A Parallel Algorithm for Bayesian Network Inference using Arithmetic Circuits

Md. Vasimuddin
Indian Institute of Technology Bombay
Mumbai, MH, India
wasim@cse.iitb.ac.in

Sriram P. Chockalingam
Georgia Institute of Technology
Atlanta, GA, USA
srirampc@gatech.edu

Srinivas Aluru
Georgia Institute of Technology
Atlanta, GA, USA
aluru@cc.gatech.edu

***Abstract*—Exact inference in Bayesian networks is NP-Hard. While many parallel algorithms have been proposed for this irregular problem, none have been shown to scale to even hundreds of processors. In this paper, we present a scalable distributed-memory parallel algorithm for exact inference based on Darwiche's approach, which poses inference as upward and downward accumulation of values computed at the nodes of an arithmetic circuit, a rooted directed acyclic graph. Our work includes parallel algorithms for both construction of the arithmetic circuit as well as inference using the circuit. We demonstrate the scalability of our algorithms for up to 1,536 cores on synthetic as well as real datasets, whose corresponding arithmetic circuits contain up to billions of nodes. The runtime for inference is only a small fraction of the runtime for circuit construction, providing the ability to quickly perform multiple inferences once the circuit is constructed.**

I. Introduction

Given a joint probability distribution of a set of random variables, the task of *probabilistic inference* is to compute the marginal distribution of a subset of variables (called *query variables*) given assignments for another subset of variables (called *evidence variables*). The size of a joint distribution increases exponentially with the number of states the random variables can take because probability values have to be specified for every possible combination of the states. By exploiting the conditional independences present in a joint distribution, Bayesian networks can represent large joint distributions in a compact form by factoring the distribution into a product of conditional distributions. Formally, a Bayesian network over a set of random variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a pair $\mathcal{N} = (\mathcal{G}, \mathcal{D})$, where $\mathcal{G}$ is a directed acyclic graph over the random variables, and $\mathcal{D}$ is the set of conditional probability tables that constitute the joint distribution as $P(\mathcal{X}) = \prod_{i=1}^{n} P(X_i|Pa(X_i))$, where $Pa(X_i)$ is the set of parents of X_i in $\mathcal{G}$. Bayesian networks can represent many complex distributions and have been applied in a diverse range of scientific domains such as medical diagnosis, expert systems, bioinformatics, and data mining.

Inference in a Bayesian network consists of computing marginal distribution of given query variables given assignments for the evidence variables. This is an NP-Hard problem [1], for which numerous algorithms, both exact and approximate, have been proposed over the last few decades (See [2] for a recent review). Of the many approaches developed, inference using junction trees [3, 4] has received significant attention. Junction tree is an auxiliary data structure built from a Bayesian network using tree decomposition of an undirected representation of the network. The size of a junction tree and the runtime for inference is bounded by $\mathcal{O}(nr^w)$, where w is the tree-width (also called the clique-width in Bayesian network literature) and r is the maximum number of states a variable can take. Note that construction of a junction tree having the lowest tree-width is an NP-Complete problem for arbitrary networks [5], and therefore many heuristics have been developed instead of an exact solution. This problem is not the focus of the work presented here.

The expensive computational requirements for inference restricts applicability of Bayesian networks to many practical applications, in which junction trees have a large clique-width w. In the past decade, numerous parallel algorithms have been proposed to scale inference to a large number of variables, specifically using the junction tree algorithm [1, 6–15]. However, almost all of these algorithms are limited in their scalability, due to the difficulties in parallelizing the operations on junction tree vertices, such as finding an appropriate distribution of the cliques of varying sizes among processors. We briefly overview these methods in section II.

In this paper, we propose a parallel algorithm for Bayesian network inference that also uses the junction tree as a starting point. However, unlike previously published works which parallelize the table operations on junction tree vertices, we present a parallel inference algorithm based on Darwiche's approach using *arithmetic circuits* [16], rooted directed acyclic graphs constructed from rooted junction trees. In section III, we discuss in detail Darwiche's arithmetic circuit construction procedure. Inference using the circuit is achieved via upward and downward passes over its structure, similar to that of a junction tree algorithm. Since an arithmetic circuit is built from the corresponding junction tree, the number of nodes in the circuit is also bouned by $O(nr^w)$.

In section IV, we present parallel algorithms to construct a distributed representation of the arithmetic circuit given the junction tree as input, as well as carrying out inference in parallel using the arithmetic circuit. The distributed representation plays a key role in efficient distribution of work among processors during parallel upward and downward passes needed for inference. In section V, we demonstrate inference in Bayesian networks using synthetic as well as real datasets having clique-width as high as 30 variables, translating to billions of nodes in the corresponding arithmetic circuit. We demonstrate the scaling performance of our algorithm from 2 to 1,536 cores

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00014

with parallel efficiency as high as 65% for arithmetic circuit construction. As a consequence of the distributed representation of the circuit, the time taken by our parallel algorithm for the upward and downward passes is a small fraction of the construction time. This is particularly advantageous in practice because arithmetic circuit creation from a Bayesian network via its junction tree is a one-time preprocessing task, while all subsequent inference queries on the Bayesian network can be performed using the same arithmetic circuit. Apart from being an efficient solution for inference, our algorithm can also serve as a basis for developing parallel methods for deep learning models that are generalizations of arithmetic circuits such as the sum-product networks [17].

II. Related Work

Significant works exist in developing parallel algorithms for Bayesian network inference [1, 6–9]. As purely theoretical explorations, Pennock [1] invented a logarithmic-time parallel algorithm on the CREW PRAM model, while D'Ambrosio [6] derived bounds on the available parallelism assuming a broadcast-compute-aggregate model.

In one of the earliest practical implementations for inference, Kozlov and Singh [8] published a parallel version of the Lauritzen and Spiegelhalter algorithm [18], the most popular among exact inference approaches. Their algorithm uses shared address-space on a system with physically distributed memory, and also takes into account the topology of the junction tree and the independent operations in large cliques. They demonstrate a speedup of at most 15X on 32 processors.

In the last decade, several approaches have been proposed to parallelize inference using junction trees and some of them have also focused on accelerators to speed up this compute-intensive task. Namasivayam *et al.* [10] proposed two levels of parallelism – distributed parallelism at the tree level using pointer jumping to propagate tables across cliques, and shared-memory parallelism for intra-clique operations. Using synthetically generated junction trees, they demonstrate scalability up to 256 processors. However the applicability of this algorithm is limited because the compute-intensive node level operations, which can grow as high as $O(r^w)$, are not distributed. Xia *et al.* [11] perform exact inference based on decomposition of junction tree into linear chains, and demonstrate performance on randomly generated junction trees having maximum clique size of ten binary variables on a machine with 64 processors. Xia *et al.* also presented a clique-level parallel algorithm [12] showing scalability up to 128 processors, with restrictive assumptions on the size of separators.

The first known work that used accelerators for inference is reported in [13], where the Cell Broadband Engine was employed for clique-level operations. Recently, Zheng *et al.* [14, 15] describe GPU versions of junction tree inference algorithm. GPUs are targeted to handle compute-intensive parts such as summing out and mapping entries of a large potential table. They show significant improvement over sequential runtime on real networks. However, accelerator implementations are bound by memory and compute capacities and thus are not suitable for datasets having large clique-widths.

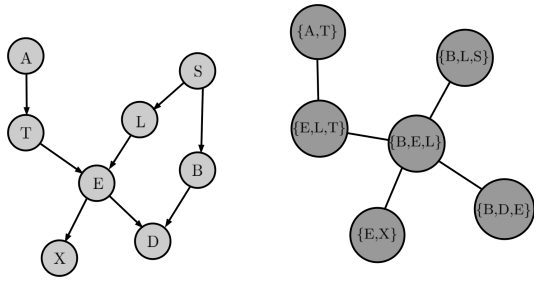


Fig. 1: (a) A BN and (b) its corresponding JT.

III. Background

A Bayesian network (abbreviated BN) is a tuple $(\mathcal{G}, \mathcal{D})$, where $\mathcal{G}$ is a directed acyclic graph (DAG) over n vertices representing random variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ and $\mathcal{D} = \{\Theta_1, \ldots, \Theta_n\}$ is the set of conditional probability tables (abbreviated CPT) associated with the random variables. The structure of a BN conforms to the Markov condition which states that a random variable is independent of all of its non-descendants in $\mathcal{G}$ given its parents in $\mathcal{G}$. BNs essentially depict graphically the factorization of the joint probability distribution over $\mathcal{X}$ as $P(\mathcal{X}) = \prod_{i=1}^{n} P(X_i|Pa(X_i))$, where $Pa(X_i)$ is the set of parents of X_i in $\mathcal{G}$.

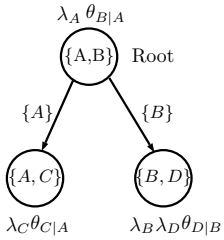
The CPT Θ_i contains conditional probability values for each state of random variable X_i given assignments for every parent state for $Pa(X_i)$. In other words, Θ_i includes one entry for each state of $Scope(\Theta_i) = \{X_i\} \cup Pa(X_i)$. The CPT entry in Θ_i corresponding to the state $X_i = x, Pa(X_i) = u$ is denoted by $\theta_{X_i=x, Pa(X_i)=u}$.

A. Junction Trees

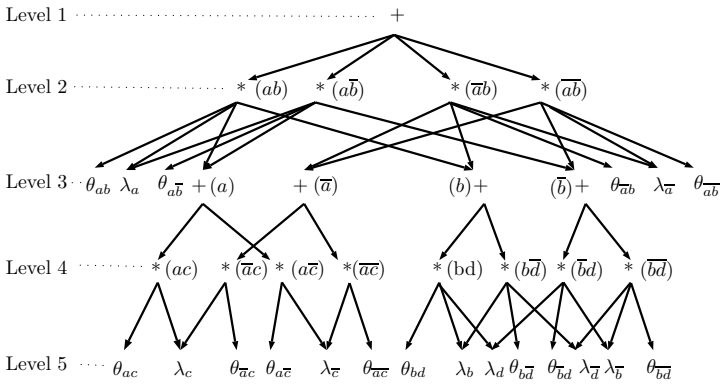
The junction tree (abbreviated JT) is a secondary data structure derived from a BN and can be used to answer inference queries on the network. A junction tree, represented by a tuple $(\mathcal{C}, \mathcal{S})$, is a tree decomposition of the *moralized Bayesian network*, a graph derived from the BN by adding edges between all pairs of nodes that share a common child and making all the edges undirected (See [19] for details). A JT satisfies the following properties:

- Each vertex $C_i \in \mathcal{C}$ in JT is a set of random variables that form a maximal clique in the moralized BN.
- Associated with an edge between C_i and C_j in the JT is a separator set (sepset), $S_{ij} = C_i \cap C_j, S_{ij} \in \mathcal{S}$.
- If $X_i \in C_i$ and $X_i \in C_j$ $(C_i \neq C_j)$, then X_i is contained in all the cliques in the path between C_i and C_j; this property is called the *running intersection property*.
- Each CPT $\Theta_k \in \mathcal{D}$ is assigned to exactly one clique C_i, such that $Scope(\Theta_k) \subseteq C_i$.

Figure 1 shows an example BN and its corresponding JT. We construct JTs from BNs using the popular open source software, UnBBayes [20].



(a) JT for a four variable BN.



(b) AC corresponding to the JT shown on the left.

Fig. 2: A rooted JT and its AC for a BN over four variables $\{A, B, C, D\}$. The two states of the variables A, B, C, and D are denoted by $\{a, \overline{a}\}, \{b, \overline{b}\}, \{c, \overline{c}\}$, and $\{d, \overline{d}\}$ respectively. The vertex assignments for λ and θ variables are shown on the left.

To answer an inference query, the JT is first rooted and then set up in two passes called evidence collection and evidence distribution. Any vertex in JT can be selected as its root. As output, a table of state probabilities called the *potential table* is computed for each JT vertex $C_i \in \mathcal{C}$. The size of the potential table for C_i is the total number of states of C_i, *i.e.,* all possible combinations of states of the variables in C_i (see [19] for further details). The inference query can now be answered by first identifying a vertex C_α of JT containing all the query variables, followed by eliminating the non-query variables from its potential table. Elimination of a set of variables from a potential table is done by adding compatible rows for non-elimination variables. Here, compatibility refers to agreement in the states of the variables.

Solutions to inference queries computed using JT are exact. The space and time complexity of the JT algorithm is $O(|\mathcal{C}|r^w)$, where r is the maximum state space of the variables and w is the clique-width of JT. In other words, the time complexity of inference using JT is linear in its size and exponential in its clique-width.

B. Inference using Arithmetic Circuits

Given a BN and its corresponding rooted JT, Darwiche [16] presented a method to construct an arithmetic circuit and infer probabilistic queries via computations defined on the circuit. An arithmetic circuit (abbreviated AC) is a directed acyclic graph of sum, product, and input nodes, with operations at a node defined in terms of its parent and child nodes. We use SUM and PROD to denote the sum and product nodes, respectively. Input nodes are denoted by λ or θ.

We now fully describe Darwiche's method [16], as our parallel algorithm follows the steps defined by this method. Darwiche's construction procedure defines two types of variables: indicator variables (denoted by λ) corresponding to each state of $X_i, \forall X_i \in \mathcal{X}$, and CPT variables (denoted by θ) corresponding to each entry of CPT $\Theta_i, \forall \Theta_i \in \mathcal{D}$. Each θ variable corresponding to Θ_i is assigned to exactly one vertex C_j such that $Scope(\Theta_i) \subseteq C_j$. Similarly, a λ variable corresponding to X_i is assigned to a vertex containing X_i.

Given a rooted JT $(\mathcal{C}, \mathcal{S})$ and valid vertex assignments for all the θ and λ variables, the corresponding AC is constructed as follows. The AC consists of the following nodes:

- A root SUM node f.
- A PROD node for each state of the vertex $C_i, \forall C_i \in \mathcal{C}$.
- A SUM node for each state of the sepset $S_{ij}, \forall S_{ij} \in \mathcal{S}$.
- A λ input node for each indicator variable, $\lambda_{X_i=x}$.
- A θ input node for each CPT variable, $\theta_{X_i=x,Pa(X_i)=z}$.

The AC includes the following directed edges:

- The root SUM node f has outgoing edges to all the PROD nodes corresponding to the root vertex of JT.
- A PROD node corresponding to a vertex C_j's state has outgoing edges to all the compatible SUM nodes of every S_{ij} such that S_{ij} is a sepset connecting C_j with C_i and C_i is a child of C_j in the rooted JT. Note that compatibility here refers to agreement in the states.
- A SUM node corresponding to a sepset S_{ij}'s sate has outgoing edges to all the compatible PROD nodes of C_i, the child vertex of S_{ij} in the rooted JT.
- The leaf nodes $\lambda_{X_i=x}$ and $\theta_{X_i=x,Pa(X_i)=z}$ have incoming edges from compatible PROD nodes following their JT vertex assignment.

AC, thus constructed, contains alternate levels of SUM and PROD nodes corresponding to the sepset and vertex states of the JT, respectively. Figure 2 shows an example JT and its corresponding AC for a BN of four binary variables.

Given an evidence $X_e = g$, evidence collection and distribution in AC is carried out via upward and downward passes on the circuit. Two values are associated with each node z in AC: an upward pass value $u(z)$ and a downward pass value $d(z)$. During the upward pass, $u(z)$ is computed for all the nodes, starting with the leaf nodes and proceeding towards the root node f. $u(z)$ is computed as follows:

- If z is a $\lambda_{X_e=x}$ node, set $u(z)$ to 1 if it corresponds to the state $X_e = g$, else 0. Also, set $u(\lambda)$ values of the λ nodes corresponding to all non-evidence variables to 1. For example, in Figure 2, if the given evidence is $C = \overline{c}$,

then set $u(\lambda_{\overline{c}}) = 1$ and $u(\lambda_c) = 0$. All other $u(\lambda)$ values are set to 1.
- If z is a θ node, set $u(z)$ to the value from the corresponding CPT entry.
- If z is a PROD node, set $u(z)$ to $\prod_{y \in \mathcal{H}(z)} u(y)$, where $\mathcal{H}(z)$ is the set of child nodes of z.
- If z is a SUM node or root node, set $u(z)$ to $\sum_{y \in \mathcal{H}(z)} u(y)$ with $\mathcal{H}(z)$ being the set of child nodes of z.

In the downward pass, $d(z)$ is computed starting from the root node f as follows:
- At root node, set $d(f)$ to 1.
- If z is a PROD node, set $d(z)$ to $d(\mathsf{parent}(z))$.
- If z is a SUM node or a leaf node (θ or λ), then set $u(z)$ to $\sum_{y \in \mathcal{R}(z)} \frac{d(y) \times u(y)}{u(z)}$, where $\mathcal{R}(z)$ is the set of z's parents.

Darwiche [16] proved that using an AC whose values are updated via upward and downward passes for some evidence $X_e = g$, a query $P(X_q | X_e = g)$ can be answered by computing $P(X_q | X_e = g) = \frac{d(\lambda_{X_q})}{u(f)}$. Since AC is an explicit representation of the operations of inference using JT, the complexity of inference using AC remains the same as the complexity of inference using JT.

IV. Proposed Parallel Algorithm

In this section, we present our parallel algorithm for Darwiche's inference using ACs. In section IV-A, we present a method to select the JT's root so as to minimize the height of the corresponding AC. In section IV-B, we examine the dependency structure of the AC induced by properties of the JT. In section IV-C, we discuss our parallel algorithm to construct a distributed AC with the goal of minimizing the computation costs of parallel upward and downward passes. Section IV-D describes our parallel algorithm for upward and downward passes on a distributed AC.

Let p be the number of processors available in the system. Let N be the size of the AC, *i.e.,* the sum of the number of nodes and edges in the AC. Given a rooted JT, we label the levels of the corresponding AC as $1, \ldots, L$ starting with the root AC node f at level 1. For any node z at level i, the computation of $u(z)$ (resp. $d(z)$) depends only upon nodes at level $i+1$ (resp. $i-1$) during the upward (resp. downward) pass. θ and λ nodes are assigned to the level immediately next to their parent PROD nodes. Note that L is odd because AC has twice the number of levels as that of the JT and an additional level for the leaf nodes connected to level $L-1$ PROD nodes.

We denote $\texttt{PROD}(i, l)$ and $\texttt{SUM}(j, k)$ to represent the l-th PROD node and k-th SUM node at levels i and j respectively. To represent an arbitrary SUM (resp. PROD) node at level i, we use $\texttt{SUM}(i, \cdot)$ (resp. $\texttt{PROD}(i, \cdot)$). We use n_i to represent the total number of nodes at level i, and m_i to represent the total number of edges connecting level i ($i > 1$) to level $i-1$.

A. Root Vertex Selection for the Junction Tree

The correctness of the inference algorithm or the sequential runtime complexity does not depend on the selection of the root vertex. However, root selection is vital for the parallel algorithm because the number of levels in the AC is determined by the choice of the root of the JT. Our goal is to minimize the number of levels of AC and thus increase the number of AC nodes available at each level so that the total available parallelism is maximized.

In our implementation, the root of the JT is selected as follows. We find a pair of vertices (C_i, C_j) in JT such that the distance between C_i and C_j is the diameter of the JT. We select the median vertex C_k on the path between C_i and C_j as the JT's root vertex. The JT with C_k as its root produces a JT with minimum height. Since the height of an AC is a linear function of the height of the corresponding rooted JT, AC constructed with C_k as the root has the least height.

In a tree, a farthest pair of vertices (C_i, C_j) and the median C_k on the path from C_i to C_j can be identified by three breadth-first searches [21]. Therefore, root can be selected in $O(|\mathcal{C}|)$ time. As the size of the JT is significantly smaller than the size of the AC, each processor locally and independently computes the diameter to select the root vertex.

After selecting the JT root, we define an arbitrary left-to-right ordering of vertices for each level of JT and also an ordering of all the n variables of the BN. This in turn defines an ordering of variables for each vertex $C_i \in \mathcal{C}$ and each sepset $S_{ij} \in \mathcal{S}$ of JT that is consistent with the BN variable ordering, leading to an ordering of the vertex states of every $C_i \in \mathcal{C}$ and every $S_{ij} \in \mathcal{S}$. Given the left-to-right ordering for each level of a rooted JT and an ordering of states for each vertex and sepset, any $\texttt{PROD}(i, l)$ or $\texttt{SUM}(j, k)$ node can be initialized or identified in $O(w)$ time using appropriate lookup tables, where w is the clique-width of the JT. In our implementation, we make use of these orderings during the construction of the AC.

B. Structure of the Arithmetic Circuit

Previously published parallel inference algorithms targeted for distributed memory systems [10–12] rely on the assumption that the number of states of any sepset S_{ij} is a constant. This assumption enables these methods to avoid the cost of broadcasting the values corresponding to the sepset states during the passes. However, for both real and randomly generated BNs, this assumption does not hold true. Table I shows that the maximum sepset-width is almost as high as the maximum clique-width in many cases.

Based on the characteristics of JTs of both real and randomly sampled BNs (Table I), we make the following assumption: The maximum degree of a vertex in the JT is bounded by a constant γ. Given that the maximum number of JT vertex states for real datasets ranges from few hundred thousands to millions while the maximum degree remains less than 25, we consider this assumption to be reasonable and is better grounded on real datasets than the restrictive assumption on the number of sepset states. This assumption leads to the following observation on AC's structure.

Observation 1. *If the maximum degree of a rooted* JT *is* γ*, then the number of children for any* PROD *node in the corresponding* AC *is at most* γ.

TABLE I: Various attributes of JTs of real and synthetic datasets.

	Real datasets						Synthetic datasets				
Datasets/Attributes	Water	Diabetes	Barley	Munin2	Munin3	Munin4	D1	D2	D3	D4	D5
Number of variables	32	413	48	1003	1044	1041	160	160	50	40	170
Max. state space of $\mathcal{X}$	4	21	67	21	21	21	2	2	2	2	2
Number of vertices	19	337	36	869	906	876	130	128	21	13	136
Max. no. of vertex states	5,308,416	190,080	13,063,680	196,000	156,800	1,000,000	2^{28}	2^{29}	2^{30}	2^{28}	2^{31}
Max. clique-width	12	6	8	8	8	9	28	29	30	28	31
Avg. clique-width	6.37	4.17	4.83	3.42	3.35	3.63	6.17	6.78	16.38	21.31	6.77
Max. sepset-width	10	4	7	7	7	8	23	24	27	27	25
Avg. sepset-width	4.94	2.96	3.6	2.27	2.20	2.44	4.98	5.58	14.7	19.75	5.5
Max. degree	4	3	5	18	22	11	3	3	8	12	3

The above observation follows directly from the AC construction procedure described in section III-B. A PROD node corresponding to a state of a JT vertex C_j has outgoing edges to all the compatible SUM nodes of C_j's child JT sepsets. Since the degree of C_j is bounded by γ, a PROD node of C_j can have at most γ child SUM nodes.

An interesting consequence of the AC construction procedure is that it induces a specific dependency structure among the operations defined on the nodes of the AC. By definition, nodes at a level i of the AC have edges only to nodes at levels $i+1$ or $i-1$, but not among themselves. Therefore, the subgraph induced by the nodes in any two consecutive levels of an AC is a bipartite graph (Figure 2 shows an example). However, when i is odd, this structure reduces to a tree, which can be exploited for efficient parallel computations.

Lemma 1. *Let i ($1 \leq i < L$) be odd. In the AC corresponding to a rooted JT, the subgraph induced by the nodes of two consecutive levels i and $i+1$ is a forest.*

Proof. When i is odd, level i consists only of SUM nodes, whose children are the compatible PROD nodes at level $i+1$. By construction, there is no direct edge between the variable nodes (*i.e.,* θ or λ nodes) and the SUM nodes. Since any $\texttt{PROD}(i+1,\cdot)$ node corresponds to a vertex C_k in JT and the $\texttt{SUM}(i,\cdot)$ node corresponds to the sepset connecting C_k to its parent in JT, the state of a $\texttt{PROD}(i+1,\cdot)$ node can be compatible with the state of at most one SUM node upstream. Thus, a PROD node can be connected to at most one $\texttt{SUM}(i,\cdot)$ node upstream. Therefore, the induced subgraph is a forest of trees, where each tree is rooted at a $\texttt{SUM}(i,\cdot)$ node, which is connected to compatible child $\texttt{PROD}(i+1,\cdot)$ nodes. □

C. Parallel Construction of the Distributed AC

After computing the root vertex of the JT, we proceed to construct a distributed representation of the AC in parallel. The primary goal of such a representation is to enable efficient distributed computation of the circuit values during parallel upward and downward passes. We first present a distributed AC representation that enables efficient computation during the upward pass, and then later show that this scheme works just as well for the downward pass also.

The upward pass circuit values at the odd levels of the AC (*i.e.,* at SUM nodes) can be computed in parallel as follows.

Lemma 2. *Let $i, 1 \leq i < L$ be odd. During the upward pass in a distributed AC constructed from a rooted JT, all the $u(\texttt{SUM}(i,\cdot))$ values at level i can be computed in parallel by a segmented parallel prefix operation.*

Proof. Let n_i be the total number of nodes at level i. In the upward pass on the AC, the computation at a $\texttt{SUM}(i,\cdot)$ node is to add the $u(\cdot)$ values corresponding to all its child $\texttt{PROD}(i+1,\cdot)$ nodes. By Lemma 1, the induced graph between the i and $i+1$ levels of the AC is a forest. To compute all the $u(\texttt{SUM}(i,\cdot))$ values in parallel, first, all the $\texttt{PROD}(i+1,\cdot)$ nodes that share a common parent $\texttt{SUM}(i,\cdot)$ node are stored contiguously such that each processor has at most $\left\lceil \frac{n_{i+1}}{p} \right\rceil$ nodes. Then, a *segmented parallel prefix* operation [22] with addition operator can compute all the $u(\texttt{SUM}(i,\cdot))$ values in parallel. □

Lemma 2 suggests that all the n_{i+1} PROD nodes at level $i+1$ be distributed such that all sibling PROD nodes are contiguously placed. PROD nodes at a level $i+1$ can be laid out in the order corresponding to the left-to-right ordering of the vertices in the JT. As each processor has a copy of the JT, a processor can compute its range of $O(n_{i+1}/p)$ PROD nodes without incurring any communication costs. As the parent node may not be located in the same processor (the location of the parent SUM node is discussed later in this section), the last one of the PROD siblings stores a remote link to its parent SUM node. We call this PROD node the *designated* PROD node.

While Lemma 2 provides an efficient way to compute the circuit values at odd levels during an upward pass, distributed computing of the $u(\cdot)$ values at the even levels is more challenging. In the latter case, the difficulty lies in the nature of the dependency structure. The generic bipartite dependency structure defies an elegant solution similar to that of the odd levels. We propose a distribution scheme for the AC nodes that enables efficient computation at the cost of collective communication rounds.

The computation cost at a PROD node during the upward pass is constant because by observation 1, a PROD node has $\leq \gamma$ child nodes and γ is a constant. Therefore, the distributed computation cost at an even level $i-1$ can be bounded to $O(n_{i-1}/p)$ by limiting the number of PROD nodes per processor to $O(n_{i-1}/p)$. However, the key challenge here is to define an appropriate distribution of their child SUM nodes.

A $\texttt{SUM}(i,\cdot)$ node has $|\mathcal{R}(\texttt{SUM}(i,\cdot))|$ different PROD nodes as its parent. If SUM nodes of a level are uniformly distributed across p processors, a $\texttt{SUM}(i,\cdot)$ node that is connected to a large number of $\texttt{PROD}(i-1,\cdot)$ nodes will skew the cost during the collective communication round, leading to poor scalability. In order to avoid such an imbalance, the distribution scheme should make sure that during the communication round, which sends the $u(\texttt{SUM}(i,\cdot))$ values to their parent $\texttt{PROD}(i-1,\cdot)$ nodes, the total number of incoming and outgoing elements for a processor is bounded. One way to accomplish this is to distribute across the processors the total number of connections out of the odd level i to its parent level $i-1$, *i.e.,* distribute the $\sum_{k=1}^{n_i}|\mathcal{R}(\texttt{SUM}(i,k))| = m_i$ upstream edges – with each edge holding a copy of the corresponding $\texttt{SUM}(i,\cdot)$ node value.

Lemma 3. *Let $i, 1 < i < L$ be odd. There exists a distributed layout of all the $\texttt{SUM}(i,\cdot)$ and $\texttt{PROD}(i-1,\cdot)$ nodes such that the $u(\texttt{SUM}(i,\cdot))$ node values can be retrieved by their parent $\texttt{PROD}(i-1,\cdot)$ nodes by a collective communication round in which the number of incoming and outgoing elements for a processor is bound to $O\left(\frac{m_i}{p}\right)$, where $m_i = \sum_{k=1}^{n_i}|\mathcal{R}(\texttt{SUM}(i,k))|$.*

Proof. The $\texttt{PROD}(i-1,\cdot)$ nodes are distributed as per lemma 2. By observation 1, each PROD node requires $\leq \gamma$ child node values. Since each processor has at most $\left\lceil\frac{n_{i-1}}{p}\right\rceil$ PROD nodes and $n_{i-1} \leq m_i$, the total number of incoming elements for a processor is $O\left(\frac{m_i}{p}\right)$.

In the case of SUM nodes, all the $\sum_{k=1}^{n_i}|\mathcal{R}(\texttt{SUM}(i,k))| = m_i$ edges going upstream out of an odd level i – each carrying a copy of the corresponding SUM node value – are distributed across p processors such that each processor has at most $\left\lceil\frac{m_i}{p}\right\rceil$ elements. Hence, the number of outgoing elements for a processor is also $O\left(\frac{m_i}{p}\right)$. □

In other words, to avoid imbalance, Lemma 3 defines a distributed layout in which a SUM node is duplicated as many times as the number of parents it has. All the SUM duplicates are placed contiguously. We call the SUM node that appears last among its duplicate siblings as the *designated* SUM node.

The proposed distributed representation of AC at even and odd levels follows Lemma 3. Acting as the transfer point between the even and odd levels are the *designated* SUM and PROD nodes. For example, during the upward pass, the *designated* child $\texttt{PROD}(i+1,\cdot)$ node communicates the $u(\cdot)$ computed for its parent SUM node to the *designated* $\texttt{SUM}(i,\cdot)$. Further details are given in the algorithm description in section IV-D.

The distributed layout for AC proposed above is also applicable for the downward pass because both the passes have similar computational dependencies. Specifically, during the downward pass, the computation at a SUM node is to add all values locally computed at their parent PROD nodes, which can be accomplished by a collective communication round from parent PROD nodes to their corresponding SUM duplicates, followed by a segmented parallel prefix operation. Similarly, the computation at a PROD node is to copy its parent SUM node's $d(\cdot)$ value, which can be accomplished by a segmented parallel prefix operation.

Algorithm 1: Parallel Arithmetic Circuit Construction

Input: BN $(\mathcal{G},\mathcal{D})$ and its JT $(\mathcal{C},\mathcal{S})$.
Output: Distributed AC.

1 **parallel** j = *processor's rank* **do**
2 Select the root of the JT.
3 Create λ and θ leaf nodes locally.
4 **for** $t \leftarrow 1$ **to** $\frac{(L-1)}{2}$ **do** // Even Levels of AC
5 $V \leftarrow$ left-to-right order of vertices at JT level t.
6 $n_{2t} \leftarrow \sum_{C_i \in V}$ No. of states of C_i.
7 $(lx, rx) \leftarrow \left(\left\lfloor\frac{n_{2t}j}{p}\right\rfloor + 1, \left\lfloor\frac{n_{2t}(j+1)}{p}\right\rfloor\right)$.
8 Construct AC level $2t$: Nodes $\texttt{PROD}(2t, lx)$ to $\texttt{PROD}(2t, rx)$.
9 Identify the *designated* PROD nodes in proc. j.
10 **for** $t \leftarrow 1$ **to** $\frac{(L-1)}{2} - 1$ **do** //Odd Levels of AC
11 $V \leftarrow$ left-to-right order of vertices at JT's level t.
12 $m_{2t+1} \leftarrow \sum_{C_i \in V} |\text{children}(C_i)| \times$ No. of states of C_i.
13 $(ly, ry) \leftarrow \left(\left\lfloor\frac{m_{2t+1}j}{p}\right\rfloor + 1, \left\lfloor\frac{m_{2t+1}(j+1)}{p}\right\rfloor\right)$.
14 Construct AC level $2t+1$ *i.e.,* node duplicates from $\texttt{SUMD}(2t+1, ly)$ to $\texttt{SUMD}(2t+1, ry)$.
15 Identify the *designated* SUMD nodes in proc. j.
16 **for** $i \leftarrow 2$ **to** $L-1$ **do** // Edge Pointers
17 Create edge pointers to remote parent/child nodes
18 **if** j = *0* **then** // Root Node f
19 Create f, the root node of AC.
20 Add edge pointer from f to the *designated* $\texttt{PROD}(2,\cdot)$.

Our parallel AC construction procedure is summarized in Algorithm 1. Each processor, after loading its copy of the BN and its JT, selects JT's root as described in section IV-A. Next, a local copy of the λ and θ leaf nodes is created. Construction of the distributed AC follows next. Lines 4–9 describe the construction of even levels where the PROD nodes are laid out as per Lemma 2, whereas lines 10–15 follow the layout suggested by Lemma 3 to construct the odd levels, *i.e.,* the SUM node duplicates. We use SUMD to denote the SUM duplicates. Similar to the PROD and SUM nodes, $\texttt{SUMD}(i,k), 1 \leq k \leq m_i$ can also be initialized or identified in $O(w)$ time with appropriate lookup tables. In our implementation, we create an explicit duplicate only when the duplicates corresponding to a SUM node cross processor boundaries. Lines 16–17 create the *edge pointers*, which are addresses of remotely located parent/child nodes. While these pointers are an extra overhead on the available memory and can be created later during the passes, they enable faster runtime of the passes. Finally, the root node f is constructed at processor 0 in lines 18–20. Since each processor can compute the range of PROD nodes and

SUMD nodes and their parent/child locations, there is no communication cost involved during the construction. The total computation cost is $O\left(|\mathcal{C}| + |\mathcal{D}| + \sum_{i=2}^{L}\left(\frac{n_i w}{p} + \frac{m_i w}{p}\right)\right) = O\left(|\mathcal{C}| + |\mathcal{D}| + \frac{Nw}{p}\right)$.

D. Parallel Upward and Downward Passes

Algorithm 2: Parallel Upward Pass

Input: Distributed AC and evidence $X_e = g$.
Output: Distributed AC w. upward pass values.

1 **parallel** j = *processor's rank* **do**
2 Initialize input node values and set $u(\lambda_{(X_e=g)}) = 1$.
3 Compute $u(\cdot)$ at $L-1$ level PROD nodes.
  // Upward Pass from levels $L-2$ to 2
4 **for** $i \leftarrow L-2$ **down to** 2 **do**
5 **if** i *is odd* **then**
6 Compute $u(\texttt{SUM}(i,\cdot))$ and save it at the designated PROD nodes (via segmented parallel prefix over $u(\texttt{PROD}(i+1,\cdot))$ with addition operator).
7 Communicate $u(\texttt{SUM}(i,\cdot))$ to the designated $\texttt{SUMD}(i,\cdot)$ nodes (via an all-to-all).
8 Copy $u(\cdot)$ from the designated $\texttt{SUMD}(i,\cdot)$ nodes to all of its duplicates (via segmented parallel prefix with copy operator).
9 **else**
10 Retrieve $\texttt{PROD}(i,\cdot)$ nodes' child $\texttt{SUMD}(i+1,\cdot)$ nodes' $u(\cdot)$ values (via an all-to-all).
11 Compute $u(\texttt{PROD}(i,\cdot))$ as the product of retrieved values and leaf nodes' $u(\cdot)$ values.
  // Upward Pass at Root
12 Compute $u(f)$ (via parallel prefix with addition operator from level 2 PROD nodes).

As described in section III-B, inference using AC consists of an upward pass followed by a downward pass, for which we present parallel algorithms here. As a first step in the upward pass (Algorithm 2), input nodes are initialized (line 2) and $u(\texttt{PROD}(L-1,\cdot))$ values are evaluated (line 3). Subsequently, $u(\cdot)$ is computed at odd and even levels as per Lemmas 2 and 3 respectively, with additional steps to transfer the results via the designated locations. Lines 5–8 describe the accumulation over the forest at odd levels followed by the communication of these results, taking two steps: First, the accumulated values are sent from the designated PROD nodes to their parent SUMD designates via collective communication (*i.e.,* all-to-all) and then, it is propagated to the respective SUMD duplicates. Lines 9–11 list the steps of computation at even levels, which consist of a collective communication round (*i.e.,* all-to-all) from the SUMD nodes to their parent PROD nodes followed by local multiplication operations of $u(\cdot)$ values. Finally, in line 12, we compute $u(f)$ via parallel prefix.

During the downward pass (Algorithm 3), we first set $d(f)$ to 1. Then, the computations at odd and even levels follow Lemmas 3 and 2 respectively. Lines 4–6 show the pseudocode

Algorithm 3: Parallel Downward Pass

Input: Distributed AC w. upward pass values for $X_e = g$.
Output: Marginal probability $P(X_q = h|X_e = g)$.

1 **parallel** j = *processor's rank* **do**
  // Downward Pass for levels $< L$
2 Set $d(f) \leftarrow 1$ (default value).
3 **for** $i \leftarrow 2$ **to** $L-1$ **do**
4 **if** i *is odd* **then**
5 Retrieve to the $\texttt{SUMD}(i,\cdot)$ nodes their parent $\texttt{PROD}(i-1,\cdot)$ nodes' $d(\cdot) \times u(\cdot)$ values (via an all-to-all).
6 Compute $d(\texttt{SUM}(i,\cdot))$ at the designated $\texttt{SUMD}(i,\cdot)$ nodes (via segmented parallel prefix with addition operator).
7 **else**
8 Communicate designated $\texttt{SUMD}(i-1,\cdot)$ nodes' $d(\cdot)$ values to child designated $\texttt{PROD}(i,\cdot)$ nodes (via an all-to-all).
9 Copy $d(\texttt{PROD}(i,\cdot))$ to all its siblings (via segmented parallel prefix with copy operator).
  // Downward pass at the leaf nodes
10 Compute partial $d(\cdot)$ values for leaf nodes (from locally accessible parent nodes).
11 Accumulate leaf nodes' $d(\cdot)$ values (via all-reduce with addition operator).
  // Compute Marginal Probability
12 $P(X_q = h|X_e = g) = \frac{d(\lambda_{X_q=h})}{u(f)}$.

for the computation at odd levels, where the $d(\cdot)$ values at parent $\texttt{PROD}(i-1,\cdot)$ nodes are retrieved by the corresponding $\texttt{SUMD}(i,\cdot)$ node duplicates, and a segmented parallel prefix is used to compute the $d(\texttt{SUM}(i,\cdot)$ values at the designated SUMDs. Lines 7–9 list the steps for the downward computation at even levels, which consist of communication from the designated $\texttt{SUMD}(i-1,\cdot)$ nodes to the designated $\texttt{PROD}(i,\cdot)$ nodes, followed by copying the $d(\cdot)$ value from the designated PROD nodes to all its siblings. Finally, each processor computes the partial $d(\cdot)$ values for the leaf nodes from their locally accessible parents, and then an all-reduce operation adds the partially computed values present in all the processors.

During both upward and downward passes, the computations at level i take $O((n_i/p) + (m_i/p))$ time, and hence the total computation cost is $O\left(\frac{N}{p}\right)$. Communication costs for both these passes are dominated by $O(L)$ rounds of all-to-all operations, with the number of incoming and outgoing elements per processor during each round bounded as per Lemma 3.

V. Experimental Results

We demonstrate the performance of our parallel algorithm on a wide range of real and synthetic datasets. We implemented the algorithm in C++ using the MPI programming model. All the experiments were conducted on a cluster having

TABLE II: Properties of ACs constructed for real and synthetic datasets (Level size in millions).

Dataset	No. of	PROD Level Size		SUM Level Size	
	Levels	Avg.	Max.	Avg.	Max.
Water	10	1.61	5.35	0.1137	0.4442
Diabetes	56	0.37	0.51	0.0209	0.0315
Barley	14	3.70	13.27	0.3029	1.6401
Munin2	28	0.28	1.40	0.0383	0.2786
Munin3	28	0.12	1.20	0.0383	0.1803
Munin4	26	1.25	6.27	0.1478	1.0201
D1	18	34.60	268.63	1.21	8.52
D2	18	75.89	536.94	2.29	17.82
D3	12	250.47	1,140.85	34.47	138.71
D4	12	188.27	285.74	71.71	142.87
D5	20	243.76	2,147.50	5.70	33.56

64 compute nodes, with each node having two 2.4 GHz 14-Core Intel E5-2680 V4 processors and 256 GB of main memory, and running RHEL Linux 6.7 operating system. The nodes are connected with a 40Gbit QDR Infiniband interconnect. We use Intel Composer XE 2015 and Intel MPI 5.1 library to compile and run our code. We use a single MPI process on each core, except for large synthetic datasets (*D3*, *D4*, and *D5* in Table IV) with 2–16 cores, where we use two compute nodes due to memory limitations. All the experiments were conducted three times and average runtimes are reported. We observed a variation of $< 5\%$ on the reported runtimes.

A. Performance Results on Real and Synthetic Datasets

We used both real and synthetic datasets in our experiments. While real datasets are used to show applicability of our algorithm for practical use-cases, the synthetic datasets are crafted to test the scalability of our algorithm. The real datasets used in our experiments were acquired from a popular on-line repository [23], which includes BNs collected from a wide range of applications. We used the UnBBayes software [20] to build JTs from the BNs. Table I lists the key properties of BNs used in our experiments and their corresponding JTs. These characteristics should be kept in mind in interpreting the results presented in this section. Computational needs for inference depend upon the state space of the random variables and the density of the BN, which translates to the state space of the clique variables and the clique-width of JT. Among the real datasets, *Barley* dataset has the largest state space for a variable and maximum state space of a vertex. Also, the AC corresponding to the *Barley* BN contains the fattest level with ≈ 13 million nodes, followed by *Munin4* (Table II).

In many applications such as modeling co-evolution of viral populations [24], researchers are restricted to JTs with smaller tree-width (and hence BNs with fewer edges) in order to avoid the computational costs of exact inference. To demonstrate the scalability of our parallel algorithm to BNs with larger tree-width JTs and to show the ability of our proposed methods to overcome size limitations of current methods, we generated larger synthetic datasets as follows: sample 100 networks from a distribution of random BNs using bnlearn [25] software, construct their JTs and select from this sample five networks that are closer to networks for real datasets w.r.t. characteristics such max. JT degree and ratio of vertex size to separator size. We set the state space of all random variables to be binary to avoid explosion in state space. Synthetic datasets, thus generated, have potential tables with hundreds of millions of entries. The last five columns of Table I describe these synthetic datasets in the order of increasing complexity. Table II reports the characteristics of the corresponding ACs.

Runtime for different stages of the parallel algorithm on real and synthetic datasets are reported in Tables III and IV respectively. Construction of AC dominates the total runtime for all datasets. Even though the scaling of the upward and downward passes is not as effective as AC construction, the runtime for the passes is a small fraction of the AC construction time, contributing to the overall scalability. Among the real datasets, *Munin4* dataset takes the longest time for AC construction, due to the presence of a large number of cliques, followed by *Barley*. Among the synthetic datasets, *D5* takes the longest due to its tree-width.

Figure 3 shows the relative speedups for AC construction for real datasets from $p = 2$ to $p = 64$. Most of the datasets scale well as the number of processor cores increases from 2 to 64. *Munin4* reports highest relative speedup with parallel efficiency of 66%. Except for *Barley* and *Munin3*, all datasets report a relative speedup with parallel efficiency of at least 65%. For *Munin3* and *Barley*, it goes down to 55.27% and 39.41%, respectively. This drop in parallel efficiency is due to the imbalance in computations among processor cores, stemming from two factors: a) differences in the number of leaf nodes per PROD node, and b) differences in the number of children for PROD nodes. Recall that PROD nodes are uniformly distributed among p processors, and hence variations in the number of children for cliques within the JT result in uneven number of child links from PROD nodes. Examination of the AC's structure revealed that while the former factor has a significant impact on *Barley* dataset, the latter one affects performance on the *Munin3* dataset. The latter factor is also a reason for the relatively poor scaling of the upward and downward passes since it leads to an imbalance across processors during the all-to-all operations. Although this drawback can be addressed by PROD node duplication (similar to that of the SUM node duplication), the runtimes for these passes do not necessitate such an effort. Since AC construction is a one time

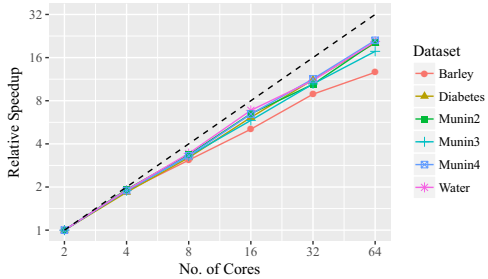


Fig. 3: Relative speedup of parallel algorithm for AC construction on real datasets upto 64 cores.

TABLE III: Runtime (in seconds) for AC creation (T_{AC}), upward pass ($T_{\uparrow}$), and downward pass ($T_{\downarrow}$) for real datasets.

Dataset → / Cores↓	Water			Diabetes			Barley			Munin2			Munin3			Munin4		
	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$
2	37.74	0.30	0.30	60.60	0.16	0.17	189.41	1.20	1.20	75.94	0.16	0.17	56.57	0.17	0.16	329.41	0.47	0.50
4	19.75	0.19	0.18	32.98	0.10	0.10	101.13	0.76	0.77	39.57	0.12	0.12	30.02	0.13	0.11	174.37	0.30	0.30
8	11.01	0.11	0.10	18.90	0.07	0.06	61.53	0.44	0.44	22.59	0.08	0.08	17.20	0.08	0.07	99.70	0.24	0.24
16	5.53	0.07	0.06	9.88	0.05	0.04	37.39	0.27	0.26	11.73	0.05	0.05	9.68	0.06	0.05	51.03	0.20	0.17
32	3.48	0.06	0.05	5.44	0.04	0.03	21.35	0.23	0.21	7.31	0.04	0.04	5.49	0.04	0.04	29.03	0.17	0.13
64	1.82	0.04	0.03	3.01	0.03	0.02	15.01	0.15	0.15	3.70	0.03	0.03	3.18	0.03	0.03	15.59	0.09	0.07

TABLE IV: Runtime (in seconds) for AC creation (T_{AC}), upward pass ($T_{\uparrow}$), and downward pass ($T_{\downarrow}$) for synthetic datasets.

Dataset → / Cores↓	D1			D2			D3			D4			D5		
	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$	T_{AC}	$T_{\uparrow}$	$T_{\downarrow}$
2	1,540.62	14.02	12.75	3,283.23	39.07	35.93	9,577.14	86.83	80.63	8,016.63	62.40	62.36	19,225.12	139.42	99.02
4	814.71	8.54	7.35	1,718.01	22.05	20.63	4,961.57	42.48	39.20	4,988.34	35.59	34.75	9,607.82	72.17	51.48
8	439.42	4.60	3.90	941.13	11.18	10.13	2,683.94	25.53	22.89	2,523.94	19.64	18.89	4,876.59	37.23	26.83
16	225.20	2.74	2.38	480.19	6.02	5.22	1,588.43	13.69	12.35	1,371.33	11.45	11.09	2,688.12	20.83	15.14
32	139.71	1.93	1.62	297.29	4.27	3.70	823.35	11.38	10.19	705.55	10.52	9.51	1,396.51	17.16	13.51
64	70.96	1.22	1.08	152.99	2.69	2.36	413.53	6.91	6.34	357.46	6.23	5.81	705.48	9.96	8.24
128	35.96	0.69	0.61	77.35	1.53	1.38	209.92	4.04	3.62	185.04	3.49	3.35	358.76	6.20	5.18
256	18.66	0.48	0.46	39.97	0.95	0.88	107.41	2.52	2.27	96.21	2.14	2.01	182.89	3.79	3.26
512	9.77	0.31	0.33	21.54	0.62	0.55	55.59	1.59	1.43	51.05	1.23	1.26	96.27	2.12	1.90
1,024	5.47	0.18	0.16	11.71	0.41	0.36	29.78	0.88	0.87	28.09	0.69	0.66	52.33	1.26	1.22
1,536	4.12	0.13	0.09	9.07	0.31	0.26	21.19	0.67	0.66	20.43	0.57	0.49	38.41	1.08	1.03

task and the time taken by the upward and downward passes is relatively small, after constructing the AC on more processors, these passes can be run using fewer processors. This can be accomplished by rewiring the edge pointers, as necessary.

Relative speedups for AC construction for the five synthetic datasets are shown in Figure 4. All the datasets scale well from 2 up to 1,536 cores, with the highest and lowest achieved parallel efficiencies of 64.8% (*D5*) and 47.5% (*D2*), respectively. We also see a proportional decrease in runtime of upward and downward passes as the number of cores increases, though their scaling is not as effective as that of AC construction due to the same issue of variation in the number of children of PROD nodes mentioned previously.

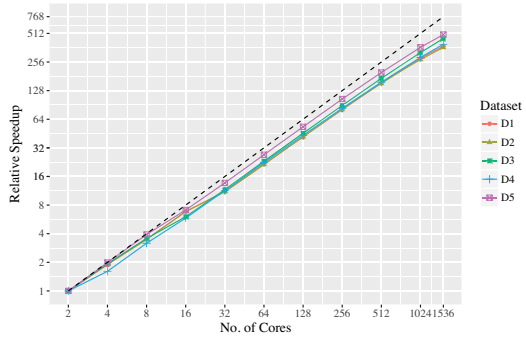


Fig. 4: Relative speedup of our parallel AC construction algorithm on synthetic datasets.

B. Comparison with Previous Work

For comparison against previous state-of-the-art, we consider two methods developed for parallel JT inference in the distributed-memory setting, both by Xia *et al.* [11] [12]. In [11], the vertices of a JT are randomly and equally distributed among p processors, irrespective of their state sizes. Also, the complete potential table of a JT vertex is placed within a single processor. Therefore, this method scales well only when the JT vertices are small enough to fit within a compute node and are approximately of the same size.

In their subsequent work [12], the authors address the above limitations by distributing the potential tables and their operations. The entries of a potential table ψ_C of a JT vertex C are distributed in proportion to its size $|\psi_C|$. However, all the assigned processors keep a local copy of the complete set of sepset tables. After the distribution of JT's vertices among p processors, distributed evidence collection and distribution is performed, equivalent to the upward and downward passes on the AC approach, respectively.

We analyze the shortcomings of the approach [12] by reproducing the following crucial but scalability-limiting step of their distributed evidence collection algorithm. After updating a local sepset table ψ_S by marginalizing the local entries of a distributed ψ_C, each processor broadcasts its copy of $|\psi_S|$ to all other processors, after which all the updates are accumulated locally. During this broadcast step, p different copies of ψ_S are being communicated across the network. In their performance evaluation, Xia *et al.* [12] do not observe poor scaling in this step because they use small values for $|\psi_S|$, within the range $[2, 16]$.

We implemented this step of broadcasting ψ_S tables and

TABLE V: Runtime (in seconds) of a single pass of broadcast step of evidence collection in [12] for *Munin4*, *Barley* and *D1*.

Cores/Datasets	Munin4	Barley	D1
16	0.01	0.10	0.80
32	0.05	0.36	2.97
64	0.38	1.79	14.24

TABLE VI: Runtime and memory consumption of inference using ACE software and the proposed parallel algorithm.

Dataset	ACE Time (s)	ACE Mem. (GB)	Parallel Time (s) (Cores)	Parallel Mem. (∼GB)
Barley	12.85	5.18	2.41 (2)	4
Munin2	1.58	1.24	0.33 (2)	1
Munin4	1.71	1.41	0.97 (2)	2
D1	37.01	15.91	26.75 (2)	56
D2	341.39	44.97	75.01 (2)	120
D3	–	–	167.45 (2)	295
D4	–	–	124.76 (2)	262
D5	–	–	238.44 (2)	430

ran it on three of the JTs listed in Table I – two of the largest real datasets, *Munin4* and *Barley*, and the smallest of the synthetic datasets, *D1*. We identified, for a JT, the largest collection of sepsets whose potential tables will be simultaneously broadcast during the evidence collection algorithm of [12] and broadcast them. Table V reports the runtimes incurred by this operation and it clearly shows that this step, and hence the algorithm, scales poorly when applied to both real and synthetic datasets. Also, when $p = 64$, we note that the runtime for this single collective communication step takes many times longer than the entire upward pass of our algorithm for all the datasets.

Memory demands of exact inference present as significant a challenge as its compute demands. To demonstrate this, we compare our implementation against ACE [26], a serial implementation of the AC inference algorithm. ACE uses different methodology to construct an AC and as a result the equivalent JT might be completely different. Therefore, given an input BN and set of queries, we compare the two approaches only based on query execution times and memory consumed. Results, listed in Table VI, shows that our parallel algorithm achieves significant speedups over ACE for large real and synthetic datasets. For large real datasets, the proposed parallel algorithm with 2 cores runs 1.8–5.33X times faster than the ACE. As the size of the datasets increases, the parallel algorithm handles both the compute and memory requirements quite well, whereas ACE exceeds the memory available for a node. This renders the serial ACE solution infeasible for datasets as big as *D3*, *D4*, or *D5*, as evident from Table VI. Due to the memory-overhead required for the pointers to remote parent and child AC nodes, the total memory usage of the parallel implementation is upto 3.5X higher than ACE.

VI. Conclusions

In this paper, we presented a parallel algorithm for inference in BNs using ACs instead of the traditional approach of parallelizing JT operations. Instead of using JT, which is challenging to parallelize because of the difficulty in uniformly distributing irregular cliques across processors, we use ACs, where operations on the JTs are decomposed down to atomic binary operations on variables, enabling efficient work allocation among processors. Through efficient distribution of the circuit computations, we demonstrated scalable inference for BNs with corresponding JT clique-widths as high as 30. The primary contribution of our work is enabling inference on large BNs that cannot be handled by any sequential as well as available parallel algorithms. We have shown that our parallel algorithm is able to solve JTs containing hundreds of millions of entries in their tables. The parallel algorithm is scaling well not only for real datasets collected from a diverse range of applications, but also for even larger synthetic datasets containing billions of nodes in the AC.

Our parallel algorithm can be extended to accelerate deep learning models such as sum-product networks [17], which we leave for future work. Another potential direction for future research is parallelization of algorithms to construct compact ACs for inference (e.g. [26]).

References

[1] D. Pennock, "Logarithmic time parallel Bayesian inference," in *UAI'98*, 1998, pp. 431–438.

[2] J. Domingos, J. Filipe *et al.*, "Brief survey on computational solutions for Bayesian inference," in *Proceedings of the UCBI Workshop*, 2015.

[3] P. P. Shenoy and G. Shafer, "Propagating belief functions with local computations," *IEEE Expert*, vol. 1, no. 3, pp. 43–52, 1986.

[4] F. Jensen, S. Lauritzen *et al.*, "Bayesian updating in recursive graphical models by local computation," *Computational Statistics Quarterly*, vol. 4, pp. 269–282, 1990.

[5] S. Arnborg, D. G. Corneil *et al.*, "Complexity of finding embeddings in ak-tree," *SIAM JADM*, vol. 8, no. 2, pp. 277–284, 1987.

[6] B. D'Ambrosio, T. Fountain *et al.*, "Parallelizing probabilistic inference: Some early explorations," in *UAI'92*, 1992, pp. 59–66.

[7] A. V. Kozlov and J. P. Singh, "Parallel implementations of probabilistic inference," *IEEE Computer*, vol. 29, pp. 33–40, 1996.

[8] A. V. Kozlov and J. P. Singh, "A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference," in *SC'94*, 1994, pp. 320–329.

[9] R. D. Shachter, S. K. Andersen *et al.*, "Global conditioning for probabilistic inference in belief networks," in *UAI'94*, 1994, pp. 514–522.

[10] V. Namasivayam and V. K. Prasanna, "Scalable parallel implementation of exact inference in Bayesian networks," in *IPDPS*, 2006, pp. 143–150.

[11] Y. Xia and V. K. Prasanna, "Junction tree decomposition for parallel exact inference," in *IPDPS*, 2008, pp. 1–12.

[12] Y. Xia and V. K. Prasanna, "Scalable node-level computation kernels for parallel exact inference," *IEEE TC*, vol. 59, no. 1, pp. 103–115, 2010.

[13] Y. Xia and V. K. Prasanna, "Parallel exact inference on the Cell Broadband Engine processor," in *SC'08*, 2008, pp. 58:1–58:12.

[14] L. Zheng, O. Mengshoel *et al.*, "Belief propagation by message passing in junction trees: Computing each message faster using GPU parallelization," in *UAI'11*, 2011.

[15] L. Zheng and O. Mengshoel, "Optimizing Parallel Belief Propagation in Junction Trees using Regression Belief Propagation in Junction Tree," in *Proceedings of the SIGKDD conference*, 2013, pp. 757–765.

[16] A. Darwiche, "A differential approach to inference in Bayesian networks," *Journal of the ACM*, vol. 50, no. 3, pp. 280–305, 2003.

[17] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *Proceedings of the ICCV Workshops*, 2011, pp. 689–690.

[18] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society.*, pp. 157–224, 1988.

[19] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[20] S. Matsumoto, R. N. Carvalho *et al.*, "UnBBayes: a Java Framework for Probabilistic Models in AI." In Java in Academia and Research., 2011.

[21] T. H. Cormen, C. E. Leiserson *et al.*, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[22] F. E. Sevilgen, S. Aluru *et al.*, "Parallel algorithms for tree accumulations," *JPDC*, vol. 65, no. 1, pp. 85–93, 2005.

[23] G. Elidan, "Bayesian network repository," 1998. [Online]. Available: http://www.cs.huji.ac.il/~galel/Repository/

[24] H. Thai, D. S. Campo *et al.*, "Convergence and coevolution of Hepatitis B virus drug resistance," *Nature communications*, vol. 3, p. 789, 2012.

[25] M. Scutari, "Learning Bayesian networks with the bnlearn R package," *Journal of Statistical Software*, vol. 35, no. i03, 2010.

[26] M. Chavira and A. Darwiche, "Compiling Bayesian networks using variable elimination," in *IJCAI*, 2007, pp. 2443–2449.