

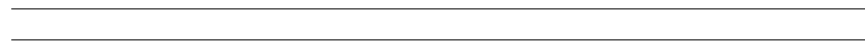
Parallel Exact Inference on the Cell Broadband Engine Processor^{☆,☆☆}

Yinglong Xia

*Computer Science Department, University of Southern California
Los Angeles, CA 90089, U.S.A.*

Viktor K. Prasanna*

*Ming Hsieh Department of Electrical Engineering, University of Southern California
Los Angeles, CA 90089, U.S.A.*



[☆]This research was partially supported by the National Science Foundation under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged.

^{☆☆}A preliminary version of this paper appears in the Proceedings of the 21th International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08).

*Corresponding author

Email addresses: yinglonx@usc.edu (Yinglong Xia), prasanna@usc.edu (Viktor K. Prasanna)

URL: <http://ceng.usc.edu/~prasanna/> (Viktor K. Prasanna)

Abstract

We present the design and implementation of a parallel exact inference algorithm on the Cell Broadband Engine (Cell BE) processor, a heterogeneous multicore architecture. Exact inference is a key problem in exploring probabilistic graphical models, where the computation complexity increases dramatically with the network structure and clique size. In this paper, we exploit parallelism in exact inference at multiple levels. We propose a rerooting method to minimize the critical path for exact inference, and an efficient scheduler to dynamically allocate SPEs. In addition, we explore potential table representation and layout to optimize DMA transfer between local store and main memory. We implemented the proposed method and conducted experiments on the Cell BE processor in the IBM QS20 Blade. We achieved speedup up to $10\times$ on the Cell, compared to state-of-the-art processors. The methodology proposed in this paper can be used for online scheduling of directed acyclic graph (DAG) structured computations.

Keywords

Exact inference, Junction tree, Graphical model, Heterogeneous multicore architecture, Cell Broadband Engine (Cell BE)

1 Introduction

A full joint probability distribution for any real-world system can be used for inference. However, such a distribution increases intractably with the number of variables used to model the system. It is known that independence and conditional independence relationships can greatly reduce the sizes of the joint probability distributions. This property is utilized by *Bayesian networks* [1]. Bayesian networks have been used in artificial intelligence since the 1960s. They have found applications in a number of domains, including medical diagnosis, consumer help desks, pattern recognition, credit assessment, data mining and genetics. [2, 3, 4].

Inference in a Bayesian network is the computation of the conditional probability of the *query* variables, given a set of *evidence* variables as the knowledge to the network. Inference in a Bayesian network can be *exact* or *approximate*. Exact inference is NP hard [5]. The most popular exact inference algorithm for multiple connected networks was proposed by Lauritzen and Spiegelhalter [1], which converts a Bayesian network into a *junction tree*, then performs exact inference in the junction tree. The complexity of the exact inference algorithm increases dramatically with the density of the network, the clique width and the number of states of the random variables. In many cases exact inference must be performed in real time. Therefore, in order to accelerate the exact inference, parallel techniques must be developed.

Several parallel algorithms for exact inference have been presented, such as Pennock [5], Kozlov and Singh [6] and Szolovits [7]. In [8, 9], the authors proposed parallel exact inference algorithms using message passing. We discuss the related work in details in Section 3. However, these algorithms assume homogeneous machine models and therefore do not exploit the specific features of heterogeneous multicore processors such as the Cell BE processor.

The Cell Broadband Engine (Cell BE) processor, jointly developed by IBM, Sony and Toshiba, is a heterogeneous chip with one PowerPC control element (PPE) coupled with eight independent synergistic processing elements (SPEs). The Cell BE processor has been used in the Sony PlayStation 3 (PS3) gaming console, Mercury Computer Systems' dual Cell based blade servers, and IBM's QS20 Cell Blades. The Cell BE processor supports concurrency of computation at the SPE level and vector parallelism with variable granularity. However, to maximize the potential of such multicore processors, algorithms must be parallelized or even redesigned. A developer must understand both the algorithmic and architectural aspects to propose efficient algorithms for heterogeneous processors. Some recent research provides insight to parallel algorithms design for the Cell [10, 11]. However, to the best of our knowledge, no exact inference algorithm has been proposed for the Cell or other heterogeneous multicore processors.

In this paper, we explore parallel exact inference on the Cell at multiple levels. We introduce a rerooting method to minimize the *critical path*. We present an efficient *scheduler* to maintain and allocate the task dependency

graph constructed from an arbitrary junction tree. The scheduler dynamically partitions large tasks and exploits parallelism at various levels of granularity. We also explore *potential table representation* and data layout for data transfer. We implement efficient *node level primitives* and *computation kernels*. The proposed scheduler can be ported to other heterogeneous parallel computing systems for the online scheduling of directed acyclic graph (DAG) structured computations.

The paper is organized as follows: Section 2 discusses the background of Bayesian networks and junction trees. Section 3 discusses related work on parallel exact inference. Section 4 presents the exact inference algorithm for the Cell BE processor. Experimental results are shown in Section 5. Section 6 concludes the paper.

2 Background

2.1 Exact inference in Bayesian Networks and Junction Trees

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to represent a joint distribution compactly. Figure 1 (a) shows a sample Bayesian network. In Figure 1 (a), each node represents a random variable. The edges indicate the probabilistic dependence relationship between two random variables. Notice that these edges can not form any directed cycles. Each random variable in the Bayesian network has a discrete (conditional) probability distribution. We use the following notations to formulate a Bayesian network and its properties. A Bayesian network is defined as $B = (\mathbb{G}, \mathbb{P})$, where \mathbb{G} is a *directed acyclic graph* (DAG) and \mathbb{P} denotes the parameter of the network. The graph \mathbb{G} is denoted $\mathbb{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{A_1, A_2, \dots, A_n\}$ is the node set and \mathcal{E} is the edge set. Each node A_i represents a random variable. If there exists an edge from A_i to A_j i.e. $(A_i, A_j) \in \mathcal{E}$, then A_i is called a *parent* of A_j . $pa(A_j)$ denotes the set of all parents of A_j . Given the value of $pa(A_j)$, A_j is conditionally independent of all other preceding variables. The parameter \mathbb{P} represents a group of *conditional probability tables*, which are defined as the conditional probability $P(A_j|pa(A_j))$ for each random variable A_j . Given the Bayesian network, a joint distribution is given by [1]: $P(\mathcal{V}) = \prod_{j=1}^n Pr(A_j|pa(A_j))$ where $A_j \in \mathcal{V}$.

The *evidence* in a Bayesian network is the variables that have been instantiated with values, e.g. $E = \{A_{e_1} = a_{e_1}, \dots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \dots, n\}$. The evidence variables are the observable nodes in a Bayesian network. In an observation, we obtain the real values of these random variables. The observed values are the evidence - also known as *belief* - to the Bayesian network. We use the observed values to update the prior (conditional) distribution. This is called *evidence absorption*. As the evidence variables depend probabilistically upon other random variables, the evidence can be absorbed and propagated according to Bayes' theorem. Propagating the evidence throughout the Bayesian network changes the distribution of any other variables accordingly. The random variables of which

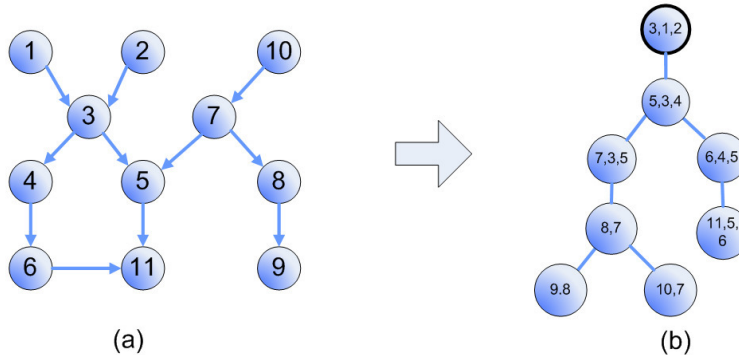


Figure 1: (a) A sample Bayesian network and (b) corresponding junction tree.

the users are concerned about the updated distribution are called *query* variables. The process of *inference* involves propagating the evidence and computing the updated distribution of the query variables. There are two categories of inference algorithms, called *exact inference* and *approximate inference*. Exact inference is proven to be NP hard [5]. The computational complexity of exact inference increases dramatically with the density of the network and the number of states of the random variables.

Traditional exact inference using Bayes' theorem fails for networks with undirected cycles [1]. Most inference methods for networks with undirected cycles convert a network to a cycle-free hypergraph called a *junction tree*. In Figure 1 (b), we illustrate a junction tree converted from the Bayesian network in Figure 1 (a). All undirected cycles in Figure 1 (a) are eliminated in Figure 1 (b). Each vertex in Figure 1 (b) contains multiple random variables. The numbers in each vertex indicate of which random variables in the Bayesian network comprise the vertex. Notice that adjacent vertices always share one or more random variables. All junction trees satisfy the *running intersection property* (RIP) [1]. The RIP requires that the shared random variables of any two vertices in a junction tree should appear in all vertices on the path between the two vertices. The RIP ensures the evidence observed at any random variable can be propagated from one vertex to another. For the sake of exploring evidence propagation in a junction tree, we use the following notations to formulate a junction tree. A junction tree is defined as $J = (\mathbb{T}, \hat{\mathbb{P}})$, where \mathbb{T} represents a tree and $\hat{\mathbb{P}}$ denotes the parameter of the tree. Each vertex \mathcal{C}_i , known as a clique of J , is a set of random variables. Assuming \mathcal{C}_i and \mathcal{C}_j are adjacent, the *separator* between them is defined as $\mathcal{C}_i \cap \mathcal{C}_j$. $\hat{\mathbb{P}}$ is a group of *potential tables*. The potential table of \mathcal{C}_i , denoted $\psi_{\mathcal{C}_i}$, can be viewed as the joint distribution of the random variables in \mathcal{C}_i . For a clique with w variables, each taking r different values, the number of entries in the potential table is r^w .

In a junction tree, exact inference proceeds as follows: Assuming evidence is $E = \{A_i = a\}$ and $A_i \in \mathcal{C}_y$, E is *absorbed* at \mathcal{C}_y by instantiating the variable A_i and renormalizing the remaining constituents of the clique. The evidence is then propagated from \mathcal{C}_y to all other cliques. Mathematically, the evidence propagation is represented

as [1]:

$$\psi_{\mathcal{S}}^* = \sum_{\mathcal{Y} \setminus \mathcal{S}} \psi_{\mathcal{Y}}^*, \quad \psi_{\mathcal{X}}^* = \psi_{\mathcal{X}} \frac{\psi_{\mathcal{S}}^*}{\psi_{\mathcal{S}}} \quad (1)$$

where \mathcal{S} is a separator between cliques \mathcal{X} and \mathcal{Y} ; ψ^* denotes the updated potential table. After all cliques are updated, the distribution of a query variable $Q \in \mathcal{C}_y$ is obtained by summing up all entries with respect to $Q = q$ for all possible q in $\psi_{\mathcal{C}_y}$.

2.2 Cell Broadband Engine Processor

The Cell Broadband Engine (Cell BE) is a novel heterogeneous multicore architecture designed by Sony, Toshiba and IBM, primarily targeting high performance multimedia and gaming applications. The Cell BE processor consists of a traditional PowerPC control element (PPE), which controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. Figure 2 sketches the architecture of the Cell. The PPE is a 64-bit core with a vector multimedia extension (VMX) unit. The PPE has 32 KB L1 instruction and data caches, and a 512 KB L2 cache. Each SPE is a micro-architecture designed for high performance data streaming and data intensive computation. The SPE is an in-order, dual-issue, statically scheduled architecture, in which two SIMD instructions can be issued per cycle. The SPE includes a 256 KB local store (LS) memory to hold both instructions and data. The SPE cannot access main memory (MM) directly, but it can issue DMA commands to the MFC to bring data into the local store or write computation results back to the main memory. DMA is non-blocking so that the SPE can continue execution when the DMA transactions are performed. The MFC can support aligned transfers of multiples of 16 bytes to a maximum of 16 KB. Using the DMA list command, we can issue up to 2048 DMA transfers. If both the effective address and the local store address are 128 bytes aligned, the DMA transfer can achieve its peak performance. Since the clock speed is 3.2 GHz for the Cell, the theoretical peak performance is 204.8 GFlops [12, 13, 14].

2.3 Model of Computation for the Cell

In this paper, we adapt the complexity model proposed by Bader [13] to analyze the performance of algorithms on the Cell. The model captures several architectural features of the Cell that can be exploited for performance. The model is represented by a triplet $\langle T_C, T_D, T_B \rangle$, where T_C denotes the computational complexity, T_D the number of DMA requests, and T_B the number of branching instructions. T_C consists of the SPE computation time $T_C^{(SPE)}$ and the PPE computation time $T_C^{(PPE)}$, where $T_C^{(SPE)}$ is the maximum of computation time of all the SPEs. Notice that $T_C^{(SPE)}$ and $T_C^{(PPE)}$ can be overlapped. Thus, the sum of $T_C^{(SPE)}$ and $T_C^{(PPE)}$ gives the upper bound of T_C .

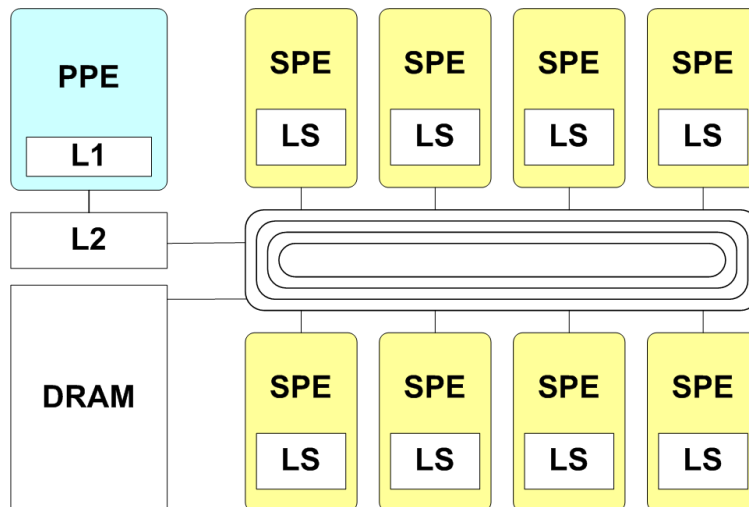


Figure 2: Architecture of the Cell BE processor.

T_D is the latency due to DMA transfers. When there is a large number of DMA requests, T_D may dominate the actual execution time. T_B is employed in the triplet because of the significant overhead due to branch misprediction in SPEs. However, since T_B can be decreased by unrolling loops and inserting branch hints, it is difficult to report accurate number of actual branches [13]. When the misprediction probability is low, T_B has little influence on execution time. Therefore, in this paper, we focus on the two parameters T_C and T_D . The total execution time is decided by data transferring between memory and local stores, and the computation in PPE and SPEs. Notice that, in Cell, the computation and data transfer can be overlapped using double buffering or multiple buffering.

Similar to the complexity model proposed in [13], the adapted model ignores several features of the Cell for the sake of simplifying the analysis. For example, we ignore the effect of floating point precision on the performance of numerical algorithms. We do not consider the SIMD features, since the SIMD results in only a constant factor of improvement. Because of the large bandwidth among PPE and SPEs, synchronization mechanisms such as mailboxes and signals are not considered either.

3 Related Work on Parallel Exact Inference

There are several early works on parallel exact inference, such as Pennock [5], Kozlov and Singh [6] and Szolovits [7]. However, some of those methods, such as [6], are dependent upon the structure of the input Bayesian network. The performance of such methods can be adversely affected when the input Bayesian network is changed. Others, such as [5], exhibit limited performance for multiple evidence inputs, since the evidence is assumed to be at the root of the junction tree. In [9], the authors discussed the structure conversion of Bayesian networks, which is a different problem. In this paper, we start with a junction tree. In [8], the node level primitives are parallelized

using message passing on clusters. Reference [15] discusses junction tree decomposition, a different problem related to exact inference on clusters. In [16], the authors presented techniques for exact inference on the Cell, which form the base of this paper. Unlike [16], we present an efficient rerooting algorithm for exact inference, and analyze the algorithm on Cell using a model of computation. We conduct experiments on the Cell and use a dataset from real application to evaluate the proposed algorithms. Regarding the related work on identifying the critical path in a graph, Shen proposed an algorithm taking $O(\log^2 N)$ time [17], where N is the number of cliques. Shen’s algorithm helps identify the workload in the critical path for junction tree rerooting. However, it requires $O(N)$ processors, while the Cell provides limited number of SPEs. In addition, the above algorithms assume a homogeneous machine model, which is different from the Cell. Several recent studies on the Cell provide insight into parallel computing on heterogeneous multicore processors. For example, Bader studied FFT and list ranking on the Cell [10, 13]. Buehrer discussed scientific computing using the Cell [18]. Petrini *et al.* developed a parallel breadth-first search (BFS) algorithm [14]. Vinjamuri *et al.* implemented Floyd’s algorithm [19]. To the best of our knowledge, no study on exact inference for heterogeneous multicore processors has been reported. In this paper, we present an efficient design and implementation of a parallel exact inference algorithm on the Cell, including designing an efficient scheduler, optimizing data layout and enhancing the performance of node level primitives.

4 Exact Inference for the Cell BE processor

4.1 Sequential Exact Inference

For the sake of completeness, we present a serial exact inference algorithm in Algorithm 1, where we use the notations defined in Section 2. In Algorithm 1, the input consists of a junction tree converted from an arbitrary Bayesian network, the evidence and query variables. All cliques in the junction tree are numbered according to the breadth first search (BFS) order. The output is the a posteriori distribution of the query variables.

In Algorithm 1, Line 1 is *evidence absorption*, where the evidence E is absorbed by cliques. Lines 2-7 perform *evidence collection*, which propagates the evidence from leaf cliques to the root (bottom up). Lines 8-11 in Algorithm 1 perform *evidence distribution*, which propagates the evidence from the root to leaf cliques (top down). Evidence collection and evidence distribution are two major phases in exact inference, which update cliques in reverse BFS order and the original BFS order, respectively. Updating each clique involves a series of computations on potential tables (see Lines 4, 5, 9 and 10 in Algorithm 1). Line 12 in Algorithm 1 computes the distribution of the query variables. The parallelism in Lines 1 and 12 is trivial. Therefore, we focus on the parallelism in evidence collection and distribution. Figure 3 illustrates the first three steps of evidence collection and distribution in a sample junction

tree.

Algorithm 1 Sequential Exact Inference

Input: Junction tree $J = (\mathbb{T}, \hat{\mathbb{P}})$, evidence E and query variables Q , BFS order of cliques α .

Output: Probability distribution of Q

- 1: Every clique absorbs evidence: $\psi_{\mathcal{C}_i} = \psi_{\mathcal{C}_i} \delta(E = e), \forall \mathcal{C}_i$
 - {Evidence collection}
 - 2: **for** $i = N - 1$ to 1 by -1 **do**
 - 3: **for** each $\mathcal{C}_j \in \{\mathcal{C}_j | pa(\mathcal{C}_j) = \mathcal{C}_{\alpha_i}\}$ **do**
 - 4: Compute separator potential table $\psi_{\mathcal{S}}^* = \sum_{\mathcal{C}_j \setminus \mathcal{C}_{\alpha_i}} \psi_{\mathcal{C}_j}$ where $\mathcal{S} = \mathcal{C}_{\alpha_i} \cap \mathcal{C}_j$
 - 5: Update clique \mathcal{C}_{α_i} using $\psi_{\mathcal{C}_{\alpha_i}} = \psi_{\mathcal{C}_{\alpha_i}} \psi_{\mathcal{S}}^* / \psi_{\mathcal{S}}$ where $\psi_{\mathcal{S}} = \sum_{\mathcal{C}_{\alpha_i} \setminus \mathcal{C}_{\alpha_i}} \psi_{\mathcal{C}_{\alpha_i}}$
 - 6: **end for**
 - 7: **end for**
 - {Evidence distribution}
 - 8: **for** $i = 2$ to N **do**
 - 9: Compute separator potential table $\psi_{\mathcal{S}}^* = \sum_{\mathcal{C}_{\alpha_i} \setminus \mathcal{C}_{pa(\alpha_i)}} \psi_{\mathcal{C}_{\alpha_i}}$ where $\mathcal{S} = \mathcal{C}_{\alpha_i} \cap pa(\mathcal{C}_{\alpha_i})$
 - 10: Update $\psi_{\mathcal{C}_{\alpha_i}}$ using $\psi_{\mathcal{C}_{\alpha_i}} = \psi_{\mathcal{C}_{\alpha_i}} \psi_{\mathcal{S}}^* / \psi_{\mathcal{S}}$ where $\psi_{\mathcal{S}} = \sum_{\mathcal{C}_{\alpha_i} \setminus \mathcal{C}_{\alpha_i}} \psi_{\mathcal{C}_{\alpha_i}}$
 - 11: **end for**
 - 12: Compute query Q from $\psi_{\mathcal{C}_i}$ if $Q \cap \mathcal{C}_i \neq \emptyset$ using $p(Q) = \frac{1}{Z} \sum_{\mathcal{C}_i \setminus Q} \psi_{\mathcal{C}_i}$
-

We analyze the computation time for evidence collection and distribution (Lines 2-11) in Algorithm 1. Lines 2 and 3 introduce two loops with $O(kN)$ iterations, where k is the maximum number of children of a clique and N is the number cliques in the junction tree. Lines 4 and 5 involve *node level primitives* on potential tables introduced in [8]. Line 4 applies a primitive called *table marginalization*. Marginalization converts the index of each entry in $\psi_{\mathcal{C}_j}$ to a *state string*, and maps the state string to an index of an entry in $\psi_{\mathcal{S}}^*$. The data in each entry of $\psi_{\mathcal{C}_j}$ is added to the corresponding entry in $\psi_{\mathcal{S}}^*$. Reference [8] presents detailed analysis for the complexity of the primitives. The marginalization takes $(|\psi_{\mathcal{C}_j}|w)$ time according to [8], where $|\psi_{\mathcal{C}_j}| = \prod_{i=1}^{w_{\mathcal{C}_j}} r_i$ is the size of $\psi_{\mathcal{C}_j}$ and $w, w_{\mathcal{S}}$ are the maximum clique widths for cliques and separators respectively. Line 5 involves *table multiplication, table division* and another table marginalization. According to [8], the total computation complexity is also $O(|\psi_{\mathcal{C}_j}|w)$. Lines 8-11 perform evidence distribution using a loop with $(N - 1)$ iterations. Lines 9 and 10 correspond to Lines 4 and 5. Thus, the computation complexity is $O(|\psi_{\mathcal{C}_j}|w)$ for both Lines 9 and 10. Based on the above analysis, the total computation complexity for Algorithm 1 is given by:

$$O\left(\sum_{i=1}^N k_i \prod_{j=1}^{w_{\mathcal{C}_i}} r_j \cdot w_{\mathcal{C}_i}\right) = O(Nkwr^w) \quad (2)$$

where k, w and r are the maximum number of children of a clique, the maximum clique width and the maximum number of states for random variables respectively. Eq. (2) implies that, when r and w are moderately large, evidence propagation is dominated by r^w .

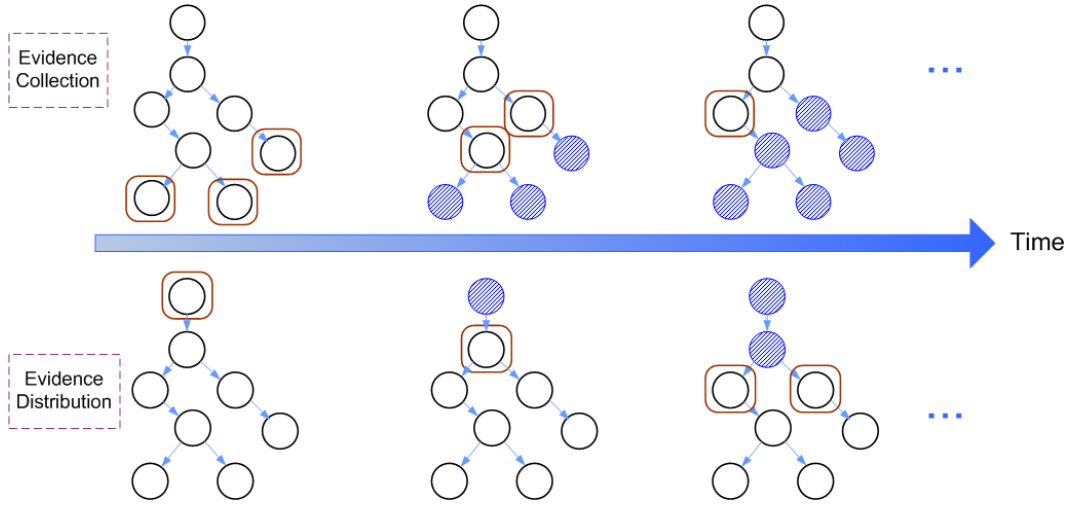


Figure 3: Illustration of evidence collection and evidence distribution in a junction tree. The cliques in boxes are under processing. The shaded cliques have been processed.

4.2 Junction Tree Rerooting

4.2.1 Minimizing Critical Path by Rerooting

A junction tree can be rerooted at any clique [5]. Consider rerooting a junction tree at clique \mathcal{C} . Let α be a preorder walk of the underlying undirected tree, starting from \mathcal{C} . Then, α encodes the desired new edge directions, i.e. an edge in the rerooted tree points from \mathcal{C}_{α_i} to \mathcal{C}_{α_j} if and only if $\alpha_i < \alpha_j$. In the rerooting procedure, we check the edges in the given junction tree, and reverse any edges inconsistent with α . The result is a new junction tree rooted at \mathcal{C} , with the same underlying undirected topology as the original tree.

Rerooting a junction tree at certain cliques leads to the acceleration of evidence propagation on parallel computing systems. For an arbitrary junction tree, the *critical path* (CP) is defined as a longest weighted path from the root to a leaf clique, where the *path weight* is the sum of the computational complexity of the cliques in the path. Rerooting a junction tree at various cliques can result in different critical paths. The one with the *minimum* critical path leads to the fastest evidence propagation on platforms with sufficient parallel processing units. For example, assuming each clique in Figure 4 (a) has a unit of computational complexity, then the complexity of the critical path is 5, since there are 5 cliques in the longest path. However, if we reroot the junction tree at Clique 2 (Figure 4 (b)), the complexity of the critical path becomes 4. If there are enough parallel processing units available, we can update the cliques at the same level in parallel. For example, Cliques 2 and 3 in Figure 4 (a) can be updated in parallel; Cliques 0, 3, 6 and 7 in Figure 4 (b) can be updated in parallel too. Thus, when a sufficient number of parallel processing units are available, evidence propagation in Figure 4 (b) is faster than that in Figure 4 (a).

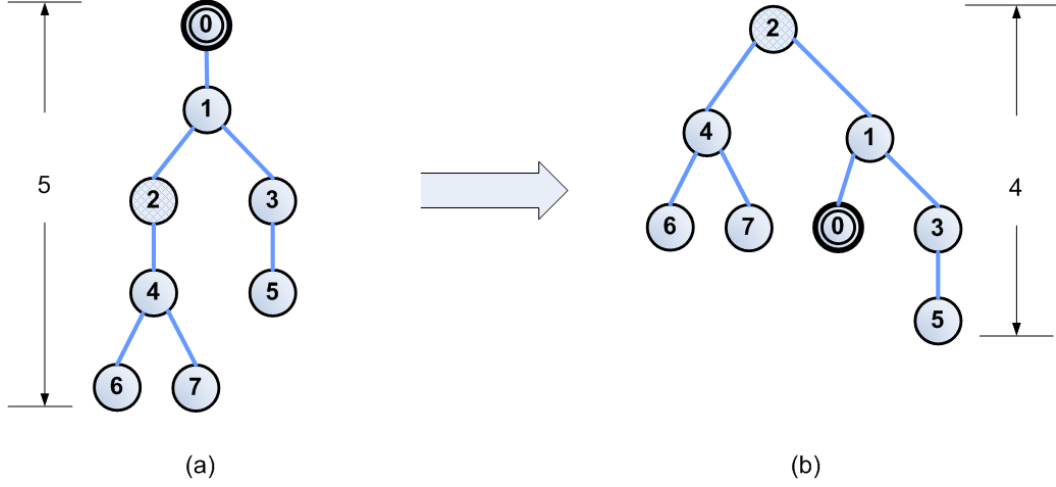


Figure 4: (a) A sample junction tree where Clique 0 is the root. The length of the critical path is 5. (b) A new junction tree after rerooting, where Clique 2 is the root. The length of the critical path is 4.

4.2.2 A Straightforward Rerooting Method

We introduce several notations for junction tree rerooting. According to Eq. (2) and the analysis for Algorithm 1, the clique complexity L_{C_i} for C_i is given by:

$$L_{C_i} = k_i w_{C_i} \prod_{j=1}^{w_{C_i}} r_j \quad (3)$$

where w_{C_i} is the clique width; k_i is the number of children of C_i and r_j is the number of states of the j -th random variable in clique C_i . Let $P(C_i, C_j) = C_i, C_{i_1}, C_{i_2}, \dots, C_j$ denote a path in a junction tree, starting at Clique C_i and terminating at C_j . Based on Eq. (3), the *path complexity* of $P(C_i, C_j)$, denoted $L_{(C_i, C_j)}$, is defined as the sum of the clique complexity of all the cliques in the path, that is,

$$L_{(C_i, C_j)} = \sum_{C_t \in P(C_i, C_j)} L_{C_t} = \sum_{C_t \in P(C_i, C_j)} k_t w_{C_t} \prod_{l=1}^{w_{C_t}} r_l \quad (4)$$

where w_{C_t} is the clique width and r_l is the number of states of the l -th random variable in clique C_t .

Algorithm 2 Straightforward root selection for minimizing critical path

Input: Junction tree J with N cliques

Output: new root C_r

- 1: **for all** $C_i \in J, \forall i = 1, 2, \dots, N$ **do**
 - 2: Let $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_N)$ be a preorder walk of the underlying undirected tree of J starting at C_i
 - 3: **for** $j = N$ to 1 **do**
 - 4: $W_{\alpha_j}^{(i)} = L_{C_{\alpha_j}} + \max_k (W_{\alpha_k}^{(i)}) \quad \forall C_{\alpha_k} \in \text{Child}(C_{\alpha_j})$
 - 5: **end for**
 - 6: **end for**
 - 7: select new root C_r s.t. $W_{\alpha_1}^{(r)} = \min_i W_{\alpha_1}^{(i)} \quad \forall i = 1, 2, \dots, N$
-

As a comparison with the efficient junction tree rerooting method in the next section, we present a straightforward

ward method to perform rerooting in Algorithm 2. Line 2 obtains α by perform breadth first search (BFS) starting at \mathcal{C}_i in the underlying undirected tree. Actually, α encodes the desired new edge directions of a junction tree rooted at \mathcal{C}_i . Lines 3-5 compute the path complexity of the critical path in each rerooted tree, where $W_{\alpha_j}^{(i)}$ denotes the path complexity of a critical path in the subtree rooted at \mathcal{C}_{α_j} . Thus, $W_{\alpha_1}^{(i)}$ gives the path complexity of a critical path for the entire tree. Notice that, when \mathcal{C}_{α_j} is a leaf clique, $\max_k(W_{\alpha_j}^{(i)})$ returns 0. Line 7 selects a clique corresponding to the rerooted junction tree with minimum critical path.

We briefly analyze the serial complexity of Algorithm 2. Both for-loops involve N iterations. Line 2 performs BFS using N time and Line 4 takes $O(w_{\mathcal{C}_{\alpha_j}})$ time to compute $L_{\mathcal{C}_{\alpha_j}}$, and $O(d_{\mathcal{C}_{\alpha_j}})$ time to compute $\max_k(W_{\alpha_j}^{(i)})$, where $d_{\mathcal{C}_{\alpha_j}}$ is the number of children of \mathcal{C}_{α_j} . Note $d_{\mathcal{C}_{\alpha_j}} \leq w_{\mathcal{C}_{\alpha_j}}$ for any junction tree. Line 7 takes N comparisons to select the new root. Therefore, the computational complexity for Algorithm 2 is $O(N^2 w_{\mathcal{C}_{\alpha_j}})$. When N is large, the straightforward rerooting method is not efficient.

Algorithm 2 can be parallelized by calculating the critical path complexity for each rerooted tree separately, i.e. replacing the for-loop in Line 1 by **for-pardo**. The computation complexity for the parallel version is $O(N^2 w_{\mathcal{C}_{\alpha_j}}/P)$, if there are P parallel processing units. However, such a parallelization is not suitable for the Cell, since Lines 2-5 involve many branches with few computation. Thus, the overhead can be dominant due to the branch curse of the SPEs in the Cell. Therefore, we develop an efficient **serial** algorithm for junction tree rerooting on the PPE.

4.2.3 An Efficient Junction Tree Rerooting Method

The key step of junction tree rerooting is to select a clique as the new root, which leads to a junction tree with the minimum critical path. Let $P(\mathcal{C}_x, \mathcal{C}_y)$ denote a path from \mathcal{C}_x to \mathcal{C}_y , and $L_{(\mathcal{C}_x, \mathcal{C}_y)}$ the sum of the clique complexity of all the cliques in path $P(\mathcal{C}_x, \mathcal{C}_y)$ (see Eq. 4). We develop an efficient algorithm to select such a root based on the following lemma:

Lemma 1: Suppose that $P(\mathcal{C}_x, \mathcal{C}_y)$ is the longest weighted path in a given junction tree, where \mathcal{C}_x and \mathcal{C}_y are leaf cliques, and $L_{(\mathcal{C}_r, \mathcal{C}_x)} \geq L_{(\mathcal{C}_r, \mathcal{C}_y)}$, where \mathcal{C}_r is the root. Then $P(\mathcal{C}_r, \mathcal{C}_x)$ is a critical path in the given junction tree.

Proof Sketch: Assume the critical path is $P(\mathcal{C}_r, \mathcal{C}_z)$ where $\mathcal{C}_z \neq \mathcal{C}_x$. Let $P(\mathcal{C}_r, \mathcal{C}_{b1})$ denote the longest common subpath between $P(\mathcal{C}_r, \mathcal{C}_x)$ and $P(\mathcal{C}_r, \mathcal{C}_y)$, where the common subpath means that, $\forall \mathcal{C}_i \in P(\mathcal{C}_r, \mathcal{C}_{b1})$, we have $\mathcal{C}_i \in P(\mathcal{C}_r, \mathcal{C}_x)$ and $\mathcal{C}_i \in P(\mathcal{C}_r, \mathcal{C}_y)$. Let $P(\mathcal{C}_r, \mathcal{C}_{b2})$ be the longest common subpath between $P(\mathcal{C}_r, \mathcal{C}_x)$ and $P(\mathcal{C}_r, \mathcal{C}_z)$. Without loss of generality, we assume $\mathcal{C}_{b2} \in P(\mathcal{C}_r, \mathcal{C}_{b1})$. Since $P(\mathcal{C}_r, \mathcal{C}_z)$ is a critical path, we have $L_{(\mathcal{C}_r, \mathcal{C}_z)} \geq L_{(\mathcal{C}_r, \mathcal{C}_x)}$. Because $L_{(\mathcal{C}_r, \mathcal{C}_z)} = L_{(\mathcal{C}_r, \mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2}, \mathcal{C}_z)}$ and $L_{(\mathcal{C}_r, \mathcal{C}_x)} = L_{(\mathcal{C}_r, \mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_x)}$, we have $L_{(\mathcal{C}_{b2}, \mathcal{C}_z)} \geq L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} +$

$L_{(\mathcal{C}_{b1}, \mathcal{C}_x)} > L_{(\mathcal{C}_{b1}, \mathcal{C}_x)}$. Thus, we can find path $P(\mathcal{C}_z, \mathcal{C}_y) = P(\mathcal{C}_z, \mathcal{C}_{b2})P(\mathcal{C}_{b2}, \mathcal{C}_{b1})P(\mathcal{C}_{b1}, \mathcal{C}_y)$, which leads to:

$$\begin{aligned} L_{(\mathcal{C}_z, \mathcal{C}_y)} &= L_{(\mathcal{C}_z, \mathcal{C}_{b2})} + L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_y)} > L_{(\mathcal{C}_{b1}, \mathcal{C}_x)} + L_{(\mathcal{C}_{b2}, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_y)} \\ &> L_{(\mathcal{C}_x, \mathcal{C}_{b1})} + L_{(\mathcal{C}_{b1}, \mathcal{C}_y)} = L_{(\mathcal{C}_x, \mathcal{C}_y)} \end{aligned} \quad (5)$$

The above inequality contradicts the assumption that $P(\mathcal{C}_x, \mathcal{C}_y)$ is the longest weighted path in the given junction tree. Thus, $P(\mathcal{C}_r, \mathcal{C}_x)$ must be a critical path. \square

Algorithm 3 Root selection for minimizing critical path

Input: Junction tree \mathcal{J}

Output: New root \mathcal{C}_r

- 1: initialize a tuple $\langle v_i, p_i, q_i \rangle = \langle k_i w_{\mathcal{C}_i} \prod_{j=1}^{w_{\mathcal{C}_i}} r_j, 0, 0 \rangle$ for each \mathcal{C}_i in \mathcal{J}
 - 2: **for** $i = N$ **downto** 1 **do**
 - 3: $p_i = \arg_j \max(v_j), \forall pa(\mathcal{C}_j) = \mathcal{C}_i$
 - 4: $q_i = \arg_j \max(v_j), \forall pa(\mathcal{C}_j) = \mathcal{C}_i$ and $j \neq p_i$
 - 5: $v_i = v_i + v_{p_i}$
 - 6: **end for**
 - 7: select \mathcal{C}_m where $m = \arg_i \max(v_i + v_{q_i}), \forall i$
 - 8: initialize path $P = \{\mathcal{C}_m\}; i = m$
 - 9: **while** \mathcal{C}_i is not a leaf clique **do**
 - 10: $i = p_i; P = \{\mathcal{C}_i\} \cup P$
 - 11: **end while**
 - 12: $P = P \cup \mathcal{C}_{q_m}; i = m$
 - 13: **while** \mathcal{C}_i is not a leaf node **do**
 - 14: $i = p_i; P = P \cup \{\mathcal{C}_i\}$
 - 15: **end while**
 - 16: denote \mathcal{C}_x and \mathcal{C}_y the two end cliques of path P
 - 17: select new root $\mathcal{C}_r = \arg_i \min |L_{(\mathcal{C}_x, \mathcal{C}_i)} - L_{(\mathcal{C}_i, \mathcal{C}_y)}| \forall \mathcal{C}_i \in P(\mathcal{C}_x, \mathcal{C}_y)$
-

According to Lemma 1, the new root can be found once we identify the longest weighted path between two leaves in the given junction tree. We introduce a tuple $\langle v_i, p_i, q_i \rangle$ for each clique \mathcal{C}_i to find the longest weighted leaf-to-leaf path (Lines 1-6, Algorithm 3), where v_i denotes the path complexity of a critical path of the subtree rooted at \mathcal{C}_i ; p_i is the index of \mathcal{C}_{p_i} , a child of \mathcal{C}_i ; and q_i is the index of another child of \mathcal{C}_i . The path from \mathcal{C}_{p_i} to a certain leaf clique in the subtree rooted at \mathcal{C}_i is the *longest* weighted path among all paths from a child of \mathcal{C}_i to a leaf clique, while the path from \mathcal{C}_{q_i} to a certain leaf clique in the subtree rooted at \mathcal{C}_i is the *second longest* weighted path. The two paths are concatenated at \mathcal{C}_i and form a leaf-to-leaf path in the original junction tree. In Algorithm 3, the tuples are initialized in Line 1 and updated in the for-loop (Lines 2-6). Notice that the indices of cliques are consistent with the preorder walk of the tree starting from the root, i.e. an edge points from \mathcal{C}_i to \mathcal{C}_j if and only if $i < j$. In Lines 3 and 4, $\arg_j \max(v_j)$ stands for the value of the given argument (parameter) j for which the value of the given expression v_j attains its maximum value. In Line 7, we detect a clique \mathcal{C}_m on the longest weighted path and identify the path in Lines 8-15 accordingly. The new root is then selected in Line 17.

We analyze the serial complexity of Algorithm 3. Line 1 take $O(w_{\mathcal{C}}N)$ computation time for initialization, where $w_{\mathcal{C}}$ is the clique width and N is the number of cliques. The loop in Line 2 has N iterations. Lines 3 and 4 each take $O(k)$ time, where k is the maximum number of children of a clique. Line 7 takes $O(N)$ time, as do Lines 8-15, since a path consists of at most N cliques. Lines 16-17 can be completed in $O(N)$ time. Since $k < w_{\mathcal{C}}$, the complexity of Algorithm 3 is $O(w_{\mathcal{C}}N)$, compared to $O(w_{\mathcal{C}}N^2)$, the complexity of the straightforward approach.

Algorithm 3 can be parallelized using techniques such as pointer jumping. However, the parallelized version involves many branches and limited computation. Due to the branch overhead in SPEs, we perform rerooting sequentially on the PPE. When r and $w_{\mathcal{C}}$ are large, the execution time for rerooting is very small compared to the evidence propagation discussed in Sections 4.4 and 4.5.

We would like to emphasize that, although junction tree rerooting can provide more parallelism for exact inference, it is an optional step for parallel exact inference. In addition to the parallelism provided by the structure of junction trees, we utilize the node level parallelism of junction trees, which will be addressed in the next section. In the following sections, we assume that junction tree rerooting has been applied during preprocessing.

4.3 Dynamic Scheduling of Cliques

4.3.1 Tasks and Task Partitioning

In our context, a *task* (denoted T) is defined as the computation to update a clique using the input separators and then generate the output separators. Each clique in the junction tree is related to *two* tasks, one for evidence collection and the other for evidence distribution. The data required by a task consist of input separators, a (partial) clique potential table and the output separators (see Figure 5 (a)). In evidence collection, the input separators for clique \mathcal{C} are the separators between \mathcal{C} and its children, i.e. $\mathcal{S}_{ch_i(\mathcal{C})}$ for all i . The output separators are the separators between \mathcal{C} and its parent, $\mathcal{S}_{pa(\mathcal{C})}$. In evidence distribution, the input and output separators are switched, as shown in Figure 5.

Due to the limited size of the local store, some tasks involving large potential tables cannot be completely loaded into an SPE. Clique potential table size increases dramatically with the clique width and the number of variable states. However, the local store of the SPE in the Cell is limited to 256 KB, shared by both instruction and data. If double buffering is used to overlap the data transfer and computation, the available local store to accommodate a task is even more limited.

We propose a scheduler to check the size of tasks to be executed. The scheduler partitions each large task into a series of small tasks to fit in the local store. Assume task T involves a large potential table $\psi_{\mathcal{C}}$, which is too large to fit in the local store. We evenly partition $\psi_{\mathcal{C}}$ into k portions ($k > 1$) and ensure that each portion can be completely

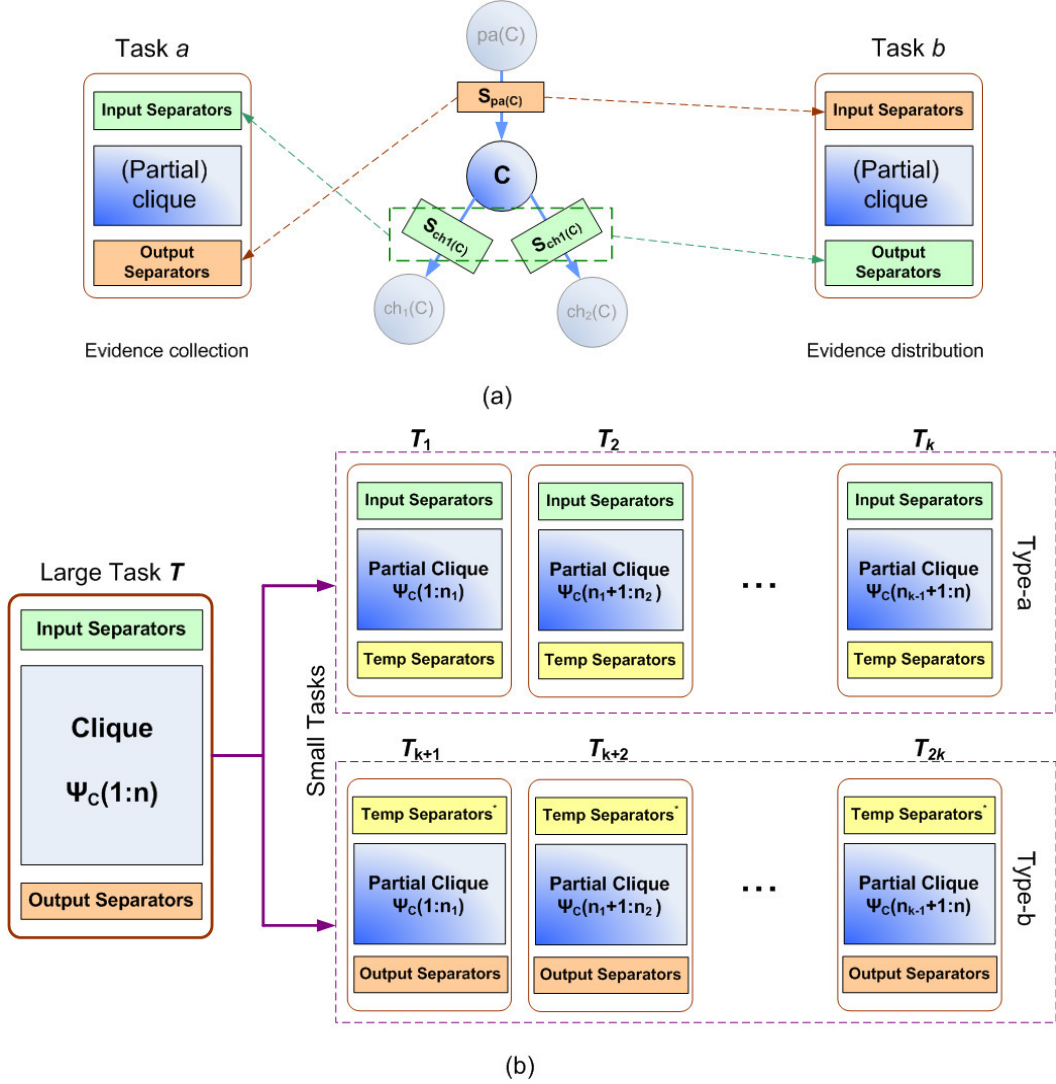


Figure 5: (a) Illustration of the relationship between a clique \mathcal{C} and the two tasks related to \mathcal{C} ; (b) Partitioning of a large task into a series of small tasks.

loaded into a single SPE. For each part, the scheduler creates two small tasks. The reason why we need two tasks to process a portion will be addressed in Section 4.4. Therefore, the scheduler partitions task T into *two* sets of small tasks, each set having k small tasks. Every small task contains n/k entries in ψ_C , where n is size of ψ_C . We denote the $2k$ small tasks T_1, T_2, \dots, T_{2k} . We call T_1, T_2, \dots, T_k *type-a* tasks of the original task T ; $T_{k+1}, T_{k+2}, \dots, T_{2k}$ are called *type-b* tasks of T (see Figure 5 (b)). The original task T is called a *regular* task. The input separators of *type-a* tasks are identical to those of the original task T . The output separators of *type-a* tasks are called *temporary separators*. The temporary separators of all *type-a* tasks must be *accumulated* as the input separators for each *type-b* task. Thus, no *type-b* task can be scheduled for execution until all the corresponding *type-a* tasks are processed. Separators accumulation is defined as calculating the sum of corresponding entries of all input separators. The output

separators of all the *type-b* tasks are also *accumulated* as the final output separators (the output separators of the original task T). Figure 5 (b) shows a sample large task T and $2k$ small tasks partitioned from T .

4.3.2 Task Dependency Graph

For the sake of scheduling tasks for both the evidence collection and distribution, we create a *task dependency graph* G according to the given junction tree (see the left hand side figure in Figure 6 (a)). Each node in G denotes a task. Each edge in G indicates the dependency between two tasks. A task can not be scheduled for execution until all dependent tasks are processed. Note that there are two subgraphs in G , which are symmetric with respect to the dashed line in Figure 6 (a): The upper subgraph is the given junction tree with all edges from children to their parents; the lower subgraph is the the given junction tree with all edges from parents to their children. The upper and lower subgraphs correspond to evidence collection and evidence distribution, respectively. Thus, each clique except the root in the junction tree is related to two nodes in the task dependency graph. Note that we do not duplicate potential tables during the construction of the task dependency graph, though each clique corresponds to two tasks. We need only to store the location of the potential table in the task.

As some tasks involving large potential tables are partitioned into a series of tasks, the task dependency graph G is modified accordingly. The scheduler is in charge of task partitioning and task dependency graph modification at runtime. We assume the bold nodes in G (see the left side figure in Figure 6 (a)) involve large potential tables. After task partitioning, the bold nodes are replaced by sets of small nodes shown in the dashed boxes (see the right hand side in Figure 6 (a)). Each node in a dashed box denotes a small task partitioned from the original task. All the *type-a* tasks are connected to the parent of the original task. Each *type-b* task depends on all the *type-a* tasks. The children of the original task depend on all the *type-b* tasks.

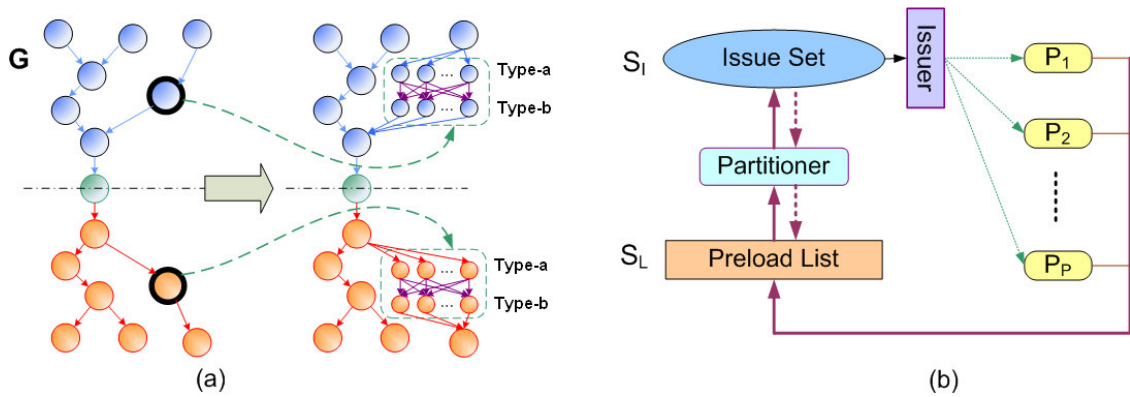


Figure 6: (a) An example of a task dependency graph for the junction tree given in Figure 3 and the corresponding task dependency graph with partitioned tasks. (b) Illustration of the scheduling scheme for exact inference.

4.3.3 Dynamic Partitioning and Scheduling

Scheduling task dependency graphs has been extensively studied (for example, see [20]). In this paper, we use an intuitive but efficient method to schedule tasks to SPEs. The input to the scheduler is an arbitrary junction tree. Initially, the scheduler constructs a task dependency graph G , according to the given junction tree. As discussed in Section 4.3.2, it is straightforward to construct G using the structure of the given junction tree.

The components of the scheduler are shown in Figure 6 (b). *Issue set* S_I is a set of tasks that are ready to be processed, i.e. for any clique $\mathcal{C} \in S_I$, the dependent cliques of \mathcal{C} given in G have been processed. The *preload list* S_L consists of all the tasks that are not ready to be processed. The *partitioner* is a crucial component of the proposed scheduler, as it dynamically explores parallelism at a finer granularity. The function of the partitioner is to check and partition tasks. In Figure 6 (b), P_1, P_2, \dots, P_P denote processing units i.e. the SPEs in the Cell. The *issuer* selects a task from S_I and allocates an SPE to it. Both the *solid arrows* and *dashed arrows* in Figure 6 (b) illustrate the data flow path in the scheduler.

The scheduler issues tasks to processing units as follows. Initially, S_I contains tasks related to the leaf cliques of the junction tree. These tasks have no parent in the task dependency graph G . Other tasks in G are placed in S_L . Each task in S_L has a property called the *dependency degree*, which is defined as the number of parents of the task in G . The issuer selects a task from S_I and issues the task to an available processing unit, repeating issuing if idle processing units exist. Several strategies for selecting tasks have been studied [21, 22]. We use a straightforward strategy where the task in S_I with largest number of children is selected. When a processing unit P_i is assigned to task T , it loads data relevant to T - including input separators, clique potential tables and output separators - from the main memory. Once P_i completes task T , P_i notifies S_L and waits for the next task. S_L receives notification from P_i and decreases the dependency degree of all tasks that directly depend on task T . If \tilde{T} is dependent upon T , and the dependency degree of \tilde{T} becomes 0 after T is processed, then \tilde{T} is moved from S_L to S_I . When \tilde{T} is moving from S_L to S_I , the partitioner checks if the data size of \tilde{T} can fit in the local store, and partitions \tilde{T} into a set of smaller tasks accordingly. If \tilde{T} is partitioned, all the *type-a* tasks generated from \tilde{T} are assigned to S_I , while *type-b* tasks are placed in S_L .

A feature of the proposed scheduler is that it dynamically exploits parallelism at finer granularity. The scheduler tracks the status of the processing units and the size of S_I . If several processing units are idling, and the size of S_I is smaller than a given threshold, the partitioner picks the largest regular task in S_I , and partitions the task into smaller tasks, even though the task can fit in the local store. The reason is that small S_I can not provide enough parallel tasks to keep all the SPEs busy, and idle SPEs adversely affect the performance. In Figure 6 (b), the dashed

arrows illustrate that a regular task is taken from S_I and get partitioned at runtime. After partitioning the task, *type-a* tasks are sent back to S_I while *type-b* tasks are placed into S_L .

4.4 Potential Table Organization and Efficient Primitives

We carefully organize the potential tables to enhance the performance of the computations. We define some terms to explain the potential table organization: We assign an order to the random variables in a clique to form a *variable vector*. We will discuss how to determine the order later in this section. For a clique \mathcal{C} , the corresponding variable vector is denoted $V_{\mathcal{C}}$. Accordingly, the combination of the states of the variables in a variable vector forms *state strings*. Assuming a clique consists of w variables, each having r states, there are r^w state strings for the clique. Each state string corresponds to an entry in the clique potential table, which is stored in memory as a 1-d array. For instance, given a state string $S = (s_1 s_2 \cdots s_w)$, where $s_i \in \{0, 1, \dots, r-1\}$ is the state of the i^{th} variable in the variable vector, we convert S to an *entry index* t of the potential table using $t = \sum_{j=1}^w s_j r^j$. Given an index t , we can also convert t to a state string by computing $s_i = \lfloor t/r^{i-1} \rfloor \% r$ for each s_i in S , where $\%$ is the modulo operator. Thus, we store the propability (potential) corresponding to state string S to the t^{th} entry of the potential table. Figure 7 (a) shows a sample clique \mathcal{C} with binary variable vector (a, b, c, d, e, f) . The potential table is given in Figure 7 (b). Notice that we need only to store potential table $\psi_{\mathcal{C}}$ in memory.

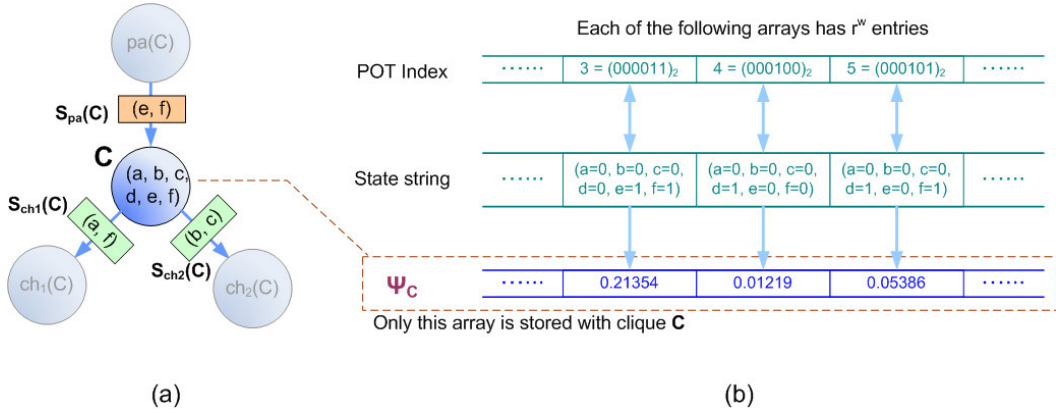


Figure 7: (a) A sample clique and its variable vector; (b) The relationships among array index, state string and potential table. For the sake of illustration, we assume that all the random variables are binary variables.

Assigning a proper order to the random variables in variable vectors ensures data locality in potential table computation. We order the variables using the following method: given a clique \mathcal{C} , the variable vector is ordered by $V_{\mathcal{C}} = (V_{\mathcal{C} \setminus S_{pa}(\mathcal{C})}, V_{S_{pa}(\mathcal{C})})$, where $V_{S_{pa}(\mathcal{C})}$ is the variable vector of the separator between \mathcal{C} and its parent, and $V_{\mathcal{C} \setminus S_{pa}(\mathcal{C})}$ consists of the remaining variables. The variable order within $V_{\mathcal{C} \setminus S_{pa}(\mathcal{C})}$ and $V_{S_{pa}(\mathcal{C})}$ is arbitrary.

We describe the advantage of the above variable ordering method by analyzing the computation of potential

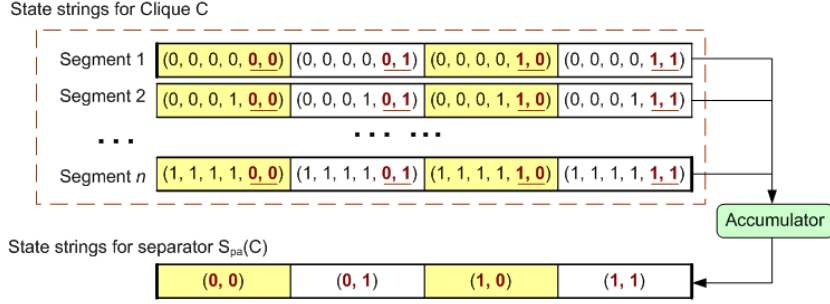


Figure 8: The relationship between entries of ψ_C and entries of $\psi_{S_{pa}(C)}$ in Figure 7 (a). Each entry of $\psi_{S_{pa}(C)}$ is the sum of the corresponding entries in all segments.

tables. Eq. (1) formulates exact inference as the computation of potential tables, including *marginalization*, *division* and *multiplication*. These computations are also known as *node level primitives*, which are defined in [8]. In this section, we propose an efficient implementation for the node level primitives that uses vectorized algebraic operations. Marginalization is used to obtain separator potential tables from a clique potential table. For example, in Figure 7 (a), we marginalize ψ_C to obtain $\psi_{S_{pa}(C)}$. Since $V_{S_{pa}(C)}$ is the lower part of V_C , the relationship between the entry in $\psi_{S_{pa}(C)}$ and the entry in ψ_C is straightforward (see Figure 8). Segment i of ψ_C is $\psi_C((i-1)|\psi_{pa}(C)| : (i|\psi_{pa}(C)|-1))$ i.e. an array consists of entries from the $(i-1)|\psi_{pa}(C)|^{th}$ to the $(i|\psi_{pa}(C)|-1)^{th}$ elements of ψ_C . Thus, this marginalization can be implemented by accumulating all the segments, without checking the variable states for each entry of the potential table. As potential table division always occurs between the updated and stale versions of the same separator potential table, the two potential tables have identical variable vectors. Thus, potential table division can be implemented by entry-wise division without checking the variable states for each entry. Potential table multiplication occurs between ψ_C and $\psi_{S_{pa}(C)}^\dagger$, where $\psi_{S_{pa}(C)}^\dagger = \frac{\psi_{S_{pa}(C)}^*}{\psi_{S_{pa}(C)}}$. Since $\psi_{S_{pa}(C)}^\dagger$ has the same state strings as $\psi_{S_{pa}(C)}$, the relationship between $\psi_{S_{pa}(C)}^\dagger$ and $S_{pa}(C)$ is very similar to that in Figure 8. Thus, multiplication can be implemented by multiplying each entry of $\psi_{S_{pa}(C)}^\dagger$ into the corresponding entries in all the segments.

However, marginalization to obtain $\psi_{S_{ch_i}(C)}$ - the separator potential table for the i^{th} child - requires more computation than that to obtain $\psi_{S_{pa}(C)}$. The reason is we need to identify the mapping relationship between $V_{S_{ch_i}(C)}$ and V_C . We define the *mapping vector* to represent the mapping relationship from V_C to $V_{S_{ch_i}(C)}$: The mapping vector is defined as $M_{ch_i(C)} = (m_1 m_2 \cdots m_w | m_j \in \{0, 1, \cdots, w_{S_{ch_i}}\})$, where w is the width of clique C and $w_{S_{ch_i}}$ is the length of $V_{ch_i(C)}$. m_j is defined as that the j^{th} variable in V_C mapped to the m_j^{th} variable in $V_{S_{ch_i}(C)}$. $m_j = 0$ if the j^{th} variable in V_C is not in $V_{S_{ch_i}(C)}$. Using the mapping vector $M_{ch_i(C)}$, we can identify the relationship between ψ_C and $\psi_{S_{ch_i}(C)}$. Given an entry $\psi_C(t)$, we convert index t to a state string $S = (s_1 s_2 \cdots s_w)$. Then, we construct a new state string \tilde{S} by assigning $s_i \in S$ to \tilde{s}_{m_i} if $m_i \neq 0$. The new state string \tilde{S} is then converted back to an index \tilde{t} . Therefore, $\psi_C(t)$ corresponds to $\psi_{S_{ch_i}(C)}(\tilde{t})$. To compute $\psi_{S_{ch_i}(C)}$ from ψ_C , we just need to identify the

relation for each t and accumulate $\psi_{\mathcal{C}}(t)$ to $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}(\tilde{t})$.

The size of potential table increases dramatically with the clique width and the number of states of variables. However, the local store of each SPE in the Cell is limited to 256 KB. In addition, using double buffering to overlap the data transfer and computation makes the available local store more limited. For example, assuming single precision and binary random variables are used, the width of a clique that can fit in LS should not be more than 14 ($14 = \lfloor \log_2(128KB/4) \rfloor$). If the potential table of a clique is too large to fit in the local store, the scheduler present in Section 4.3 partitions the clique into n/k portions, where n is the size of the potential table and k is the size of the portion which can fit in the local store. Each portion is processed by an SPE. However, as marginalization must sum up entries that may be distributed to all portions, the partial marginalization results from each portion must be accumulated. According to Algorithm 1, accumulation is needed after Lines 5 and 10. Thus, for each portion of the potential table, we create two small tasks (*type-a* and *type-b* tasks). Accumulation is applied after both *type-a* tasks and *type-b* tasks are processed.

4.5 Data Layout for Optimizing Cell Performance

To perform exact inference in a junction tree, we must store the following data in memory: the structure of the junction tree, the clique potential table for each clique in the junction tree, the separator in each edge, and the properties of the cliques and separators (such as clique width, table size, etc.). For the Cell, the optimized data layout helps data transfer between main memory and local stores, and the vectorized computation in SPEs.

The data that an SPE must transfer between its local store and the main memory depend on the direction of the evidence propagation. Figure 9 (c) demonstrates the components related to evidence propagation in a clique. In evidence collection, cliques $ch_1(\mathcal{C})$ and $ch_2(\mathcal{C})$ update separators $S_{ch_1}(\mathcal{C})$ and $S_{ch_2}(\mathcal{C})$ respectively. Then, clique \mathcal{C} is updated using $S_{ch_1}(\mathcal{C})$ and $S_{ch_2}(\mathcal{C})$. Lastly, \mathcal{C} updates $S_{pa}(\mathcal{C})$. Notice that, according to Eq. (1), updating clique potential table $\psi_{\mathcal{C}}$ needs both $\psi_{\mathcal{C}}$ and the updated separator, while updating separator potential table $\psi_{S_{pa}(\mathcal{C})}$ needs only the updated $\psi_{\mathcal{C}}$. In evidence distribution, $\psi_{\mathcal{C}}$ is updated using $\psi_{S_{pa}(\mathcal{C})}$ and then the updated $\psi_{\mathcal{C}}$ is used to propagate evidence to $S_{ch_1}(\mathcal{C})$ and $S_{ch_2}(\mathcal{C})$.

We store the junction tree data in the main memory (Figure 9). The SPE issues DMA commands to transfer data between the local store and main memory. For the sake of decreasing the DMA transfer overhead, we minimize the number of DMAs arising from each SPE. If all the data required for processing a clique are stored in continuous locations in main memory, the SPE needs only to issue one DMA command (or a DMA list, if the data size is larger than 16KB) to load or save all the data. However, note that a separator is shared by two cliques. If all data needed for processing a clique are stored in contiguous locations, separators must be duplicated. In addition, to maintain

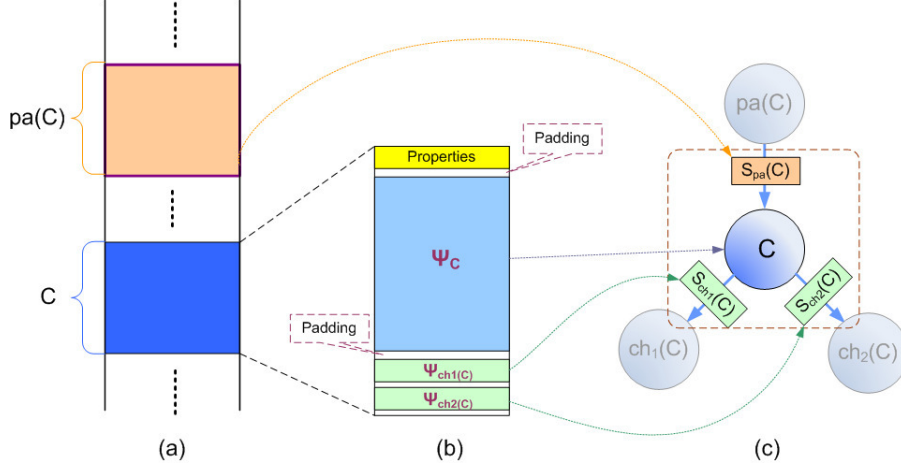


Figure 9: (a) Data layout for junction tree in main memory; (b) Data layout for each clique; (c) Relationship between the layout and partial junction tree structure.

the consistency of the copies of separators causes extra memory access. Considering a clique has only one parent and several children, we store a clique together with all the child separators, but leave the parent separator to the parent of the clique. Each clique corresponds to a block in Figure 9 (a), while each block consists of a clique potential table, child separators, the properties of the clique, and child separators (Figure 9 (b)). All blocks are 16-byte aligned for DMA transfer. We also insert paddings to ensure each potential table within a block is also 16-byte aligned. Such alignment benefits the computation in SPEs [12]. When the PPE schedules a clique C to an SPE, the PPE sends the starting address of the data block of clique C and the starting address of the parent separator to the SPE.

4.6 Complete Algorithm for Parallel Exact Inference

The parallel exact inference algorithm proposed in this paper consists of task dependency graph scheduling (Section 4.3) and potential table update (Sections 4.4). For the Cell, PPE is responsible for the task scheduling, and SPEs perform the potential table update.

Using the scheduling definition and notations in Section 4.3, we present an algorithm for task dependency graph scheduling in Algorithm 4. Lines 1 and 2 in Algorithm 4 are initial steps. Since all tasks in S_I are ready to be processed, Lines 4-9 issue the tasks to available SPEs. Lines 10-14 check if the size of S_I is less than a given threshold δ_S , which is a small constant. If $|S_I| < \delta_S$, some SPEs may keep idling, since there are not enough parallel tasks. In Section 5, we let δ_S be the number of SPEs. If S_I is less than δ_S , the largest regular task in S_I is partitioned into a set of small tasks, so that there are enough parallel tasks for SPEs. For each completed task in the SPEs, Line 15 adjusts the dependency degree for child tasks. If the dependency degree of a child task becomes 0, the task is moved from S_L to S_I (Lines 19-24). In Line 19, δ_T is a constant ensuring any tasks smaller than δ_T can fit in the local

store. If the task to be moved is too large to fit in the local store, the scheduler partitions it (Lines 22-23).

Algorithm 4 PPE: Task dependency graph scheduling

Input: Junction tree \mathbb{J} , size of clique potential tables and separators, thresholds δ_T, δ_S , number of SPEs P

Output: Scheduling sequence for exact inference

```

1: Generate task dependency graph  $G$  from (rerooted)  $\mathbb{J}$ 
2:  $S_I = \emptyset; S_L = \{\text{all tasks in } G\}$ 
3: while  $S_I \cup S_L \neq \emptyset$  do
4:   for  $i = 1$  to  $\min(|S_I|, P)$  do
5:     if  $SPE_i$  is idling then
6:        $T = \text{get a task in } S_I; S_I = S_I \setminus T$ 
7:       assign task  $T$  to  $SPE_i$ 
8:     end if
9:   end for
10:  if  $|S_I| < \delta_S$  and  $S_I$  contains regular tasks then
11:     $T = \text{the largest regular task in } S_I$ 
12:    partition  $T$  into small task sets  $T_{type-a}, T_{type-b}$ 
13:     $S_I = S_I \cup T_{type-a}; S_L = S_L \cup T_{type-b}$ 
14:  end if
15:  for  $T \in \{\text{completed tasks in all SPEs}\}$  do
16:    for  $\tilde{T} \in \{\text{children of } T\}$  do
17:      decrease the dependency degree of  $\tilde{T}$  by 1
18:      if dependency degree of  $\tilde{T} = 0$  then
19:        if  $|\tilde{T}| < \delta_T$  then
20:           $S_L = S_L \setminus \{\tilde{T}\}; S_I = S_I \cup \{\tilde{T}\}$ 
21:        else
22:          partition  $\tilde{T}$  into small task sets  $\tilde{T}_{type-a}, \tilde{T}_{type-b}$ 
23:           $S_I = S_I \cup \tilde{T}_{type-a}; S_L = S_L \cup \tilde{T}_{type-b}$ 
24:        end if
25:      end if
26:    end for
27:  end for
28: end while

```

We analyze the complexity of Algorithm 4 using the model of computation presented in Section 2.3, where the computation complexity for PPE is denoted $T_C^{(PPE)}$. In Line 1 of Algorithm 4, rerooting \mathbb{J} is an optional preprocessing step. Its complexity is addressed in Section 4.2. Line 1 first takes $O(N)$ time to copy the junction tree structure and reverse the direction of edges. Note that there are $(N - 1)$ edges for a junction tree with N cliques. Then, Line 1 connects the original structure of the junction tree and that with reversed edges to form a task dependency graph G . This takes $O(1)$ time. Line 2 takes $O(1)$ time to initialize S_I and $O(N)$ time to put $2N - 1$ tasks to S_L . In Line 3, the size of $S_L \cup S_I$ is bounded by Nr^w/δ_m . Lines 4-9 perform a loop with $O(P)$ iterations, where P is the number of SPEs. In each iteration, Lines 6 and 7 both take constant time for computation. However, data transfer is invoked by Line 7, which will be addressed in the analysis of Algorithm 5. Line 8 takes $O(1)$ time to judge if the condition is satisfied. Selecting the largest regular task in Line 11 takes $O(1)$ time if tasks are organized as a heap according to their sizes. Assume tasks smaller than δ_m can not be further decomposed ($\delta_m \leq \delta_T$); then a task T is

split into at most $2T/\delta_m$ small tasks. Thus, Line 12 takes $O(|T|/\delta_m) = O(r^w/\delta_m)$ time to decompose T to $2T/\delta_m$ small tasks, and Line 13 takes $O(|T|/\delta_m)$ time to insert the small tasks into the task dependency graph. Line 15 checks P SPEs if their current tasks have been completed. Assuming each task has at most k children, Lines 15-27 consist of $O(Pk)$ iterations. For each completed task, the dependency of each child is decreased by 1 using constant time. Line 20 takes $\log |S_L|$ time to remove a task from S_L , and $\log |S_I|$ time to insert a task into S_I . The size of S_L is bounded by $O(N)$. $O(\sum_{i=1}^N \prod_{j=1}^{w_{c_j}} r_{i,j}/\delta_m) = O(Nr^w/\delta_m)$ is a loose upper bound on the size of S_I , where r is the maximum number of states of random variables, and w is the maximum clique width. Line 22 takes $O(|\tilde{T}|/\delta_T)$ time to decompose \tilde{T} . Note $|\tilde{T}| \leq r^w$. Similar to Line 13, Line 23 takes $O(|\tilde{T}|/\delta_T) = O(r^w/\delta_T)$ time to insert the small tasks into the task dependency graph. Let t denote the maximum number of small tasks partitioned from a regular task. The upper bound on t is given by $O(r^w/\delta_m)$. Based on the above analysis, the computation complexity for Algorithm 4 is given by:

$$\begin{aligned} T_C^{(PPE)} &= O(2N + Nr^w/\delta_m(P(r^w/\delta_m + 1) + 2r^w/\delta_m + Pk(1 + r^w/\delta_T + \log Nr^w/\delta_T))) \\ &= O(Nt(Pt + 2t + Pk(t + \log Nt))) = O(NPkt(t + \log N)) \approx O(PktN \log N) \end{aligned} \quad (6)$$

where we assume $N \gg t$ for the sake of simplification.

Using the data layout in Section 4.5, we present an algorithm for updating potential tables (Algorithm 5). Two computation kernels for potential table updating are shown in Algorithms 6 and 7. Double buffering is used in Algorithm 5 to overlap computation and data transfer. While loading data relevant to \tilde{T} , we perform computations on task T . Lines 5-7 compute ψ_{temp} for regular and *type-a* tasks. ψ_{temp} is the output for *type-a* tasks, but is an intermediate result for regular tasks (see Lines 8-12 in Algorithm 5). Lines 13-19 process T using one of the two computation kernels, depending on the status of clique \mathcal{C} . Line 21 notifies the PPE to prepare a new task for this SPE.

In Algorithm 5, assume a regular task is decomposed into t small tasks on average and all tasks are distributed to SPEs evenly. Therefore, the size of each task can be represented by $O(\sum_{i=1}^N \prod_{j=1}^{w_{c_j}} r_{i,j}/(Nt)) = O(r^w/t)$ and each SPE processes Nt/P tasks, where w is the maximum clique width and r is the maximum number of states for random variables. Line 1 transfers $O(r^w/t)$ data from main memory to the local store of each SPE. Lines 2-23 form the main body of this algorithm, which consists of a loop with $(Nt/P - 1)$ iterations. Line 3 starts data transfer for the next task processed on the SPE. The data size for \tilde{T} is also $O(r^w/t)$. Line 6 marginalizes a (partial) potential table $\psi_{\mathcal{C}}$, if the condition in Line 5 is satisfied. Using the analysis for the computation complexity of node level primitives in Section 4.1, we know the computation complexity for Line 6 is $O(|\psi_{\mathcal{C}}|w) = O(r^w w/t)$, where w_S is the maximum separator width. Regardless of the condition in Line 8, performing either Line 9 or Line 11 takes

Algorithm 5 SPE: process tasks using double buffering

Input: task T and its relevant data $\psi_{in}, \psi_{\mathcal{C}}, \psi_{out}$; task \tilde{T} and its relevant data

Output: updated data for T and \tilde{T}

```
1:  $T =$  receive a task from PPE
2: while  $T \neq \emptyset$  do
3:    $\tilde{T} =$  receive a new task from PPE
4:   wait for the loading of  $\psi_{in}$  and  $\psi_{\mathcal{C}}$  for  $T$  to complete
5:   if  $T$  is a regular task or type-a task then
6:     marginalize  $\psi_{\mathcal{C}}$  to obtain  $\psi_{temp}$ 
7:   end if
8:   if  $T$  is a type-a task then
9:      $\psi_{out} = \psi_{temp}$ 
10:  else if  $T$  is a type-b task then
11:     $\psi_{temp} = \psi_{in}$ 
12:  end if
13:  if  $T$  is a regular task or type-b task then
14:    if clique  $\mathcal{C}$  in  $T$  has not been updated then
15:      update  $\psi_{\mathcal{C}}$  and  $\psi_{out}$  using computation kernel of evidence collection (Algorithm 6)
16:    else
17:      update  $\psi_{\mathcal{C}}$  and  $\psi_{out}$  using computation kernel of evidence distribution (Algorithm 7)
18:    end if
19:  end if
20:  store updated  $\psi_{\mathcal{C}}$  and  $\psi_{out}$  to the main memory
21:  notify PPE that task  $T$  is done
22:  let  $T = \tilde{T}$ 
23: end while
```

constant time. Lines 15 and 17 invoke computation kernels in Algorithms 4 and 5. We will analyze the complexity of Algorithm 4 and 5 later in this section. Line 20 transfers $O(r^w/t)$ data from the local store to the main memory. Note the size of ψ_S is ignored because it is smaller than that of $\psi_{\mathcal{C}}$. Lines 21 and 22 take constant time. Therefore, the number of DMA requests is $T_D = O(2Nt/P + 1) = O(Nt/P)$ and the size of transferred data is $O(Nr^w/P)$. The total computation time for Algorithm 5 is given by:

$$\begin{aligned} T_C^{(SPE)} &= O\left(\frac{Nt}{P} (|\psi_{\mathcal{C}}|w + 1 + |\psi_{\mathcal{C}}|kw)\right) \\ &= O\left(\frac{Nt}{P} \frac{r^w}{t} kw\right) = O(Nkwr^w/P) \end{aligned} \quad (7)$$

Two computation kernels (Algorithms 6 and 7) are used for evidence collection and evidence distribution, respectively. The two kernels apply Eq. 1 to the given task. Lines 1-9 in Algorithm 6 absorb evidence from each child of clique \mathcal{C} and update $\psi_{\mathcal{C}}$. Lines 4-7 implement potential table multiplication using mapping vectors (see details in Section 4.4). Lines 10-13 marginalize the updated $\psi_{\mathcal{C}}$. Benefiting from the data organization presented in Section 4.4, potential table division (Line 2) and marginalization (Line 12) are simplified to vectorized algebraic operations, which perform efficiently on SPEs as well as other SIMD machines. Algorithm 7 is similar to Algorithm 6, where potential table division (Line 1) and multiplication (Line 3) are simplified to vectorized algebraic operations.

Algorithm 6 Computation kernel of evidence collection

Input: input separators ψ_{in_i} , (partial) clique potential table ψ_C , output separator ψ_{out} , temporary separator ψ_{temp} , index offset t_δ , mapping vector M_{in}

Output: updated ψ_C and ψ_{out}

```
1: for  $i = 1$  to (Number of children of  $\mathcal{C}$ ) do
2:    $\psi_{in_i}(1 : |\psi_{in_i}|) = \psi_{in_i}(1 : |\psi_{in_i}|) / \psi_{temp}(1 : |\psi_{in_i}|)$ 
3:   for  $t = 0$  to  $|\psi_C| - 1$  do
4:     Convert  $t + t_\delta$  to  $S = (s_1 s_2 \cdots s_w)$ 
5:     Construct  $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{|in_i|})$  from  $S$  using mapping vector  $M_{in_i}$ 
6:     Convert  $\tilde{S}$  to  $\tilde{t}$ 
7:      $\psi_C(\tilde{t}) = \psi_C(\tilde{t}) * \psi_{in_i}(t)$ 
8:   end for
9: end for
10:  $\psi_{out}(1 : |\psi_{out}|) = \vec{0}$ 
11: for  $i = 1$  to  $|\psi_C|$  step  $|\psi_{out}|$  do
12:    $\psi_{out}(0 : |\psi_{out}|) = \psi_{out}(0 : |\psi_{out}|) + \psi_C(i : i + |\psi_{out}|)$ 
13: end for
```

Mapping vectors are utilized to implement potential table marginalization (Lines 6-12).

Algorithm 7 Computation kernel of evidence distribution

Input: input separator ψ_{in} , (partial) clique potential table ψ_C , output separators ψ_{out_i} , temporary separator ψ_{temp} , index offset t_δ , mapping vector M_{out_i}

Output: updated ψ_C and ψ_{out}

```
1:  $\psi_{in}(1 : |\psi_{in}|) = \psi_{in}(1 : |\psi_{in}|) / \psi_{temp}(1 : |\psi_{in}|)$ 
2: for  $i = 1$  to  $|\psi_C|$  step  $|\psi_{in}|$  do
3:    $\psi_C(i : i + |\psi_{in}|) = \psi_C(i : i + |\psi_{in}|) * \psi_{in}(1 : |\psi_{in}|)$ 
4: end for
5: for  $i = 1$  to (Number of children of  $\mathcal{C}$ ) do
6:    $\psi_{out_i}(1 : |\psi_{out_i}|) = \vec{0}$ 
7:   for  $t = 0$  to  $|\psi_C| - 1$  do
8:     Convert  $t + t_\delta$  to  $S = (s_1 s_2 \cdots s_w)$ 
9:     Construct  $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{|out_i|})$  from  $S$  using mapping vector  $M_{out_i}$ 
10:    Convert  $\tilde{S}$  to  $\tilde{t}$ 
11:     $\psi_{out_i}(\tilde{t}) = \psi_{out_i}(\tilde{t}) + \psi_C(t)$ 
12:   end for
13: end for
```

We analyze Algorithm 6 using the model proposed in Section 2.3. Line 1 is a for-loop with $O(k)$ iterations, where k is the maximum number of children of a clique. Line 2 simplifies table division by performing a vector division. Line 2 takes $O(|\psi_S|) = O(r^{w_S})$ time, where w_S is the maximum separator width and r is the maximum number of states for random variables. Using SIMD operations, the execution time for Line 2 can be speeded up by 4. However, this does not affect the complexity. Lines 3-8 form an embedded loop with $O(|\psi_C|) = O(\prod_{j=1}^{w_C} r_j) = O(r^w)$ iterations, where w is the maximum clique width. In each iteration, Line 4 takes $3w$ time to convert an index to a state string, since it computes w_C elements, each involving three scalar operations: a division, a modulo and an increase of the product of $(\prod_i r_i)$ (see [8]). Line 5 takes $O(w_S)$ time to obtain \tilde{S} from S . Line 6 is

the reverse operation to Line 4, which also takes $3w$ time. Line 8 takes $O(1)$ time to perform a multiplication between two scalar numbers. Line 10 takes constant time to initialize a vector. Lines 11-13 form another loop with $O(|\psi_C|/|\psi_{out}|) = O(r^w/r^{ws}) = O(r^{w-ws})$ iterations. The computation complexity for Lines 11-13 is $O(r^w)$. Thus, the total computation complexity of Algorithm 6 is $O(kr^w w)$.

For Algorithm 7, Line 1 takes $O(|\psi_S|) = O(r^{ws})$ time to perform vector division. Lines 2-4 take $O(|\psi_C|) = O(r^w)$ time to perform vector multiplication. The loop starting at Line 5 has k iterations. Line 6 takes constant time to initialize ψ_{out_i} . Lines 7-12 form a loop with $O(|\psi_C|) = O(r^w)$ iterations. Line 8 takes $3w$ time for computation, similar to Line 4 in Algorithm 6. Line 9 takes $O(w_S)$ time to obtain \tilde{S} . Line 10 takes $O(w_S)$ time and Line 11 takes $O(1)$ to perform scalar addition. Based on above analysis, the total computation complexity for Algorithm 7 is given by $O(kr^w w)$, which is the same as that for Algorithm 6.

5 Experiments

We conducted experiments on an IBM BladeCenter QS20 [23]. The IBM BladeCenter QS20 has two 3.2 GHz Cell BE processors with 512 KB Level 2 cache per processor. Each processor has 8 SPEs available. The two processors share a 1 GB main memory. We used one Cell processor to measure the performance. The IBM BladeCenter QS20 was installed with the Fedora Core 7, an RPM-based general purpose Linux distribution developed by the community-supported Fedora Project and sponsored by Red Hat. The Cell BE SDK 3.0 Developer package was installed for our experiments. We compiled the code using the gcc compiler provided with the SDK, with level 3 optimization.

In our experiments, we used junction trees of various sizes to analyze and evaluate the performance of our method. The junction trees were generated using Bayes Net Toolbox [24]. The first junction tree (J1) had 1024 cliques, and the average clique width was 10. The average degree for each clique was 3. Thus, each clique potential table had 1024 entries, and the size of the separators varied from 2 to 32 entries. The second junction tree (J2) had 512 cliques, and an average clique width of 8. The average degree for the cliques in the second junction tree was also 3. Thus, each clique potential table had 256 entries. The third junction tree (J3) had 1024 cliques. Each clique contained quaternary variables, and the average clique width was 5. Therefore, the potential table also had approximately 1024 entries. In our experiments, we used single precision floating point numbers to represent the probabilities and potentials.

As a comparison with our experimental results, we show the scalability of exact inference using the Intel's Open-Source Probabilistic Networks Library (PNL) [25]. The PNL is a full function, free, open source, graphical model.

The PNL provides an implementation for junction tree inference with discrete parameters. The parallel version of the PNL is now available [25], developed by the Intel Russia Research Center and the University of Nizhni Novgorod. The Intel China Research Center and Intel Architecture Laboratory were also involved in the development process. The scalability of exact inference using PNL is shown in Figure 10. Note that the Cell was not supported by the PNL at the time. The results shown in Figure 10 (a) were obtained on a IBM P655 multiprocessor system, where each processor runs at 1.5 GHz with 2 GB of memory. We can see from Figure 10 (a) that, for all three junction trees, the execution time increased when the number of processors was greater than 4.

In addition to the above three synthetic junction trees, we conducted an experiment using a Bayesian network from a real application. The Bayesian network is called the Quick Medical Reference decision theoretical version (QMR-DT), which is used in a microcomputer-based decision support tool for diagnosis in internal medicine [26]. There were 1000 nodes in such a network. These nodes formed two layers, one representing diseases and the other symptoms. Each disease has one or more edges pointing to the corresponding symptoms. All the random variables (nodes) were binary. We converted the Bayesian network to a junction tree offline and then performed exact inference in the resulting junction tree. The resulting junction tree consists of 114 cliques, while the average clique width is 10. In Figure 10 (b), we illustrate the experimental results using the PNL and our proposed method respectively. The PNL based method cost more execution time and did not show scalability using 8 processors.

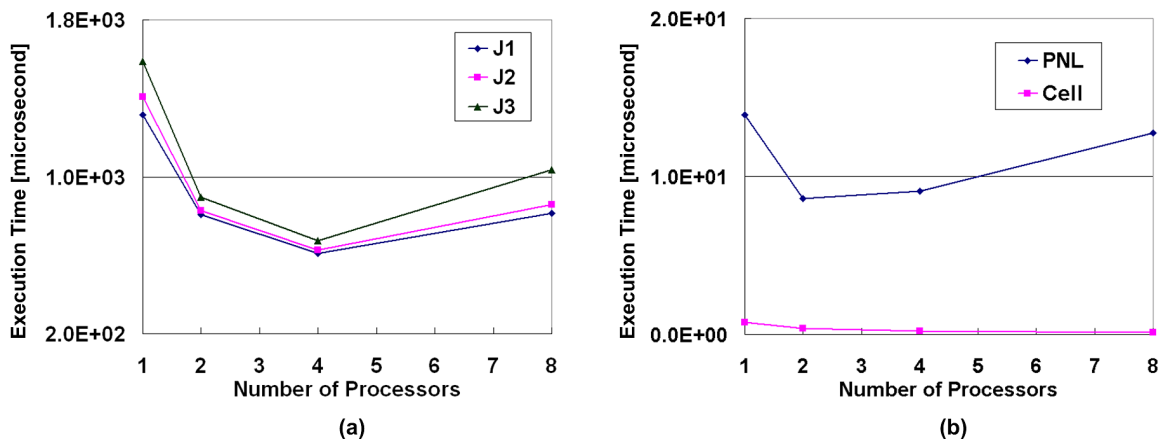


Figure 10: (a) Scalability of exact inference using PNL for various junction trees. (b) Scalability of exact inference using PNL for QMR-DT application.

For the experiments on the Cell, we stored the data of a junction tree as two parts in the main memory. The first part was an adjacency list representing the structure of the junction. The second part was an array of structures (AoS), where each element was a structured variable consisting of a clique potential table, child separators and auxiliary variables for the clique (see Section 4.5). The data layout in the local store contained two buffers, each

corresponding to an element of the array of structures. In addition, we kept the parent separator potential table for each clique in the local store. For each separator, we also reserved an array of the same size as the separators for computation.

We implemented the algorithms in Section 4.6 using C/C++ Language Extension for the Cell. We used double buffering to overlap the computation and data transfer. We also vectorized the computation kernel for evidence collection and the computation kernel for evidence distribution. In addition, we unrolled loops to reduce the number of branch instructions and improve performance.

To evaluate the performance of the junction tree rerooting method shown in Algorithm 3, we constructed a junction tree model called TOY shown in Figure 11. The model is a tree initially rooted at R . There were b branches in the tree. The i -th branch was called Branch i . However, for the sake of simplifying the description, the path $P(R, R')$ was called Branch 0. Letting $b = 1, 2, 4, 8$, we obtained the structures of four junction trees. We let each junction tree have 512 cliques consisting of 8 binary variables. Thus, the serial complexity of each Branch was approximately equal. According to Algorithm 3, clique R' became the new root after rerooting. For each junction tree, we performed evidence propagation on both the original and rerooted versions, using various numbers of cores. We disabled task partitioning, which provided parallelism at a fine grained level.

The results are shown in Figure 12. The speedup in Figure 12 was defined as $Sp = t_R/t_{R'}$, where t_R is the execution time for evidence propagation in the original junction tree and $t_{R'}$ is the execution time for the rerooted tree. According to Section 4.2, we know that when clique R is the root, Branch 0 concatenating Branch 1 is a critical path. When R' is the root, Branch 0 itself is a critical path. Thus, the maximum speedup is $Sp = 2$ for the four junction trees, if the number of concurrent threads P was larger than the number of branches b (see Figure 11). When $P < b$, Sp was less than 2, since some cliques without precedence constraint can not be processed in parallel. From the results in Figure 12, we can see that the rerooted tree led to speedup around 1.9, when 8 cores were used. In addition, the maximum speedup was achieved using more threads as b increased. These observations matched the analysis above. Notice that some speedup curves fell slightly when 8 concurrent threads were used. This was caused by the overheads such as the lock contention.

In Figure 13, we compared the critical paths of the junction trees before and after rerooting. We used the following junction trees in this experiment: J1, J2, J3, the junction tree for the QMR-DT, and the junction tree generated by the TOY model in Figure 11. For the junction tree generated by the TOY model, we let $b = 8$. We calculated the path complexity of the critical path for each junction tree before and after the rerooting algorithm was applied. For the sake of comparison, we normalized the complexity of the original critical path to be 1 for each junction tree. As we can see from the results shown in Figure 13, all the junction trees benefit from rerooting, since

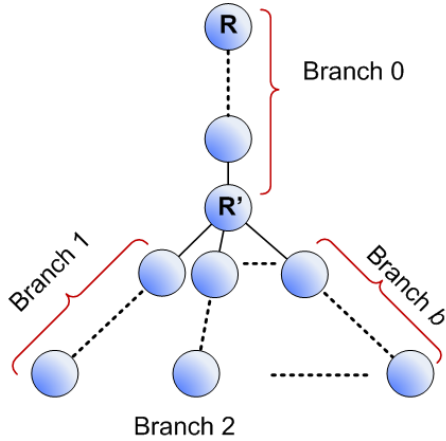


Figure 11: Junction tree for evaluating rerooting algorithm.

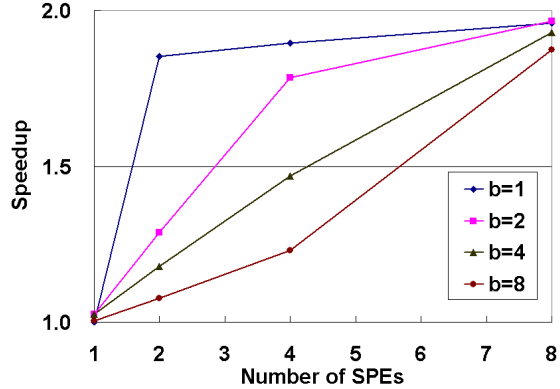


Figure 12: Speedup due to rerooting the junction tree in Figure 11.

the path complexity of the critical path was reduced.

In Figure 14, we measured the execution time for exact inference on the Cell BE processor. We used junction trees J1, J2 and J3 defined above. We show the parameters of these junction trees in Figure 14. In order to show the scalability of the proposed method with respect to various junction trees, we normalized the execution time. Using the results in Figure 14, we calculated the speedup of exact inference for all the input junction trees. The real and ideal speedups are given in Figure 15. The results of speedups for various junction trees were very similar. The ideal speedup given in Figure 15 increased linearly with respect to the number of SPEs, which is the theoretical upper bound of the real speedup.

For the sake of illustrating the load balancing of the proposed algorithm, we measured the workload of each SPE (Figure 16). Unlike Figure 14, where the bars represent normalized execution time, each bar in Figure 16 indicates the the normalized workload. Figure 16 (a) shows the result without load balancing (see the last paragraph in Section 4.3.3). Figure 16 (b) shows without load balancing. In each subfigure, for the sake of comparison, we normalized the heaviest workload to be 1. In our implementation, the overhead of the scheduling was much less than that for potential table computation (less than 1%). The scheduling algorithm was executed in PPE, while the potential table computation was performed in one or more SPEs. Thus, the time taken for scheduling was partially overlapped with that for potential table computation. Since the overhead of scheduler was small, we could focus on the parallelization of the potential table computation.

The experimental results shown in this section illustrate the superior performance of the proposed algorithm and implementation on the Cell. Compared to the results shown in Figure 10, our experimental results exhibited linear speedup and a larger scalability range. From Table 1 and Figure 14, we observed that, even though we used junction

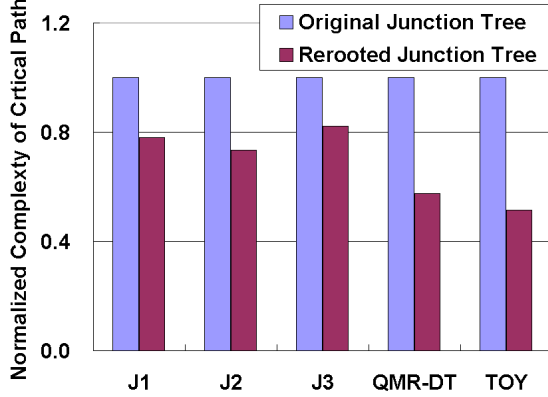


Figure 13: The normalized computational complexity of the critical path for various junction trees.

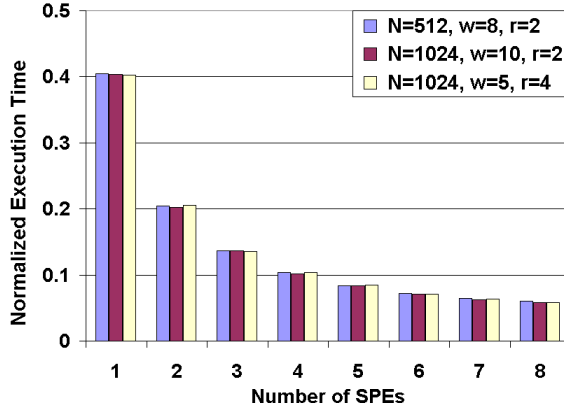


Figure 14: Execution time of exact inference on IBM QS20 for various junction trees using various numbers of SPEs.

trees with various parameters, the experimental results exhibited very stable speedup for all kinds of input. We can see from Figure 15 that, for input junction trees with various parameters, the speedups achieved in our experiments were nearly equal, suggesting that the speedup obtained was not sensitive to the parameters of the input junction trees. The proposed method exhibited similar performance in terms of speedup for all kinds of inputs. In addition, all the speedup curves in Figure 15 were close to the ideal speedup, which is the theoretical upper bound of the real speedup. When the number of SPEs used in experiments was less than 5, the real speedup was almost the same as the ideal speedup. When 6 or more SPEs were used, the speedup was still very close to the upper bound. The stability in experimental results shows that the proposed method is useful for exact inference for arbitrary junction trees.

The experimental results in Figure 16 show that the dynamic scheduler proposed in Section 4.3 evenly distributed workload across all the SPEs, regardless of number of cliques in the junction tree or the size of the potential tables of the cliques. The scheduler dynamically exploits parallelism at multiple granularity levels, leading to load balancing

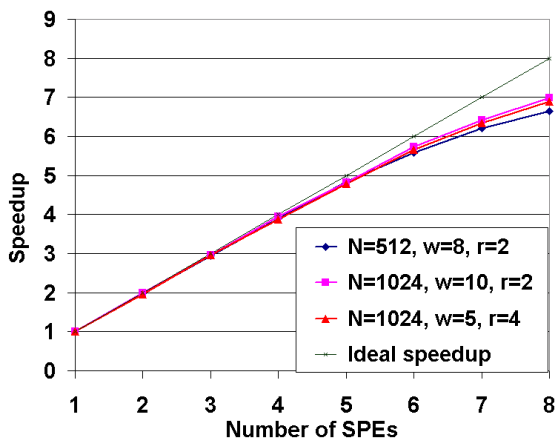


Figure 15: Observed speedup of exact inference on IBM QS20 for various junction trees.

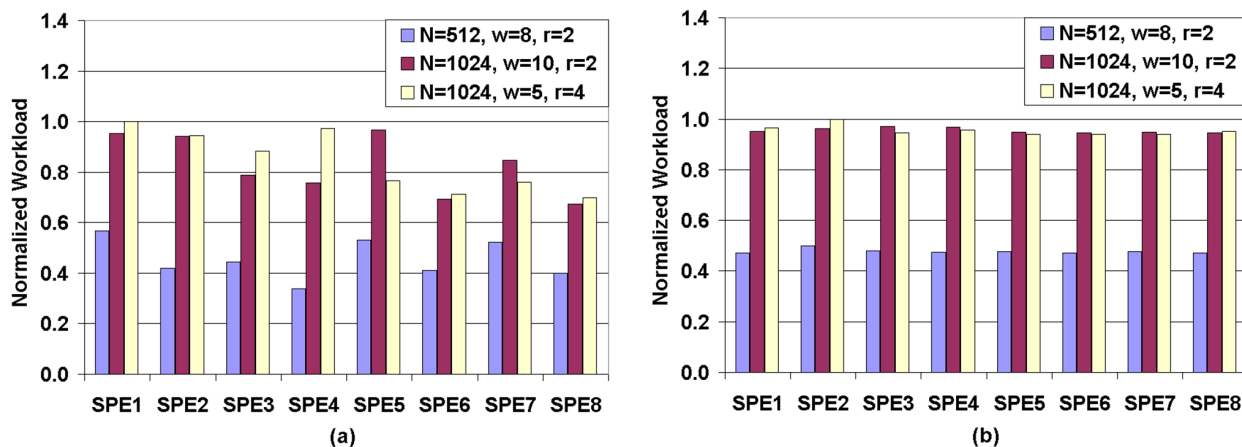


Figure 16: The workload of each SPE for various junction trees (a) without load balancing and (b) with load balancing.

in our experiments. Thus, the SPE resources were sufficiently used. In Figure 16 (b), the workloads were nearly equal for various the input junction trees. The bar length for a junction tree with 512 cliques was much shorter than that for the other two junction trees, because the workload for the junction tree with 512 cliques was much lighter. In Figure 16 (a), the scheduler randomly distributed the workload across all the available SPEs without considering the current workload for the SPEs. As a result, the workload was not evenly distributed. Although the scheduler needed to keep track of several data, the proposed scheduler was efficient and the scheduling overhead was very small. The time taken for scheduling (e.g. checking task size and partitioning large tasks) took a tiny fraction of the entire execution time, which was no more than 1%. Notice that the scheduling algorithm was performed in PPE, while the potential table computation was executed in SPEs. The potential table computation, which took a large portion of the overall execution time, was parallelized. The time taken for scheduling, although not parallelized, was partially

(N, w, r)	1 SPE	2 SPEs	4 SPEs	6 SPEs	8 SPEs
(512, 8, 2)	1	1.994	3.937	5.742	6.670
(1024, 10, 2)	1	1.992	3.925	5.733	6.984
(1024, 5, 4)	1	1.987	3.912	5.644	6.895

Table 1: Speedup with respect to various number of SPEs

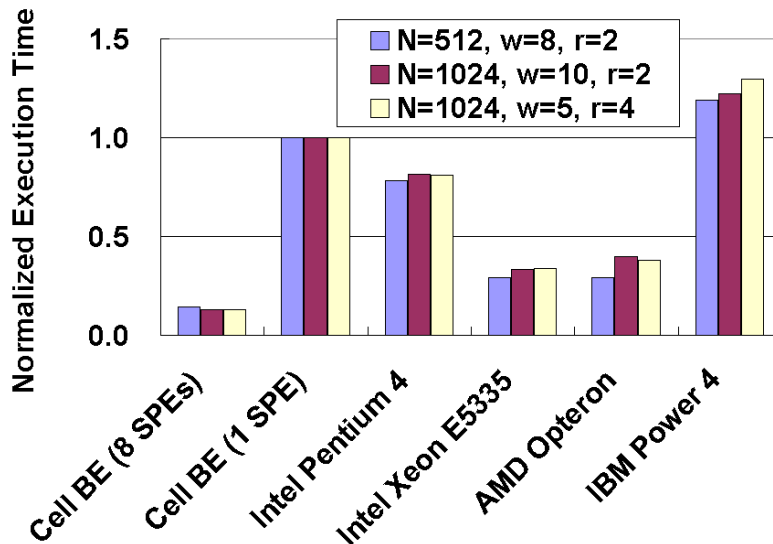


Figure 17: Execution time of the Cell versus that of other processors.

overlapped with the potential table computation.

For the sake of illustrating the capability of the Cell for exact inference, we implemented a serial code for exact inference using the same potential table organization and data layout mentioned in Section 4.5. We compiled the code using gcc with level 3 optimization as well, and executed the code on various platforms, including the Intel Pentium 4 (3.0 GHz, L1 16 KB L2 2 MB) and IBM Power 4 (1.5 GHz, L1 128 KB + 64 KB, L2 1.4 MB, L3 32 MB). We also parallelized the potential table computation of the serial code using Pthreads and executed the code on the AMD Opteron 270 (2.0 GHz, L1 64 KB, L2 1 MB) and Intel Xeon (2.0 GHz, L1 128 KB, L2 8 MB). The results are shown in Figure 17.

Using the 8 SPEs in the Cell, the exact inference on the Cell achieved speedups of 3.0, 6.2, 2.7 and 9.9 over Opteron, Pentium 4, Xeon and Power 4, respectively. For the sake of comparison, we also show the normalized execution time of the proposed method using only one SPE. In Figure 17, the execution time of exact inference on the Cell with 8 SPEs was the lowest. The deviations in execution time among these processors were caused by several factors, such as the clock rate, cache configuration and processor architecture. For example, the execution time on the Intel Pentium 4 was higher than that on the Xeon, because the Pentium processor had only a single core. The IBM Power 4 processor was equipped with 3 level caches, and the size of L3 cache was 32 MB, which is much larger

than that of other processors, but it had a limited clock frequency. Such variety leads to the deviations in execution time. Note that both the Intel Xeon and the AMD Opteron systems contained two quadcore processors with a shared memory. We used Pthreads to distribute the computation to all the cores. However, the parallelized version on the Intel or AMD processors still took more time than the Cell to perform the exact inference. In conclusion, compared with these state-of-the-art processors, the Cell exhibited superior performance for parallel exact inference.

6 Conclusion

In this paper, we presented the design and implementation of a parallel exact inference algorithm on the Cell BE processor. We studied junction tree rerooting to minimize the critical path. We proposed an efficient scheduler to check the complexity of tasks at runtime, partition the large tasks and allocate the SPE resources. We explored optimized potential table representation and data layout for data transfer. We also implemented efficient primitives for evidence propagation. The scheduler proposed in this paper can be utilized for the online scheduling of DAG structured computations. The optimized data organization and efficient primitives can be used in studies involving probabilistic computation, such as particle filtering [27] and MCMC simulation [28]. As part of our future work, we intend to investigate the optimization of the scheduling scheme and analyze the proposed parallel algorithm. We will study the heuristic algorithms for the issuer in the scheduler and explore the relationship between rerooting and the critical path of junction trees. We also plan to work on the junction tree decomposition to exploit efficiently the parallelism in exact inference at multiple levels.

References

- [1] S. L. Lauritzen, D. J. Spiegelhalter, Local computation with probabilities and graphical structures and their application to expert systems, *J. Royal Statistical Society B* 50 (1988) 157–224.
- [2] D. Heckerman, Bayesian networks for data mining, in: *In Data Mining and Knowledge Discovery*, 1997.
- [3] S. J. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach* (2nd Edition), Prentice Hall, 2002.
- [4] E. Segal, B. Taskar, A. Gasch, N. Friedman, D. Koller, Rich probabilistic models for gene expression, in: *9th International Conference on Intelligent Systems for Molecular Biology*, 2001, pp. 243–252.
- [5] D. Pennock, Logarithmic time parallel Bayesian inference, in: *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*, 1998, pp. 431–438.

- [6] A. V. Kozlov, J. P. Singh, A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference, in: Supercomputing, 1994, pp. 320–329.
- [7] R. D. Shachter, S. K. Andersen, P. Szolovits, Global conditioning for probabilistic inference in belief networks, in: Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence, 1994, pp. 514–522.
- [8] Y. Xia, V. K. Prasanna, Node level primitives for parallel exact inference, in: Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, 2007, pp. 221–228.
- [9] Y. Xia, V. K. Prasanna, Parallel exact inference, in: Parallel Computing: Architectures, Algorithms and Applications, Vol. 38, 2007, pp. 185–192.
- [10] D. Bader, V. Agarwal, FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine, in: The 14th Annual IEEE International Conference on High Performance Computing, 2007, pp. 172–184.
- [11] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, The potential of the Cell processor for scientific computing, in: Proceedings of the 3rd ACM Conference on Computing Frontiers, 2006, pp. 9–20.
- [12] IBM Cell BE programming tutorial, <http://www.ibm.com/developer/power/cell>.
URL <http://www.ibm.com/developer/power/cell>
- [13] D. Bader, V. Agarwal, K. Madduri, S. Kang, High performance combinatorial algorithm design on the Cell Broadband Engine processor, *Parallel Computing* 33 (2007) 720–740.
- [14] F. Petrini, O. Villa, D. Scarpazza, Efficient breadth-first search on the Cell BE processor, *IEEE Transactions on Parallel and Distributed Systems* (10) (2007) 1381 – 1395.
- [15] Y. Xia, V. K. Prasanna, Junction tree decomposition for parallel exact inference, in: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, 2008.
- [16] Y. Xia, V. K. Prasanna, Parallel exact inference on the cell broadband engine processor, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008, pp. 1–12.
- [17] H. Shen, Fast parallel algorithm for finding the kth longest path in a tree, in: Advances in Parallel and Distributed Computing Conference, 1997, pp. 164–172.
- [18] G. Buehrer, S. Parthasarathy, The potential of the Cell Broadband Engine for data mining, Tech. Rep. TR-2007-22, Department of Computer Science and Engineering, Ohio State University (2007).

- [19] S. Vinjamuri, V. K. Prasanna, Transitive closure on the Cell BE processor, Tech. rep., University of Southern California (2008).
- [20] J. JáJá, An Introduction to Parallel Algorithms, USA: Addison-Wesley, Reading, MA, 1992.
- [21] P. Bellens, J. Perez, R. Badia, J. Labarta, CellSs: a programming model for the cell be architecture, in: Proceedings of the ACM/IEEE Supercomputing 2006 Conference, 2006, pp. 5–5.
- [22] M. Iverson, F. Ozguner, Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment, in: HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop, 1998, p. 70.
- [23] IBM BladeCenter QS20, <http://www.ibm.com/technology/splash/qs20/>.
URL <http://www.ibm.com/technology/splash/qs20/>
- [24] K. Murphy, Bayes net toolbox, <http://www.cs.ubc.ca/~murphyk/software/bnt/bnt.html>.
URL <http://www.cs.ubc.ca/~murphyk/Software/BNT/bnt.html>
- [25] Intel Open Source Probabilistic Networks Library, <http://www.intel.com/technology/computing/pnl/>.
URL <http://www.intel.com/technology/computing/pnl/>
- [26] B. Middleton, M. S. M. S, D. Heckerman, H. L. M. D, G. Cooper, Probabilistic diagnosis using a reformulation of the INTERNIST-1/QMR knowledge base, *Medicine* 30 (1991) 241–255.
- [27] V. Teulire, O. Brun, Parallelisation of the particle filtering technique and application to doppler-bearing tracking of maneuvering sources, *Parallel Computing* 29 (8) (2003) 1069–1090.
- [28] J. Corander, M. Gyllenberg, T. Koski, Bayesian model learning based on a parallel MCMC strategy, *Statistics and Computing* 16 (4) (2006) 355–362. doi:<http://dx.doi.org/10.1007/s11222-006-9391-y>.
- [29] J. Jingxi, B. Veeravalli, D. Ghose, Adaptive load distribution strategies for divisible load processing on resource unaware multilevel tree networks, *IEEE Transactions on Computers*.
- [30] B.-L. Cheung, C.-L. Wang, A segment-based dsm supporting large shared object space, in: the 20th International Parallel and Distributed Processing Symposium (IPDPS), 2006.
- [31] G. Han, Y. Yang, Scheduling and performance analysis of multicast interconnects, *Journal of Supercomputer* 40 (2) (2007) 109–125.
- [32] D. Bader, Petascale computing for large-scale graph problems, in: the 21th International Parallel and Distributed Processing Symposium (IPDPS), 2007.

- [33] I. Patel, J. R. Gilbert, An empirical study of the performance and productivity of two parallel programming models, in: the 22th International Parallel and Distributed Processing Symposium (IPDPS), 2008.
- [34] M. Tan, H. J. Siegel, J. K. Antonio, Y. A. Li, Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system, *IEEE Transactions on Parallel and Distributed Systems* 8 (8) (1997) 857–871.
- [35] I. Troxel, A. George, Scheduling tradeoffs for heterogeneous computing on an advanced space processing platform, 12th International Conference on Parallel and Distributed Systems.
- [36] E. Grobelny, D. Bueno, I. Troxel, A. George, J. Vetter, Fase: A framework for scalable performance prediction of hpc systems and applications, *Simulation* 83 (10) (2007) 721–745.
- [37] R. Noronha, D. K. Panda, Improving scalability of OpenMP applications on multi-core systems using large page support, in: the 21th International Parallel and Distributed Processing Symposium (IPDPS), 2007, pp. 1–8.
- [38] J.-U. In, P. Avery, R. Cavanaugh, S. Ranka, Policy based scheduling for simple quality of service in grid computing, in: the 18th International Parallel and Distributed Processing Symposium (IPDPS), 2004, pp. 1–8.
- [39] D. Scarpazza, O. Villa, F. Petrini, High-speed string searching against large dictionaries on the Cell BE processor, in: the 22th International Parallel and Distributed Processing Symposium (IPDPS), 2008, pp. 1–8.
- [40] A. Wirawan, K. C. Keong, B. Schmidt, Parallel DNA sequence alignment on the Cell Broadband Engine.
- [41] Y. Pang, L. Sun, S. Guo, S. Yang, Spatial and temporal data parallelization of multi-view video encoding algorithm, in: *IEEE 9th Workshop on Multimedia Signal Processing*, 2007, pp. 441–444.
- [42] A. Sarje, S. Aluru, Parallel biological sequence alignments on the Cell Broadband Engine, in: the 22th International Parallel and Distributed Processing Symposium (IPDPS), 2008, pp. 1–12.
- [43] Y. Chen, X.-H. Sun, M. Wu, Algorithm-system scalability of heterogeneous computing, *Journal of Parallel Distributed Computation* 68 (11) (2008) 1403–1412.
- [44] J. Gonzalez, Y. Low, C. Guestrin, Residual splash for optimally parallelizing belief propagation, in: *In Artificial Intelligence and Statistics (AISTATS)*, 2009, pp. 1–8.