Exploring Multiple Dimensions of Parallelism in Junction Tree Message Passing

Lu Zheng
Electrical and Computer Engineering
Carnegie Mellon University

Ole J. Mengshoel
Electrical and Computer Engineering
Carnegie Mellon University

Abstract

Belief propagation over junction trees is known to be computationally challenging in the general case. One way of addressing this computational challenge is to use node-level parallel computing, and parallelize the computation associated with each separator potential table cell. However, this approach is not efficient for junction trees that mainly contain small separators. In this paper, we analyze this problem, and address it by studying a new dimension of node-level parallelism, namely *arithmetic parallelism.* In addition, on the graph level, we use a clique merging technique to further adapt junction trees to parallel computing platforms. We apply our parallel approach to both marginal and most probable explanation (MPE) inference in junction trees. In experiments with a Graphics Processing Unit (GPU), we obtain for marginal inference an average speedup of 5.54x and a maximum speedup of 11.94x; speedups for MPE inference are similar.

1 INTRODUCTION

Bayesian networks (BN) are frequently used to represent and reason about uncertainty. The junction tree is a secondary data structure which can be compiled from a BN [2, 4, 5, 9, 10, 19]. Junction trees can be used for both marginal and most probable explanation (MPE) inference in BNs. Sum-product belief propagation on junction tree is perhaps the most popular exact marginal inference algorithm [8], and max-product belief propagation can be used to compute the most probable explanations [2, 15]. However, belief propagation is computationally hard and the computational difficulty increases dramatically with the density of the BN, the number of states of each network node, and the treewidth of BN, which is upper bounded by the generated junction tree [13]. This computational challenge may hinder the application of BNs in cases where real-time inference is required.

Parallelization of Bayesian network computation is a feasible way of addressing this computational challenge [1, 6, 7, 9, 11, 12, 14, 19, 20]. A data parallel implementation for junction tree inference has been developed for a cache-coherent shared-address-space machine with physically distributed main memory [9]. Parallelism in the basic sum-product computation has been investigated for Graphics Processing Units (GPUs) [19]. The efficiency in using disk memory for exact inference, using parallelism and other techniques, has been improved [7]. An algorithm for parallel BN inference using pointer jumping has been developed [14]. Both parallelization based on graph structure [12] as well as node level primitives for parallel computing based on a table extension idea have been introduced [20]; this idea was later implemented on a GPU [6]. Gonzalez et al. developed a parallel belief propagation algorithm based on parallel graph traversal to accelerate the computation [3].

A parallel message computation algorithm for junction tree belief propagation, based on the cluster-sepset mapping method [4], has been introduced [22]. Cluster-sepset based node level parallelism (denoted *element-wise parallelism* in this paper) can accelerate the junction tree algorithm [22]; unfortunately the performance varies substantially between different junction trees. In particular, for small separators in junction trees, element-wise parallelism [22] provides limited parallel opportunity as explained in this paper.

Our work aims at addressing the small separator issue. Specifically, this paper makes these contributions that further speed up computation and make performance more robust over different BNs from applications:

- We discuss another dimension of parallelism, namely *arithmetic parallelism* (Section 3.1). Integrating arithmetic parallelism with *element-wise parallelism*, we develop an improved parallel sum-product propagation algorithm as discussed in Section 3.2.

- We also develop and test a parallel max-product (Section 3.3) propagation algorithm based on the two dimensions of parallelism.
- On the graph level, we use a clique merging technique (Section 4), which leverages the two dimensions of parallelism, to adapt the various Bayesian networks to the parallel computing platform.

In our GPU experiments, we test the novel two-dimensional parallel approach for both regular sum-propagation and max-propagation. Results show that our algorithms improve the performance of both kinds of belief propagation significantly.

Our paper is organized as follows: In Section 2, we review BNs, junction trees parallel computing using GPUs, and the small-separator problem. In Section 3 and Section 4, we describe our parallel approach to message computation for belief propagation in junction trees. Theoretical analysis of our approach is in Section 5. Experimental results are discussed in Section 6, while Section 7 concludes and outlines future research.

2 BACKGROUND

2.1 Belief Propagation in Junction Trees

A BN is a compact representation of a joint distribution over a set of random variables $\mathcal{X}$. A BN is structured as a directed acyclic graph (DAG) whose vertices are the random variables. The directed edges induce dependence and independence relationships among the random variables. The evidence in a Bayesian network consists of instantiated random variables.

The junction tree algorithm propagates beliefs (or posteriors) over a derived graph called a junction tree. A junction tree is generated from a BN by means of moralization and triangulation [10]. Each vertex C_i of the junction tree contains a subset of the random variables and forms a clique in the moralized and triangulated BN, denoted by $\mathcal{X}_i \subseteq \mathcal{X}$. Each vertex of the junction tree has a potential table $\phi_{\mathcal{X}_i}$. With the above notations, a junction tree can be defined as $J = (\mathbb{T}, \Phi)$, where $\mathbb{T}$ represents a tree and Φ represents all the potential tables associated with this tree. Assuming C_i and C_j are adjacent, a separator S_{ij} is induced on a connecting edge. The variables contained in S_{ij} are defined to be $\mathcal{X}_i \cap \mathcal{X}_j$.

The junction tree size, and hence also junction tree computation, can be lower bounded by *treewidth*, which is defined to be the minimal size of the largest junction tree clique minus one. Considering a junction tree with a treewidth t_w, the amount of computation is lower-bounded by $O(\exp(c * t_w))$ where c is a constant.

Belief propagation is invoked when we get new evidence $\boldsymbol{e}$ for a set of variables $\mathcal{E} \subseteq \mathcal{X}$. We need to update the potential tables Φ to reflect this new information. To do this, belief propagation over the junction tree is used. This is a two-phase procedure: evidence collection and evidence distribution. For the evidence collection phase, messages are collected from the leaf vertices all the way up to a designated root vertex. For the evidence distribution phase, messages are distributed from the root vertex to the leaf vertices.

2.2 Junction Trees and Parallelism

Current emerging many-core platforms, like the recent Graphical Processing Units (GPUs) from NVIDIA and Intel's Knights Ferry, are built around an array of processors running many threads of execution in parallel. These chips employ a Single Instruction Multiple Data (SIMD) architecture. Threads are grouped using a SIMD structure and each group shares a multithreaded instruction unit. The key to good performance on such platforms is finding enough parallel opportunities.

We now consider opportunities for parallel computing in junction trees. Associated with each junction tree vertex $\mathcal{C}_i$ and its variables $\mathcal{X}_i$, there is a potential table $\phi_{\mathcal{X}_i}$ containing non-negative real numbers that are proportional to the joint distribution of $\mathcal{X}_i$. If each variable contains s_j states, the minimal size of the potential table is $|\phi_{\mathcal{X}_i}| = \prod_{j=1}^{|\mathcal{X}_i|} s_j$, where $|\mathcal{X}_i|$ is the cardinality of $\mathcal{X}_i$.

Message passing from $\mathcal{C}_i$ to an adjacent vertex $\mathcal{C}_k$, with separator $\mathcal{S}_{ik}$, involves two steps:

1. **Reduction step**. In sum-propagation, the potential table $\phi_{\mathcal{S}_{ik}}$ of the separator is updated to $\phi^*_{\mathcal{S}_{ik}}$ by reducing the potential table $\phi_{\mathcal{X}_i}$:
$$\phi^*_{\mathcal{S}_{ik}} = \sum_{\mathcal{X}_i/\mathcal{S}_{ik}} \phi_{\mathcal{X}_i}. \tag{1}$$
2. **Scattering step**. The potential table of $\mathcal{C}_k$ is updated using both the old and new table of $\mathcal{S}_{ik}$:
$$\phi^*_{\mathcal{X}_k} = \phi_{\mathcal{X}_k} \frac{\phi^*_{\mathcal{S}_{ik}}}{\phi_{\mathcal{S}_{ik}}}. \tag{2}$$
We define $\frac{0}{0} = 0$ in this case, that is, if the denominator in (2) is zero, then we simply set the corresponding $\phi^*_{\mathcal{X}_k}$ to zeros.

Figure 1: Histograms of the separator potential table sizes of junction trees *Pigs* and *Munin3*. For both junction trees, the great majority of the separator tables contain 20 or fewer elements.

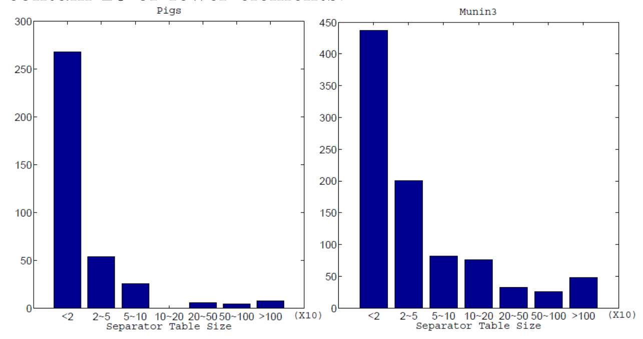
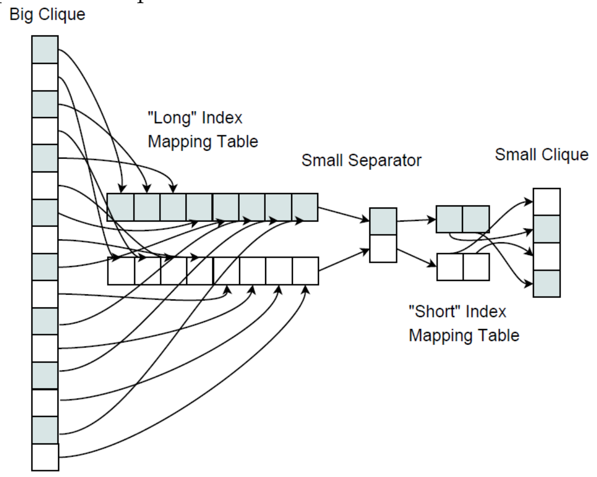


Figure 2: Due to the small separator in the *B*-*S*-*S* pattern, a long index mapping table is produced. If only element-wise parallelism is used, there is just one thread per index mapping table, resulting in slow sequential computation.



Equation (1) and (2) reveal two dimensions of parallelism opportunity. The first dimension, which we return to in Section 3, is arithmetic parallelism. The second dimension is element-wise parallelism [22].

Element-wise parallelism in junction trees is based on the fact that the computation related to each separator potential table cell are independent, and takes advantage of an index mapping table, see Figure 2. In Figure 2, this independence is illustrated by the white and grey coloring of cells in the cliques, the separator, and the index mapping tables. More formally, an *index mapping table* $\mu_{\mathcal{X},\mathcal{S}}$ stores the index mappings from $\phi_{\mathcal{X}}$ to $\phi_{\mathcal{S}}$ [4]. We create $|\phi_{\mathcal{S}_{ik}}|$ such mapping tables. In each mapping table $\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)}$ we store the indices of the elements of $\phi_{\mathcal{X}_i}$ mapping to the j-th separator table element. Mathematically,

$$\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)} = \{r \in [0, |\phi_{\mathcal{X}_i}| - 1] \mid \phi_{\mathcal{X}_i}(r) \text{ is mapped to } \phi_{\mathcal{S}_{ik}}(j)\}.$$

With the index mapping table, element-wise parallelism is obtained by assigning one thread per mapping table of a separator potential table as illustrated in Figure 2 and Figure 3. Consequently, belief propagation over junction trees can often be sped up by using a hybrid CPU/GPU approach [22].

2.3 Small Separator Problem

Figure 1 contains the histograms of two real-world BNs *Pigs* and *Munin3*.[1] We see that most separators in these two BNs are quite small and have a potential table size of less than 20.

In general, small separators can be found in these three scenarios: (i) due to two small neighboring cliques (we call it the *S*-*S*-*S* pattern); (ii) due to a small intersection set of two big neighboring cliques (the *B*-*S*-*B* pattern); and (iii) due to one small neighboring clique and one big neighboring clique (the *B*-*S*-*S* pattern).[2] Due to parallel computing issues, detailed next, these three patterns characterize what we call the *small-separator problem.*

Unfortunately, element-wise parallelism may not provide enough parallel opportunities when a separator is very small and the mapping tables to one or both cliques are very long. A small-scale example, reflecting the *B*-*S*-*S* pattern, is shown in Figure 2.[3] While the mapping tables may be processed in parallel, the long mapping tables result in a significant amount of sequential computation within each mapping table.

A state of the art GPU typically supports more than one thousand concurrent threads, thus message passing through small separators will leave most of the GPU resources idle. This is a major bottleneck for the performance, which we address next.

In this paper, to handle the small separator problem, we use clique merging to eliminate small separators (see Section 4) resulting from the *S*-*S*-*S* pattern and arithmetic parallelism (see Section 3) to attack the *B*-

[1]These BNs can be downloaded at `http://bndg.cs.aau.dk/html/bayesian_networks.html`

[2]These patterns, when read left-to-right, describe the size of a clique, the size of a neigboring separator, and the size of a neigboring clique (different from the first) as found in a junction tree. For example, the pattern *B*-*S*-*S* describes a big clique, a small separator, and a small clique.

[3]In fact, both the "long" and "short" index mapping tables have for presentation purposes been made short–in a realistic junction tree a "long" table can have more than 10,000 entries.

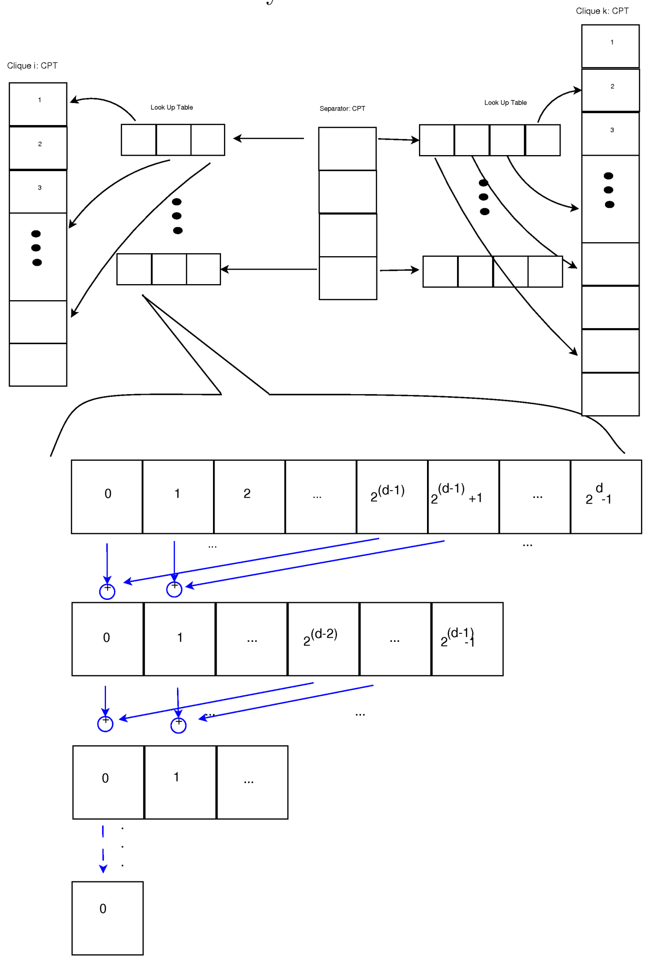


Figure 3: Example of data structure for two GPU cliques and their separator. Arithmetic parallelism (the reverse pyramid at the bottom) is integrated with element-wise parallelism (mapping tables at the top). The arithmetic parallelism achieves parallel summation of 2^d terms in d cycles.

S-*S* and *B*-*S*-*B* patterns.

3 PARALLEL MESSAGE PASSING IN JUNCTION TREES

In order to handle the small-separator problem, we discuss another dimension of parallelism in addition to *element-wise parallelism*, namely *arithmetic parallelism*. Arithmetic parallelim explores the parallel opportunity in the sum of (1) and in the multiplication of (2). By considering also arithmetic parallelism, we can better match the junction tree and the many-core GPU platform by optimizing the computing resources allocated to the two dimensions of parallelism.

Mathematically, this optimization can be modeled as a computation time minimization problem:

$$\begin{array}{c} \min_{p_e, p_a} T(p_e, p_a, \mathcal{C}_i, \mathcal{C}_j, \Phi), \\ \text{subject to} : p_e + p_a \leq p_{tot} \end{array} \tag{3}$$

where $T(\cdot)$ is the time consumed for a message passing from clique $\mathcal{C}_i$ to clique $\mathcal{C}_j$; p_e and p_a are the number of parallel threads allocated to the element-wise and arithmetic dimensions respectively; p_{tot} is the total number of parallel threads available in the GPU; and Φ is a collection of GPU-related parameters, such as the cache size, etc. Equation (3) is a formulation of optimizing algorithm performance on a parallel computing platform. Unfortunately, traditional optimization techniques can typically not be applied to this optimization problem. This is because the analytical form of $T(\cdot)$ is usually not available, due to the complexity of the hardware platform. So in our work we choose p_e and p_a empirically for our implementation.

In the rest of this section, we will describe our algorithm design, seeking to explore both element-wise and arithmetic parallelism.

3.1 Arithmetic Parallelism

Arithmetic parallelism needs to be explored in different ways for reduction and scattering, and also integrated with element-wise parallelism, as we will discuss now.

For reduction, given a certain fixed element j, Equation (1) is essentially a summation over all the clique potential table $\phi_{\mathcal{X}_i}$ elements indicated by the corresponding mapping table $\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)}$. The number of sums is $|\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)}|$. We compute the summation in parallel by using the approach illustrated in Figure 3. This improves the handling of long index mapping tables induced, for example, by the *B*-*S*-*B* and *B*-*S*-*S* patterns. The summation is done in several iterations. In each iteration, the numbers are divided into two groups and the corresponding two numbers in each group are added in parallel. At the end of this recursive process, the sum is obtained, as shown in Algorithm 1. The input parameter d is discussed in Section 3.2; the remaining inputs are an array of floats.

Algorithm 1 ParAdd($d, op(0), \ldots, op(2^d - 1)$)

```
Input: $d, op(0), \ldots, op(2^d - 1)$.
sum = 0
for $i = 1$ to $d$ do
    for $j = 0$ to $2^{d-i} - 1$ in parallel do
        $op(j) = op(j) + op(j + 2^{d-i})$
    end for
end for
return $op(0)$
```

For scattering, Equation (2) updates the elements of $\phi_{\mathcal{X}_k}$ independently despite that $\phi_{\mathcal{S}_{ik}}$ and $\phi^*_{\mathcal{S}_{ik}}$ are re-used to update different elements. Therefore, we can compute each multiplication in (2) with a single thread. The parallel multiplication algorithm is given in Algorithm 2. The input parameter p is discussed in Section 3.2; the remaining inputs are an array of floats.

Algorithm 2 ParMul($p, op1, op2(0), \ldots, op2(p-1)$)

```
Input: $p, op1, op2(0), \ldots, op2(p-1)$.
sum = 0
for $j = 0$ to $p - 1$ in parallel do
    $op2(j) = op1 * op2(j)$
end for
```

3.2 Parallel Belief Propagation

Combining element-wise and arithmetic parallelism, we design the reduction and scattering operations as shown in Algorithm 3 and Algorithm 4. Both of these algorithms take advantage of arithmetic parallelism. In Algorithm 3, the parameter d is the number of cycles used to compute the reduction (summation), while in Algorithm 4, p is the number of threads operating in parallel. Both p and d are parameters that determine the degree of arithmetic parallelism. They can be viewed as a special form of p_a in (3). Based on Algorithm 3 and Algorithm 4, junction tree message passing can be written as shown in Algorithm 5.

Belief propagation can be done using both breadth-first and depth-first traversal over a junction tree. We use the Hugin algorithm [5], which adopts depth-first belief propagation. Given a junction tree J with root vertex C_{root}, we first initialize the junction tree by multiplying together the Bayesian network potential tables (CPTs). Then, two phase belief propagation is adopted [10]: collect evidence and then distribute evidence [22].

Algorithm 3 Reduction($d, \phi_{\mathcal{X}_i}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$)

```
Input: $d, \phi_{\mathcal{X}_i}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$.
for $n = 1$ to $|\phi_{\mathcal{S}_{ik}}|$ in parallel do
    for $j = 0$ to $\lceil |\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(n)}|/2^d \rceil$ do
        sum = sum + ParAdd($d, \phi_{\mathcal{X}_i}(\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(n)}(j * 2^d)), \ldots, \phi_{\mathcal{X}_i}(\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(n)}((j+1) * 2^d - 1))$)
    end for
end for
```

Algorithm 4 Scattering($p, \phi_{\mathcal{X}_k}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_k,\mathcal{S}_{ik}}$)

```
Input: $p, \phi_{\mathcal{X}_k}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_k,\mathcal{S}_{ik}}$.
for $n = 1$ to $|\phi_{\mathcal{S}_{ik}}|$ in parallel do
    for $j = 0$ to $\lceil |\phi_{\mathcal{S}_{ik}}|/p \rceil$ do
        sum = sum + ParMul($p, \frac{\phi^*_{\mathcal{S}_{ik}}(n)}{\phi_{\mathcal{S}_{ik}}(n)}, \phi_{\mathcal{X}_k}(\mu_{\mathcal{X}_k,\mathcal{S}_{ik}(n)}(j * p)), \ldots, \phi_{\mathcal{X}_k}(\mu_{\mathcal{X}_k,\mathcal{S}_{ik}(n)}((j+1) * p - 1))$)
    end for
end for
```

3.3 Max-product Belief Propagation

In this paper, we also apply our parallel techniques to max-product propagation (or in short, max-propagation), which is also referred as the Viterbi algorithm. Max-propagation solves the problem of computing a most probable explanation. For max-propagation, the $\sum$ in (1) is replaced by max [2,15]. In this paper we use sum-product propagation to explain our parallel algorithms; the explanation can generally be changed to discuss max-propagation by replacing add with max.

4 CLIQUE MERGING FOR JUNCTION TREES

The performance of our parallel algorithm is to a large extent determined by the degree of parallelism available in message passing, which intuitively can be measured by the separator size $|\phi_{\mathcal{S}_{ik}}|$, which determines the element-wise parallelism, and the mapping table size $|\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(n)}|$ which upper bounds the arithmetic parallelism. In other words, the larger $|\phi_{\mathcal{S}_{ik}}|$ and $|\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(n)}|$ are, the greater is the parallelism opportunity. Therefore, message passing between small cliques (the *S-S-S* pattern), where $|\phi_{\mathcal{S}_{ik}}|$ and $|\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(n)}|$ are small, is not expected to have good performance. There is not enough parallelism to make full use of

Algorithm 5 PassMessage($p, d, \phi_{\mathcal{X}_i}, \phi_{\mathcal{X}_k}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$)

```
Input: $p, d, \phi_{\mathcal{X}_i}, \phi_{\mathcal{X}_k}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$.
Reduction($d, \phi_{\mathcal{X}_i}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$)
Scattering($p, \phi_{\mathcal{X}_k}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_k,\mathcal{S}_{ik}}$)
```

a GPU's computing resources.

In order to better use the GPU computing power, we propose to remove small separators (that follow the *S*-*S*-*S* pattern) by selectively merging neighboring cliques. This increases the length of mapping tables, however the arithmetic parallelism techniques introduced in Section 3 can handle this. Clique merging can be done offline according to this theorem [8].

Theorem 4.1 *Two neighboring cliques C_i and C_j in a junction tree J with the separator S_{ij} can be merged together into an equivalent new clique C_{ij} with the potential function*

$$\phi(x_{C_{ij}}) = \frac{\phi(x_{C_i})\phi(x_{C_j})}{\phi(x_{S_{ij}})}, \tag{4}$$

while keeping all the other part of the junction tree unchanged.

The result of merging cliques is three-fold: (i) it produces larger clique nodes and thus longer mapping tables; (ii) it eliminates small separators; and (iii) it reduces the number of cliques. Larger clique nodes will result in more computation and therefore longer processing time for each single thread, but getting rid of small separators will improve utilization of the GPU and reduce computation time. We have to manage these two conflicting objectives to improve the overall performance of our parallel junction tree algorithm.

Our algorithm for clique merging is shown in Algorithm 6. It uses two heuristic thresholds, the separator size threshold τ_s and the index mapping table size threshold τ_μ, to control the above-mentioned two effects. We only merge two neighboring cliques C_i and C_j into a new clique C_{ij} when $|\phi_{\mathcal{S}_{ij}}| < \tau_s$ and $|\mu_{\mathcal{X}_j,\phi_S}| < \tau_\mu$.

Algorithm 6 MergeCliques(J, τ_s, τ_μ)

```
merge_flag = 1
while merge_flag do
  merge_flag = 0
  for each adjacent clique pair $(C_i, C_j)$ in $J$ do
    if $\phi_{\mathcal{S}} < \tau_s$ and $|\mu_{\mathcal{X}_j,\phi_S}| < \tau_\mu$ then
      Merge $(J, C_i, C_j)$
      merge_flag = 1
    end if
  end for
end while
```

Given an *S*-*S*-*S*-*S*-*S* pattern, Algorithm 6 may merge two *S* cliques and produce an *S*-*B*-*S* pattern. Here, *B* is the merged clique. Note that the *B*-*S* sub-pattern creates a long index mapping table, which is exactly what arithmetic parallelism handles. There is in other words potential synergy between clique merging and arithmetic parallelism, as is further explored in experiments in Section 6.

5 ANALYSIS AND DISCUSSION

In this section, we analyze the theoretical speedup for our two-dimensional parallel junction tree inference algorithm under the idealized assumption that there is unlimited parallel threads available from the many-core computing platform.

The degree of parallelism opportunity is jointly determined by the size of the separators' potential table, $|\phi_{\mathcal{S}}|$, and the size of the index mapping table $|\mu_{\mathcal{X},\phi_S}|$. Consider a message passed from $\mathcal{C}_i$ to $\mathcal{C}_k$. Since we employ separator table element-wise parallelism in our algorithm, we only need to focus on the computation related to one particular separator table element. With the assumption of unlimited parallel threads, we can choose $d = \lceil \log |\mu_{\mathcal{X}_i,\phi_S}| \rceil$. The time complexity for the reduction is then $\lceil \log |\mu_{\mathcal{X}_i,\phi_S}| \rceil$, due to our use of summation parallelism.[4] Note since $|\mu_{\mathcal{X}_i,\phi_S}| = \frac{|\phi_{\mathcal{X}_i}|}{|\phi_{\mathcal{S}}|}$, the time complexity can be written as $\lceil \log |\phi_{\mathcal{X}_i}| - \log |\phi_{\mathcal{S}}| \rceil$. For the scattering phase, we choose $p = |\mu_{\mathcal{X}_k,\phi_S}|$ and the time complexity is given by $|\mu_{\mathcal{X}_k,\phi_S}|/p + 1 = 2$ due to the multiplication parallelism. Thus the overall time complexity of the two-dimensional belief propagation algorithm is:

$$\lceil \log |\phi_{\mathcal{X}_i}| - \log |\phi_{\mathcal{S}}| \rceil + 2, \tag{5}$$

which is the theoretical optimal time complexity under the assumption of an infinite number of threads. Nevertheless, this value is hard to achieve in practice since the value of d and p are subject to the concurrency limit of the computing platform. For example, in the above-mentioned BN *Pigs*, some message passing requires $p = 1120$ while the GTX460 GPU supports at most 1024 threads per thread block.

Belief propagation is a sequence of messages passed in a certain order [10], for both CPU and GPU [22]. Let $Ne(\mathcal{C})$ denote the neighbors of $\mathcal{C}$ in the junction tree. The time complexity for belief propagation is

$$\sum_i \sum_{k \in Ne(\mathcal{C}_i)} \left(\lceil \log |\phi_{\mathcal{X}_i}| - \log |\phi_{\mathcal{S}}| \rceil + 2 \right),$$

Kernel invocation overhead, incurred each time Algorithm 5 is invoked, turns out to be an important performance factor. If we model the invocation overhead for each kernel call to be a constant τ, then the time

[4]We assume, for simplicity, sum-propagation. The analysis for max-propagation is similar.

complexity becomes

$$\sum_i d_i \tau + \sum_i \sum_{k \in Ne(\mathcal{C}_i)} \left(\lceil \log |\phi_{\mathcal{X}_i}| - \log |\phi_{\mathcal{S}}| \rceil + 2 \right),$$

where d_i is the degree of a node $\mathcal{C}_i$. In a tree structure, $\sum d_i = 2(n-1)$. Thus the GPU time complexity is

$$2(n-1)\tau + \sum_i \sum_{k \in Ne(\mathcal{C}_i)} \left(\lceil \log |\phi_{\mathcal{X}_i}| - \log |\phi_{\mathcal{S}}| \rceil + 2 \right).$$

From this equation, we can see that the junction tree topology impacts GPU performance in at least two ways: the total invocation overhead is proportional to the number of nodes in the junction tree, while the separator and clique table sizes determine the degree of parallelism.

The overall speedup of our parallel belief propagation approach is determined by the equation

$$Speedup = \frac{\sum_i \sum_{k \in Ne(\mathcal{C}_i)} (|\phi_{\mathcal{X}_i}| + |\phi_{\mathcal{X}_k}|)}{2(n-1)\tau + \sum_i \sum_{k \in Ne(\mathcal{C}_i)} \left(\left\lceil \log \frac{|\phi_{\mathcal{X}_i}|}{|\phi_{\mathcal{S}}|} \right\rceil + 2 \right)}.$$

Clearly, the speedup depends on the distribution of the sizes of the separators' and cliques' potential tables. That is the reason we propose the clique merging technique. Using clique merging, we change the number of nodes in the junction tree and distribution of the size of the separators' and cliques' potential table as well, adapting the junction tree for the specific parallel computing platform.

From the equations above, we can estimate the overall belief propagation speedup. However, taking into account that the CPU/GPU platform incurs invocation overhead and the long memory latency when loading data from slow device memory to fast shared memory, the theoretical speedup is hard to achieve in practice. We take an experimental approach to study how the structure of the junction trees affects the performance of our parallel technique on the CPU/GPU setting in Section 6.

6 EXPERIMENTAL RESULTS

In experiments, we study Bayesian networks compiled into junction trees. We not only want to compare the two-dimensional parallel junction tree algorithm to the sequential algorithm, but also study how effective the arithmetic parallelism and clique merging methods are. Consequently, we experiment with different combinations of element-wise parallelism (EP), arithmetic parallelism (AP), and clique merging (CM).

6.1 Computing Platform

We use the NVIDIA GeForce GTX460 as the platform for our implementation. This GPU consists of seven multiprocessors, and each multiprocessor consists of 48 cores and 48K on-chip shared memory per thread block. The peak thread level parallelism achieves 907GFlop/s. In addition to the fast shared memory, a much larger but slower off-chip global memory (785 MB) that is shared by all multiprocessors is provided. The bandwidth between the global and shared memories is about 90 Gbps. In the junction tree computations we are using single precision for the GPU and the thread block size is set to 256.

6.2 Methods and Data

For the purpose of comparison, we use the same set of BNs as used previously [22] (see `http://bndg.cs.aau.dk/html/bayesian_networks.html`). They are from different domains, with varying structures and state spaces. These differences lead to very different junction trees, see Table 1, resulting in varying opportunities for element-wise and arithmetic parallelism. Thus, we use clique merging to carefully control our two dimensions of parallelism to optimize performance. The Bayesian networks are compiled into junction trees and merged offline and then junction tree propagation is performed.

6.3 GPU Optimization: Arithmetic Parallelism

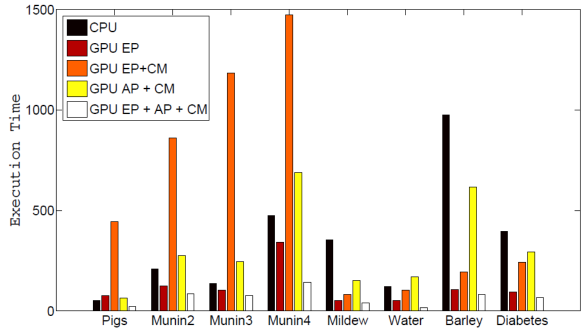
Arithmetic parallelism gives us more freedom to match the parallelism in message passing and the concurrency provided by a GPU: when there is not enough potential table element-wise parallelism available, we can increase the degree of arithmetic parallelism. The number of threads assigned to arithmetic parallelism affects the performance significantly. The parameter p in parallel scattering and the parameter 2^d in the parallel reduction should be chosen carefully (see Algorithm 1 and 2). Since the GPU can provide only limited concurrency, we need to balance the arithmetic parallelism and the element-wise parallelism for each message passing to get the best performance.

Consider message passing between big cliques, for example according to the B-S-B pattern. Intuitively, the values of the arithmetic parallelism parameters p and d should be set higher than for the message passing between smaller cliques. Thus, based on extensive experimentation, we currently employ a simple heuristic parameter selection scheme for the scattering parameter p

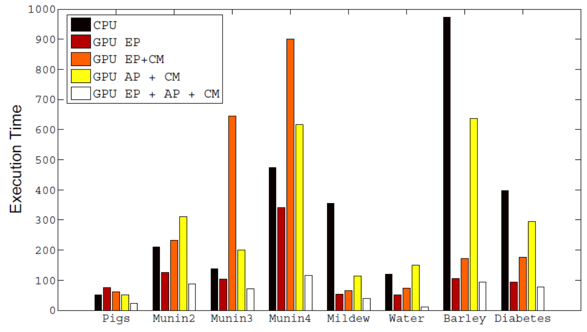
$$p = \begin{cases} 4 & \text{if} |\mu_{\mathcal{X}_i, \mathcal{S}_{ik}(n)}| \leq 100 \\ 128 & \text{if} |\mu_{\mathcal{X}_i, \mathcal{S}_{ik}(n)}| > 100 \end{cases} \quad (6)$$

Dataset	**Pigs**	**Munin2**	**Munin3**	**Munin4**	**Mildew**	**Water**	**Barley**	**Diabetes**
# of original JT nodes	368	860	904	872	28	20	36	337
# of JT nodes after merge	162	553	653	564	22	18	35	334
Avg. CPT size before merge	1,972	5,653	3,443	16,444	341,651	173,297	512,044	32,443
Avg. CPT size after merge	5,393	10,191	7,374	26,720	447,268	192,870	527,902	33,445
Avg. SPT size before merge	339	713	533	2,099	9,273	26,065	39,318	1,845
Avg. SPT size after merge	757	1,104	865	3,214	11,883	29,129	40,475	1,860
GPU time (sum-prop) [ms]	22.61	86.40	74.99	141.08	41.31	16.33	81.82	68.26
GPU time (max-prop) [ms]	22.8	86.8	72.6	114.9	38.6	12.1	94.3	78.3
CPU time (sum-prop) [ms]	51	210	137	473	355	120	974	397
CPU time (max-prop) [ms]	59	258	119	505	259	133	894	415
Speedup (sum-prop)	**2.26x**	**2.43x**	**1.82x**	**3.35x**	**8.59x**	**7.35x**	**11.94x**	**5.81x**
Speedup (max-prop)	**2.58x**	**2.97x**	**1.64x**	**4.39x**	**6.71x**	**10.99x**	**9.48x**	**5.30x**

Table 1: Junction tree (JT) statistics and belief propagation (BP) performance for various junction trees, with speedup for our GPU approach (GPU EP + AP + CM) compared to CPU-only in the two bottom rows. The row "CPU time (sum-prop)" gives previous results [22].



(a) Junction tree sum-propagation



(b) Junction tree max-propagation

Figure 4: Comparison of combinations of junction tree optimization techniques CM, AP, and EP for (a) sum- and (b) max-propagation. Best performance is achieved for GPU EP + AP + CM.

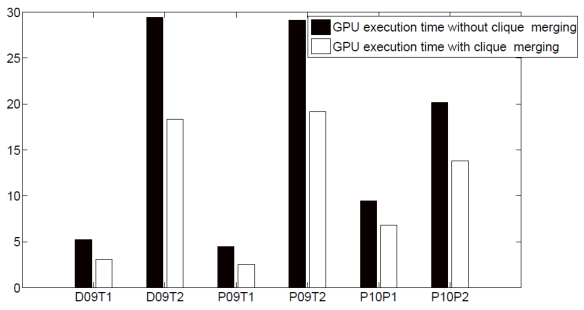


Figure 5: GPU execution times with CM (GPU EP + AP + CM) and without CM (GPU EP + AP) for junction trees compiled from sparse BNs representing electrical power systems.

and the reduction parameter d

$$d = \begin{cases} 2 & \text{if} |\mu_{\mathcal{X}_i, \mathcal{S}_{ik}(n)}| \leq 100 \\ 7 & \text{if} |\mu_{\mathcal{X}_i, \mathcal{S}_{ik}(n)}| > 100 \end{cases} \quad (7)$$

We compare the execution time when using element-wise parallelism alone and the case when element-wise parallelism is used in combination with arithmetic parallelism. Results, for both sum-propagation and max-propagation, are shown in Figure 4(a) and Figure 4(b). In all cases, the GPU EP + AP + CM outperforms all the other approaches.[5]

6.4 GPU Optimization: Clique Merging

Clique merging is based on the observation that many junction trees mostly consist of small cliques. This lack of parallelism opportunity will hinder the efficient

[5]The GPU EP + CM combination is included for completeness, but as expected it often performs very poorly. The reason for this is that CM, by merging cliques, creates larger mapping tables that EP is not equipped to handle.

use of the available computing resources, since a single message passing will not be able to occupy the GPU. Merging neighboring small cliques, found in the *S*-*S*-*S* pattern, can help to increase the average size of separators and cliques. Clique merging also reduces the total number of nodes in the junction tree, which in turn reduces the invocation overhead.

We use two parameters to determine which cliques should be merged–one is τ_s, the threshold for separators' potential table size and the other is τ_μ, the threshold for the size of the index mapping table. These parameters are set manually in this paper, however in a companion paper [21] this parameter optimization process is automated by means of machine learning.

In the experiments, we used both arithmetic parallelism and element-wise parallelism. This experiment presents how much extra speedup can be obtained by using clique merging and arithmetic parallelism. The experimental results can be found in Table 1, in the rows showing the GPU execution times for both sum-propagation and max-propagation. In junction trees *Pigs*, *Munin2*, *Munin3* and *Munin4*, a considerable fraction of cliques (and consequently, separators) are small, in other words the *S*-*S*-*S* pattern is common. By merging cliques, we can significantly increase the average separators' and cliques' potential size and thus provide more parallelism.

Comparing GPU EP + AP with GPU EP + AP + CM, the speedup for these junction trees ranged from 10% to 36% when clique merging is used. However, in junction trees *Mildew*, *Water*, *Barley* and *Diabetes*, clique merging does not help much since they mainly consist of large cliques to start with.

Another set of experiments were performed with junction trees that represent an electrical power system ADAPT [16–18]. These junction trees contain many small cliques, due to their underlying BNs being relatively sparsely connected.[6] The experimental results are shown in Figure 5. Using clique merging, the GPU execution times are shortened by 30%-50% for these BNs compared to not using clique merging.

6.5 Performance Comparison: CPU

As a baseline, we implemented a sequential program on an Intel Core 2 Quad CPU with 8MB cache and a 2.5GHz clock. The execution time of the program is comparable to that of GeNie/SMILE, a widely used C++ software package for BN inference.[7] We do not directly use GeNie/SMILE as the baseline here, because we do not know the implementation details of GeNie/SMILE.

In Table 1, the bottom six rows give the execution time comparison for our CPU/GPU hybrid versus a traditional CPU implementation. The CPU/GPU hybrid uses arithmetic parallelism, element-wise parallelism and clique merging. The obtained speedup for sum-propagation ranges from 1.82x to 11.94x, with an arithmetic average of 5.44x and a geometric average of 4.42.

The speedup for max-propagation is similar to, but different from sum-propagation in non-trivial ways. The performance is an overall effect of many factors such as parallelism, memory latency, kernel invocation overhead, etc. Those factors, in turn, are closely correlated with the underlying structures of the junction trees. The speedup for max-propagation ranges from 1.64x to 10.99x, with an arithmetic average of 5.51x and a geometric average of 4.61x.

6.6 Performance Comparison: Previous GPU

We now compare the GPU EP + AP + CM technique introduced in this paper with our previous GPU EP approach [22]. From results in Table 1, compared with the GPU EP approach [22], the arithmetic average cross platform speedup increases from 3.38x (or 338%) to 5.44x (or 544%) for sum-propagation. For max-propagation[8] the speedup increases from 3.22x (or 322%) to 5.51x (or 551%).

7 CONCLUSION AND FUTURE WORK

In this paper, we identified small separators as bottlenecks for parallel computing in junction trees and developed a novel two-dimensional parallel approach for belief propagation over junction trees. We enhanced these two dimensions of parallelism by careful clique merging in order to make better use of the parallel computing resources of a given platform.

In experiments with a CUDA implementation on an NVIDIA GeForce GTX460 GPU, we explored how the performance of our approach varies with different junction trees from applications and how clique merging can improve the performance for junction trees that contains many small cliques. For sum-propagation, the average speedup is 5.44x and the maximum speedup is 11.94x. The average speedup for max-propagation is 5.51x while the maximum speedup is 10.99x.

In the future, we would like to see research on parameter optimization for both clique merging and message

[6] `http://works.bepress.com/ole_mengshoel/`

[7] `http://genie.sis.pitt.edu/`

[8] We implemented max-propagation based on the approach developed previously [22].

passing. It would be useful to automatically change the merging parameters for different junction trees based on the size distribution of the cliques and separators. In addition, we also want to automatically change the kernel running parameters for each single message passing according to the size of a message. In fact, we have already made progress along these lines, taking a machine learning approach [21].

Acknowledgments

This material is based, in part, upon work supported by NSF awards CCF0937044 and ECCS0931978.

References

[1] R. Bekkerman, M. Bilenko, and J. Langford, editors. *Scaling up Machine Learning: Parallel and Distributed Approaches.* Cambridge University Press, 2011.

[2] A. P. Dawid. Applications of a general propagation algorithm for probabilistic expert systems. *Statistics and Computing*, 2:25–36, 1992.

[3] J. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *Proc. of the 25th Conference in Uncertainty in Artificial Intelligence (UAI-09)*, 2009.

[4] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, (3):225–263, 1994.

[5] F. V. Jensen, S. L. Lauritzen, and K. G. Olesen. Bayesian updating in causal probabilistic network by local computations. *Computational Statistics*, pages 269–282, 1990.

[6] H. Jeon, Y. Xia, and V. K. Prasanna. Parallel exact inference on a CPU-GPGPU heterogenous system. In *Proc. of the 39th International Conference on Parallel Processing*, pages 61–70, 2010.

[7] K. Kask, R. Dechter, and A. Gelfand. BEEM: bucket elimination with external memory. In *Proc. of the 26th Annual Conference on Uncertainty in Artificial Intelligence (UAI-10)*, pages 268–276, 2010.

[8] D. Koller and N. Friedman, editors. *Probablistic graphic model: principle and techniques.* The MIT Press, 2009.

[9] A. V. Kozlov and J. P. Singh. A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In *Proc. of the 1994 ACM/IEEE conference on Supercomputing*, pages 320–329, 1994.

[10] 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*, 50(2):157–224, 1988.

[11] M. D. Linderman, R. Bruggner, V. Athalye, T. H. Meng, N. B. Asadi, and G. P. Nolan. High-throughput Bayesian network learning using heterogeneous multicore computers. In *Proc. of the 24th ACM International Conference on Supercomputing*, pages 95–104, 2010.

[12] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Proc. of the 26th Annual Conference on Uncertainty in Artificial Intelligence (UAI-10)*, pages 340–349, 2010.

[13] O. J. Mengshoel. Understanding the scalability of Bayesian network inference using clique tree growth curves. *Artificial Intelligence*, 174:984–1006, 2010.

[14] V. K. Namasivayam and V. K. Prasanna. Scalable parallel implementation of exact inference in Bayesian networks. In *Proc. of the 12th International Conference on Parallel and Distributed System*, pages 143–150, 2006.

[15] D. Nilsson. An efficient algorithm for finding the m most probable configurations in probablistic expert system. *Statistics and Computing*, pages 159–173, 1998.

[16] B. W. Ricks and O. J. Mengshoel. The diagnostic challenge competition: Probabilistic techniques for fault diagnosis in electrical power systems. In *Proc. of the 20th International Workshop on Principles of Diagnosis (DX-09)*, pages 415–422, Stockholm, Sweden, 2009.

[17] B. W. Ricks and O. J. Mengshoel. Methods for probabilistic fault diagnosis: An electrical power system case study. In *Proc. of Annual Conference of the PHM Society, 2009 (PHM-09)*, San Diego, CA, 2009.

[18] B. W. Ricks and O. J. Mengshoel. Diagnosing intermittent and persistent faults using static bayesian networks. In *Proc. of the 21st International Workshop on Principles of Diagnosis (DX-10)*, Portland, OR, 2010.

[19] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *Proc. of the 22nd ACM International Conference on Supercomputing*, pages 309–318, 2008.

[20] Y. Xia and V. K. Prasanna. Node level primitives for parallel exact inference. In *Proc. of the 19th International Symposium on Computer Architechture and High Performance Computing*, pages 221–228, 2007.

[21] L. Zheng and O. J. Mengshoel. Optimizing parallel belief propagation in junction trees using regression. In *Proc. of 19th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD-13)*, Chicago, IL, August 2013.

[22] L. Zheng, O. J. Mengshoel, and J. Chong. Belief propagation by message passing in junction trees: Computing each message faster using GPU parallelization. In *Proc. of the 27th Conference in Uncertainty in Artificial Intelligence (UAI-11)*, pages 822–830, Barcelona, Spain, 2011.