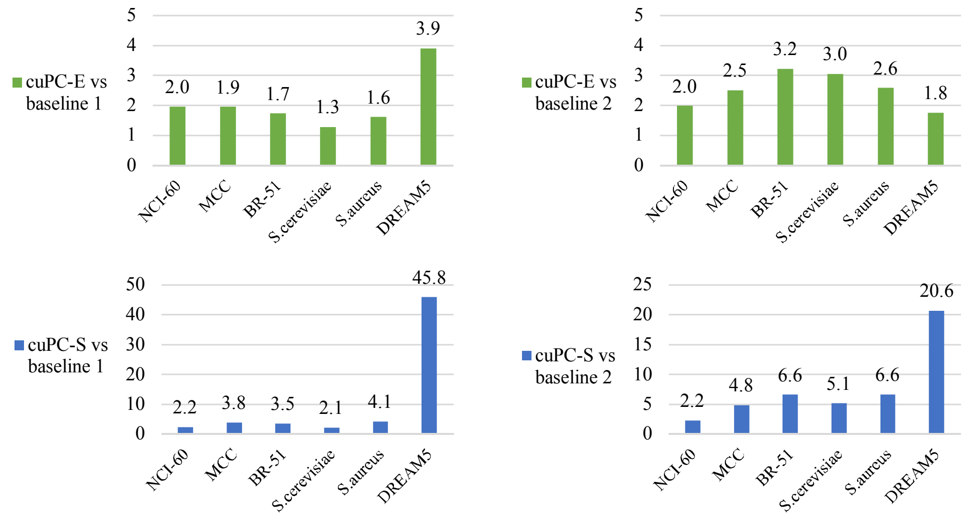


Fig. 5. Comparing the performance of cuPC-E and cuPC-S with two baseline GPU-parallel algorithms. Every bar represents a ratio between two runtime values. For instance, the bottom-right bar means cuPC-S is 20.6 X faster than baseline algorithm 2 in DREAM5-Insilico dataset.

i.e., all edges (V_i, V_j), are processed in parallel in separate blocks. In block ij, all CI tests of edge (V_i, V_j) are processed in parallel. Again, the same compact, shared memory, and early termination strategies are also applied.

As illustrated in Fig. 5, cuPC-E is 1.3 X to 3.9 X faster than baseline algorithm 1, and 1.8 X to 3.2 X faster than baseline algorithm 2. This shows that cuPC-E judiciously strikes a balance between the available degrees of parallelism and thus achieves higher performance compared to both of the baseline methods. cuPC-S is faster than cuPC-E. For instance in `DREAM5-Insilico`, which is the most challenging dataset, cuPC-S is 45.8 X and 20.6 X faster than the two baseline methods.

Comparing Different Levels:

Fig. 6 shows distribution of the runtime values in different levels in cuPC-E and cuPC-S. Note that the reported runtime of every level includes all the corresponding overheads such as forming A'_G. In the first five benchmarks, level 1 takes between 49% to 83% of the total runtime. However, in the last benchmark, level 1 takes less than 10%, but levels 2 to 5 take 90% and 70% of the total runtime in cuPC-E and cuPC-S, respectively. This figure shows that the computations in all levels contribute to the total runtime.

5.4 Configuration Parameters

cuPC-E and cuPC-S have configuration parameters which can be adjusted to improve the performance. The above results are based on executing cuPC-E with $\beta = 2$ and $\gamma = 32$, and cuPC-S with $\theta = 64$ and $\delta = 2$. We denote these selected configurations as cuPC-E-2-32 and cuPC-S-64-2. The effect of different configurations on the performance of cuPC-E and cuPC-S is evaluated in this section.

cuPC-E: 30 different configurations are experimented for cuPC-E. In specific, γ and β are selected from the set $\{1, 2, 4, \ldots, 128, 256\}$ such that $32 \leq \gamma \times \beta \leq 256$. This bounds the number of threads in every block from 32 to 256. Note that the number of blocks in cuPC-E is equal to $n \times {}^{n'}/_{\beta}$ and the number of threads in every block is equal to $\gamma \times \beta$.

The heat maps in Fig. 7 show the performance improvement or degradation of cuPC-E with different configurations compared to the selected configuration. The heat maps show a variation between 0.3 X to 1.3 X. This is mainly due to the underlying graph structure in the benchmark datasets. In particular, in denser graphs, the number of adjacent nodes is larger, and therefore, the number of CI tests required for every edge grows. As a result, in every row in the heat maps, configurations with larger γ show higher performance because more CI tests are executed in parallel. Note that the

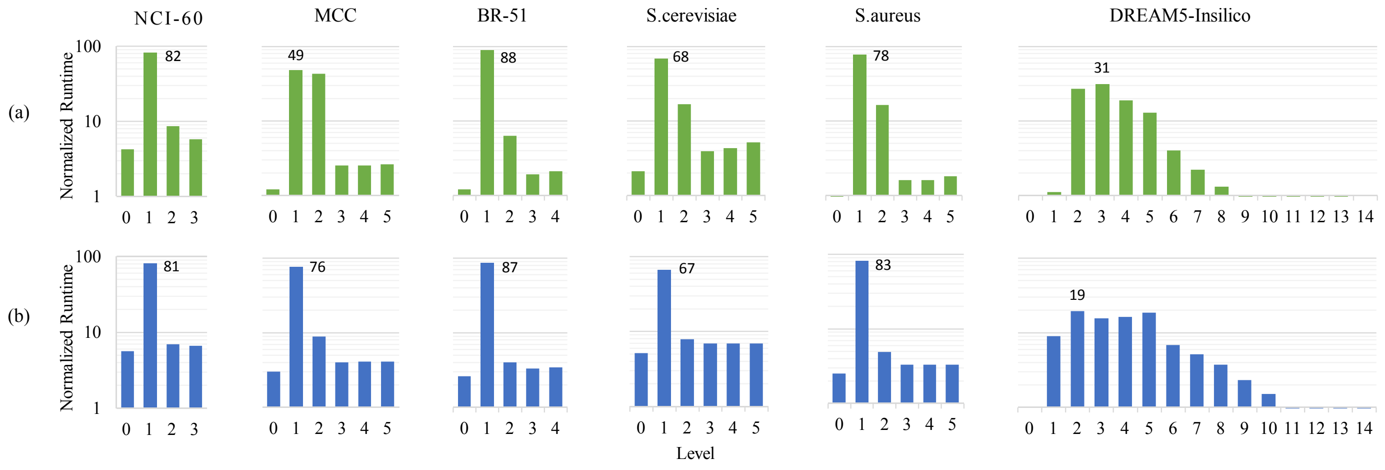
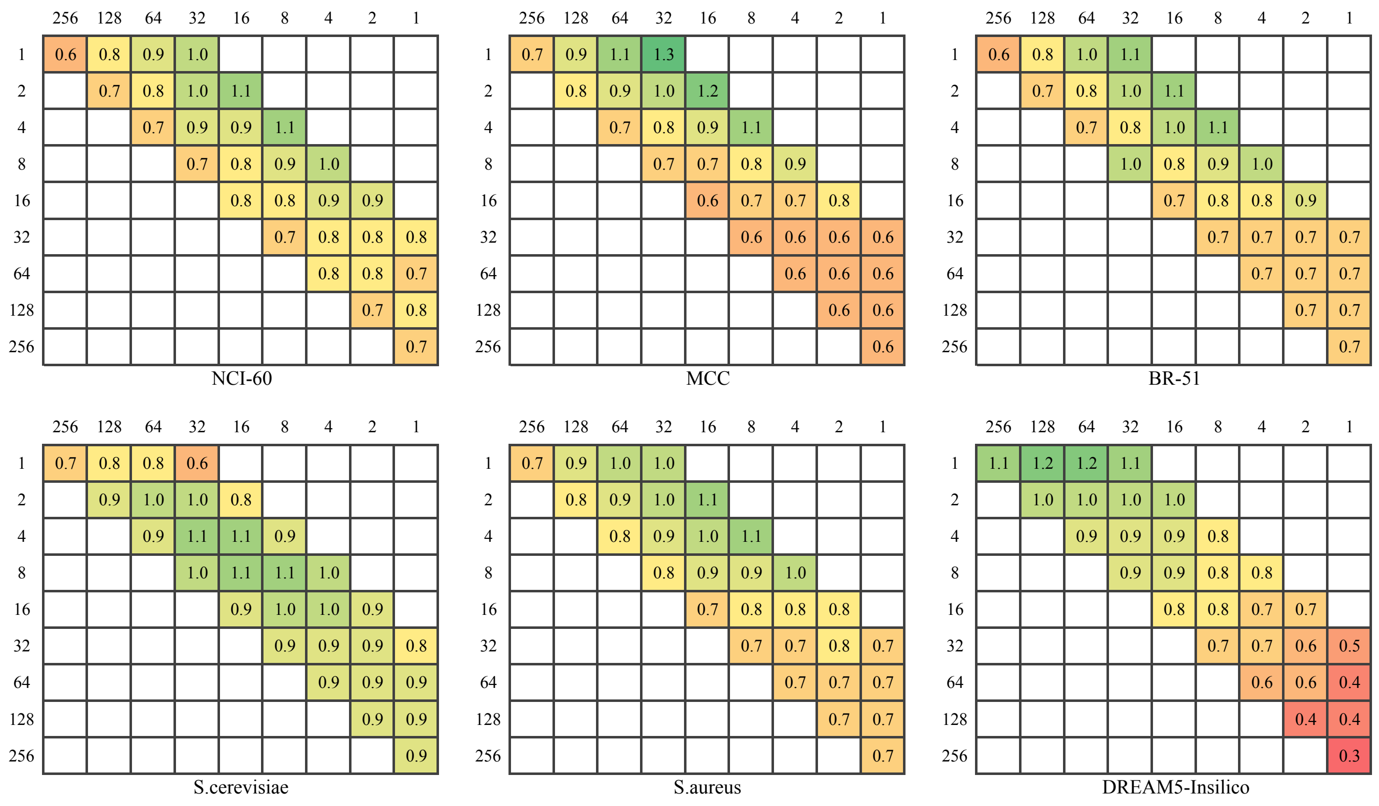


Fig. 6. Distribution of the runtime (in percent) for a) cuPC-E and b) cuPC-S, in different levels. The values are normalized to the total runtime in every benchmark.



NCI-60

	256	128	64	32	16	8	4	2	1
1	0.6	0.8	0.9	1.0					
2		0.7	0.8	1.0	1.1				
4			0.7	0.9	0.9	1.1			
8				0.7	0.8	0.9	1.0		
16					0.8	0.8	0.9	0.9	
32						0.7	0.8	0.8	0.8
64							0.8	0.8	0.7
128								0.7	0.8
256									0.7

MCC

	256	128	64	32	16	8	4	2	1
1	0.7	0.9	1.1	1.3					
2		0.8	0.9	1.0	1.2				
4			0.7	0.8	0.9	1.1			
8				0.7	0.7	0.8	0.9		
16					0.6	0.7	0.7	0.8	
32						0.6	0.6	0.6	0.6
64							0.6	0.6	0.6
128								0.6	0.6
256									0.6

BR-51

	256	128	64	32	16	8	4	2	1
1	0.6	0.8	1.0	1.1					
2		0.7	0.8	1.0	1.1				
4			0.7	0.8	1.0	1.1			
8				1.0	0.8	0.9	1.0		
16					0.7	0.8	0.8	0.9	
32						0.7	0.7	0.7	0.7
64							0.7	0.7	0.7
128								0.7	0.7
256									0.7

S.cerevisiae

	256	128	64	32	16	8	4	2	1
1	0.7	0.8	0.8	0.6					
2		0.9	1.0	1.0	0.8				
4			0.9	1.1	1.1	0.9			
8				1.0	1.1	1.1	1.0		
16					0.9	1.0	1.0	0.9	
32						0.9	0.9	0.9	0.8
64							0.9	0.9	0.9
128								0.9	0.9
256									0.9

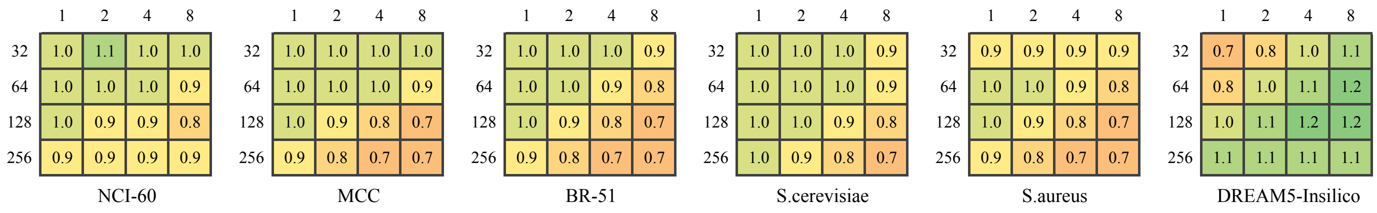
S.aureus

	256	128	64	32	16	8	4	2	1
1	0.7	0.9	1.0	1.0					
2		0.8	0.9	1.0	1.1				
4			0.8	0.9	1.0	1.1			
8				0.8	0.9	0.9	1.0		
16					0.7	0.8	0.8	0.8	
32						0.7	0.7	0.8	0.7
64							0.7	0.7	0.7
128								0.7	0.7
256									0.7

DREAM5-Insilico

	256	128	64	32	16	8	4	2	1
1	1.1	1.2	1.2	1.1					
2		1.0	1.0	1.0	1.0				
4			0.9	0.9	0.9	0.8			
8				0.9	0.9	0.8	0.8		
16					0.8	0.8	0.7	0.7	
32						0.7	0.7	0.6	0.5
64							0.6	0.6	0.4
128								0.4	0.4
256									0.3

Fig. 7. Comparing different configurations of cuPC-E with the selected configuration ($\beta = 2$ and $\gamma = 32$). The Y axis is β and the X axis is γ. Green color means higher speed and red color means lower speed.



NCI-60

	1	2	4	8
32	1.0	1.1	1.0	1.0
64	1.0	1.0	1.0	0.9
128	1.0	0.9	0.9	0.8
256	0.9	0.9	0.9	0.9

MCC

	1	2	4	8
32	1.0	1.0	1.0	1.0
64	1.0	1.0	1.0	0.9
128	1.0	0.9	0.8	0.7
256	0.9	0.8	0.7	0.7

BR-51

	1	2	4	8
32	1.0	1.0	1.0	0.9
64	1.0	1.0	0.9	0.8
128	1.0	0.9	0.8	0.7
256	0.9	0.8	0.7	0.7

S.cerevisiae

	1	2	4	8
32	1.0	1.0	1.0	0.9
64	1.0	1.0	1.0	0.9
128	1.0	1.0	0.9	0.8
256	1.0	0.9	0.8	0.7

S.aureus

	1	2	4	8
32	0.9	0.9	0.9	0.9
64	1.0	1.0	0.9	0.8
128	1.0	0.9	0.8	0.7
256	0.9	0.8	0.7	0.7

DREAM5-Insilico

	1	2	4	8
32	0.7	0.8	1.0	1.1
64	0.8	1.0	1.1	1.2
128	1.0	1.1	1.2	1.2
256	1.1	1.1	1.1	1.1

Fig. 8. Comparing different configurations of cuPC-S with the selected configuration ($\theta = 64$ and $\delta = 2$). The Y axis is θ and the X axis is δ. Green color means higher speed and red color means lower speed.

number of threads for the CI tests of an edge in cuPC-E is equal to γ. For instance, in DREAM5-Insilico, cuPC-E-4-64 shows 10% higher performance compared to cuPC-E-4-8 because 64 threads are assigned to the CI tests of an edge instead of 8. Note that there is a limit to this gain. In DREAM5-Insilico, the configuration 1-256 shows 10% lower performance compared to 1-128 because large number of parallel threads result in too many unnecessary CI tests.

As opposed to dense graphs, in sparse graphs higher performance is achieved in configurations with smaller γ in every row in the heat maps. For instance, in NCI-60, cuPC-E-2-128 shows 40% lower performance compared to cuPC-E-2-16.

cuPC-S: 16 different configurations are experimented for cuPC-S. In specific, $\theta \in \{32, 64, 128, 256\}$ and $\delta \in \{1, 2, 4, 8\}$. Note that the number of blocks in cuPC-S is equal to $n \times \delta$ and the number of threads in every block is equal to θ. Fig. 8 shows the performance improvement or degradation of cuPC-S with different configurations compared to the selected configuration.

The heat maps in Fig. 8 show a variation between 0.7 X to 1.2 X. Hence, compared to cuPC-E, cuPC-S shows less variation to the configuration parameters. This is mainly because in cuPC-S, threads are assigned to the conditional sets S instead of the edges. In other words, the number of adjacent nodes of V_i, i.e., n'_i, and hence, n' varies in dense or sparse graphs. This causes imbalance workloads in different blocks in cuPC-E. However, in cuPC-S, since $\binom{n'_i}{\ell}$ is normally much larger than n'_i, blocks are fully loaded and their workloads are more balanced.

5.5 Global Sharing vs Local Sharing in cuPC-S

As mentioned at the end of Section 3, conditional sets S can be shared either locally or globally in cuPC-S in order to save redundant computations and increase the overall speed. We employ a local sharing strategy in which only the CI tests from one row in A'_G share a set S. Global sharing among all CI tests from the entire graph is time consuming because it requires searching the entire graph to find all such CI tests. The amount of extra savings yielded by global sharing is not large enough to justify the additional cost of global search. In this section, we experimentally show the above point.

Fig. 9 shows a histogram. The value of each bin $[b_i, b_{i+1})$ is equal to the number of conditional sets S that appear in CI tests from at least b_i to at most $b_{i+1} - 1$ rows of A'_G in level 2 in DREAM5-Insilico dataset. The figure shows that about 95% of the redundant conditional sets S appear in at most 40 rows of A'_G. This is much smaller than the total number of rows in this dataset, i.e., $n = 1643$. Hence, the cost of global search is not justified.

5.6 Scalability

Scalability of the proposed parallel algorithms are evaluated in this section. In specific, performance of cuPC-E and cuPC-S are experimented for different number of variables (n), different number of samples (m), and different graph densities (d).

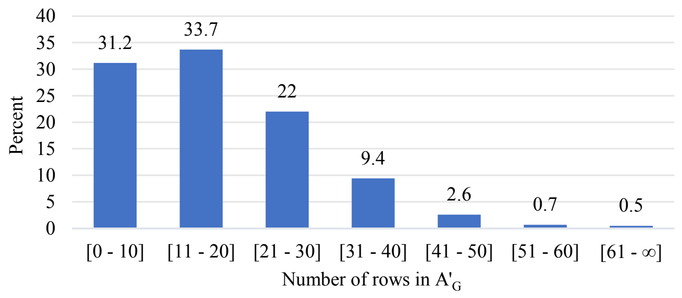


Fig. 9. The percentage of redundant conditional sets S in the entire graph in level 2 of DREAM5-Insilico dataset. See Section 5.5 for further details.

To evaluate the impact of scaling the number of variables, we consider n = 1000, 2000, 3000 and 4000. In every case, ten graphs are generated by randomly drawing an edge between any pairs of variables with probability $d = 0.1$. In particular, we first generate a random adjacency matrix A_G with independent realizations of Bernoulli random variable with parameter d in the lower triangle of the matrix and zeros in the remaining entries. Next, we replace the ones in A_G by independent realizations of a uniform random variable in the range $[0.1, 1]$. A non-zero entry $A_G[i, j]$ shows that there is a direct causal effect from V_j to V_i. Next, from $i = 0$ to $i = n - 1$, i.e., from top to bottom, the samples are generated as $V_i = N_i + \sum_{j=0}^{i-1} A_G[i, j]V_j$, where the random variables N_i's have normal distribution and are mutually independent. The sample size for every random variable is set to $m = 10000$.

Next, cuPC-E and cuPC-S are executed and the runtimes are measured in every case. The results are shown in Fig. 10(a). Runtime increases with n, but cuPC-S always has higher performance compared to cuPC-E. We also executed the C implementation of PC-stable on the same datasets. However, even in the smaller graphs (n = 1000), PC-stable could not produce results after 48 hours, and thus, we aborted the job. Hence, cuPC-E is at least 48×3600 / 20.3 sec. $\simeq$ 8500 X faster than PC-stable in this case.

Next, the impact of scaling the sample size is experimented. We consider m = 2000, 4000, 6000, 8000, and 10000. Here, n = 1000 and d = 0.1. In every case, ten random graphs are generated as discussed above and runtimes are measured. The results are shown in Fig. 10(b). The runtime increases linearly with the sample size. Increasing the sample size, improves the accuracy of the CI tests. This decreases the number of edges that are removed in level ℓ, which in turn, increases the number of CI tests required to be performed in level $\ell + 1$.

Finally, the impact of scaling the graph density is experimented. We consider d = 0.1, 0.2, 0.3, 0.4, and 0.5. Here, n = 1000 and m = 10000. The results are shown in Fig. 10(c). Increasing d means the graph is more dense, the number of remaining edges are increased, and hence, the runtime should increase. Runtime of cuPC-E and cuPC-S increase almost linearly from density 0.2 to 0.5. However, at density 0.1, the runtime is much smaller. This is because the

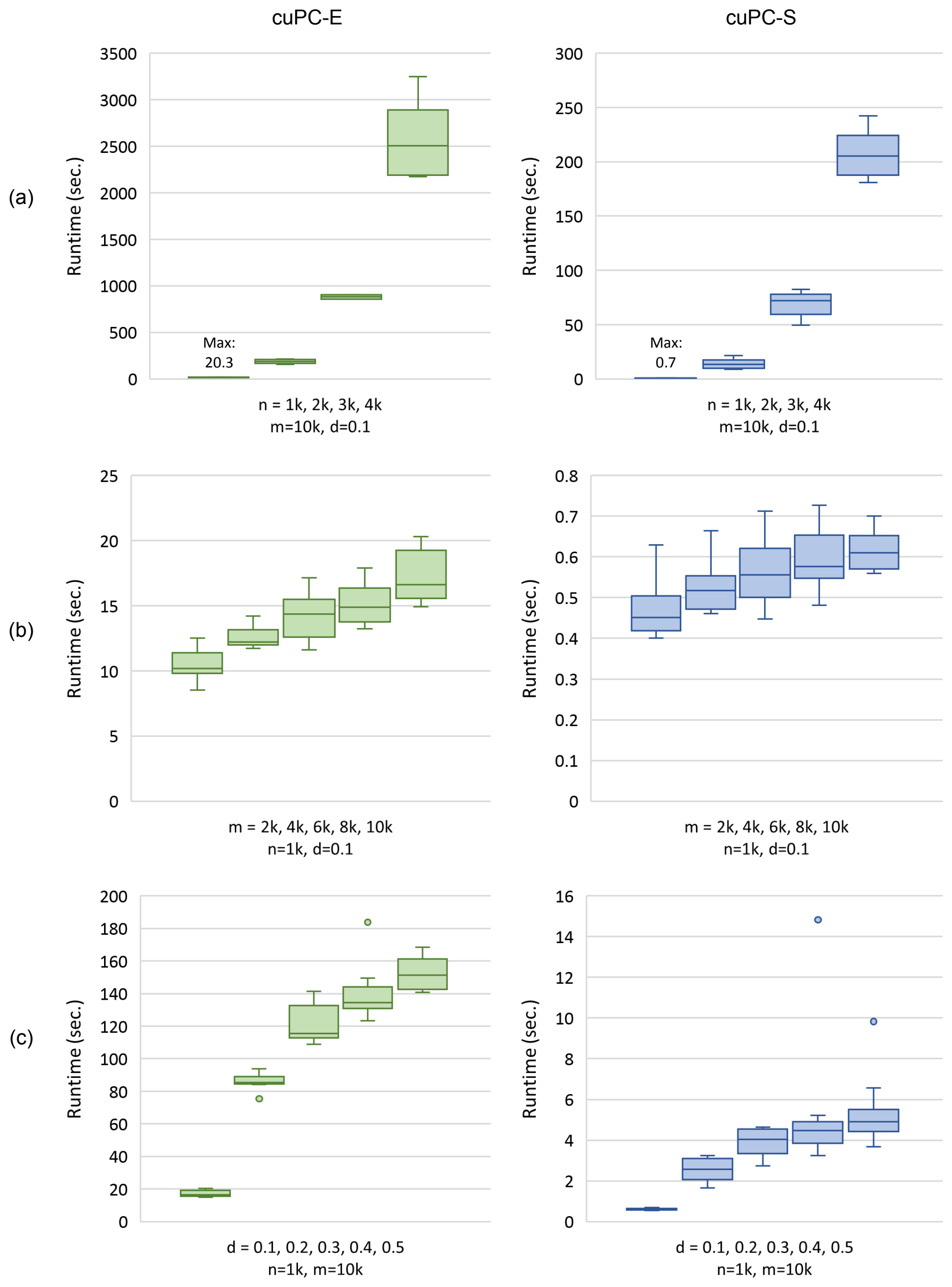


Fig. 10. Runtime of cuPC-E and cuPC-S with a) different number of variables, b) different sample sizes, and c) different graph densities. Every box-and-whisker plot shows quartiles 1, 2 (median), and 3, plus the lowest point still within 1.5 IQR of the lower quartile, and the highest point still within 1.5 IQR of the upper quartile. The outliers are shown as small circles.

runtime changes by optimizing the configuration parameters in every case, while we employ the same configuration across all values of d. Therefore, in some cases, e.g., in $d = 0.1$, the selected configuration is a better fit and the algorithm runs faster.

6 Conclusion

In empirical sciences, it is often vital to recover the underlying causal relationships among variables in real-world high-dimensional datasets. In this paper, we proposed a GPU-based parallel algorithm for PC-stable with two variants, i.e., cuPC-E and cuPC-S, to learn causal structures from observational data. Experiments showed the scalability of our prospered algorithms with respect to the number of variables, the number of samples, and different graph densities. Note that the proposed solution also helps to accelerate some other causal structure learning algorithms such as CCD, FCI, and RFCI, because they use PC algorithm as a subroutine.

References

[1] Source code is available at http://lis.ee.sharif.edu/pub/cupc and also https://github.com/LIS-Laboratory/cupc.

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

[3] J. Pearl, "Causality: models, reasoning, and inference," *Econometric Theory*, vol. 19, no. 675-685, p. 46, 2003.

[4] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search*, 2nd ed. MIT press, 2000.

[5] D. M. Chickering, D. Geiger, D. Heckerman *et al.*, "Learning bayesian networks is np-hard," Citeseer, Tech. Rep., 1994.

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

[7] M. Scutari, "Learning bayesian networks with the bnlearn r package," *arXiv preprint arXiv:0908.3817*, 2009.

[8] The Tetrad Project. [Online]. Available: http://www.phil.cmu.edu/tetrad

[9] 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, 2011.

[10] 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, p. 247, 2010.

[11] 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.

[12] T. Richardson, "A discovery algorithm for directed cyclic graphs," in *Proceedings of the 12th international conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1996, pp. 454–461.

[13] D. Marbach, J. C. Costello, R. Küffner, N. M. Vega, R. J. Prill, D. M. Camacho, K. R. Allison, A. Aderhold, R. Bonneau, Y. Chen *et al.*, "Wisdom of crowds for robust gene network inference," *Nature methods*, vol. 9, no. 8, p. 796, 2012.

[14] A. L. Madsen, F. Jensen, A. Salmerón, H. Langseth, and T. D. Nielsen, "Parallelisation of the pc algorithm," in *Conference of the Spanish Association for Artificial Intelligence*. Springer, 2015, pp. 14–24.

[15] ——, "A parallel algorithm for bayesian network structure learning from large data sets," *Knowledge-Based Systems*, vol. 117, pp. 46–55, 2017.

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

[17] T. Le, T. Hoang, J. Li, L. Liu, H. Liu, and S. Hu, "A fast pc algorithm for high dimensional causal discovery with multi-core pcs," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016.

[18] C. Schmidt, J. Huegle, and M. Uflacker, "Order-independent constraint-based causal structure learning for gaussian distribution models using gpus," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*. ACM, 2018, p. 19.

[19] J. Peters, D. Janzing, and B. Schölkopf, *Elements of causal inference: foundations and learning algorithms*. MIT press, 2017.

[20] J. Pearl, *Causality*. Cambridge university press, 2009.

[21] D. Koller, N. Friedman, and F. Bach, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[22] P. Parviainen and S. Kaski, "Learning structures of bayesian networks for variable groups," *International Journal of Approximate Reasoning*, vol. 88, pp. 110–127, 2017.

[23] S. A. Andersson, D. Madigan, M. D. Perlman *et al.*, "A characterization of markov equivalence classes for acyclic digraphs," *The Annals of Statistics*, vol. 25, no. 2, pp. 505–541, 1997.

[24] J. Peters, D. Janzing, and B. Schlkopf, *Elements of Causal Inference: Foundations and Learning Algorithms*. MIT press, 2017.

[25] N. Harris and M. Drton, "Pc algorithm for nonparanormal graphical models," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 3365–3383, 2013.

[26] T. V. J. udea Pearl, "Equivalence and synthesis of causal models," in *Proceedings of Sixth Conference on Uncertainty in Artificial Intelligence*, 1991, pp. 220–227.

[27] C. Chow and C. Liu, "Approximating discrete probability distributions with dependence trees," *IEEE transactions on Information Theory*, vol. 14, no. 3, pp. 462–467, 1968.

[28] D. M. Chickering, "Optimal structure identification with greedy search," *Journal of machine learning research*, vol. 3, no. Nov, pp. 507–554, 2002.

[29] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. Kerminen, "A linear non-gaussian acyclic model for causal discovery," *Journal of Machine Learning Research*, vol. 7, no. Oct, pp. 2003–2030, 2006.

[30] P. O. Hoyer, S. Shimizu, A. J. Kerminen, and M. Palviainen, "Estimation of causal effects using linear non-gaussian causal models with hidden variables," *International Journal of Approximate Reasoning*, vol. 49, no. 2, pp. 362–378, 2008.

[31] D. Rothenhäusler, C. Heinze, J. Peters, and N. Meinshausen, "Backshift: Learning causal cyclic graphs from unknown shift interventions," in *Advances in Neural Information Processing Systems*, 2015, pp. 1513–1521.

[32] C. Heinze-Deml, M. H. Maathuis, and N. Meinshausen, "Causal structure learning," *Annual Review of Statistics and Its Application*, vol. 5, pp. 371–391, 2018.

[33] M. Scutari, C. E. Graafland, and J. M. Gutiérrez, "Who learns better bayesian network structures: Constraint-based, score-based or hybrid algorithms?" in *International Conference on Probabilistic Graphical Models*, 2018, pp. 416–427.

[34] M. Kalisch and P. Bühlmann, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm," *Journal of Machine Learning Research*, vol. 8, no. Mar, pp. 613–636, 2007.

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

[36] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations." [Online]. Available: https://www-users.cs.umn.edu/~saad/software/SPARSKIT/

[37] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[38] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the conference on high performance graphics 2009*. ACM, 2009, pp. 159–166.

[39] 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.

[40] P. Courrieu, "Fast computation of moore-penrose inverse matrices," *arXiv preprint arXiv:0804.4809*, 2008.

[41] Pcalg. [Online]. Available: https://cran.r-project.org/web/packages/pcalg/index.html

[42] ParallelPC. [Online]. Available: http://nugget.unisa.edu.au/ParallelPC/

[43] T. D. Le, L. Liu, J. Zhang, B. Liu, and J. Li, "From mirna regulation to mirna–tf co-regulation: computational approaches and challenges," *Briefings in bioinformatics*, vol. 16, no. 3, pp. 475–496, 2014.

Behrooz Zarebavani received the B.Sc. degree in electrical engineering from Amirkabir University of Technology, Tehran, Iran, in 2017. He is currently working towards the M.Sc. degree in electrical engineering at Sharif University of Technology, Tehran, Iran. His research interests include parallel processing, machine learning, and casual inference.

Foad Jafarinejad is currently working towards the B.Sc. degree in electrical engineering at Sharif University of Technology, Tehran, Iran. His research interests include machine learning, graphical model learning, and parallel computing.

Matin Hashemi received the B.Sc. degree in electrical engineering from Sharif University of Technology, Tehran, Iran, in 2005, and the M.Sc. and Ph.D. degrees in computer engineering from University of California, Davis, in 2008 and 2011, respectively. He is currently an assistant professor of electrical engineering at Sharif University. His research interests include algorithm design and hardware acceleration for machine learning and big data applications.

Saber Salehkaleybar received the B.Sc., M.Sc. and Ph.D. degrees in electrical engineering from Sharif University of Technology, Tehran, Iran, in 2009, 2011, and 2015, respectively. He is currently an assistant professor of electrical engineering at Sharif University of Technology. His research interests include distributed systems, machine learning, and causal inference.