Parallel Exact Inference on the Cell Broadband Engine Processor

Yinglong Xia
Computer Science Department
University of Southern California
Los Angeles, CA 90089
Email: yinglonx@usc.edu

Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: prasanna@usc.edu

***Abstract*—We present the design and implementation of a parallel exact inference algorithm on the Cell Broadband Engine (Cell BE). Exact inference is a key problem in exploring probabilistic graphical models. In such a model, the computation complexity increases dramatically with the network structure and clique size. In this paper, we exploit parallelism at multiple levels. We present an efficient scheduler to dynamically partition large tasks and allocate synergistic processing elements (SPEs). We explore potential table representation and data layout to optimize DMA transfer between the local store and main memory. We also optimized the computation kernels. We achieved linear speedup and superior performance, compared with state-of-the-art processors such as the AMD Opteron, Intel Xeon and Pentium 4. The methodology proposed in this paper can be used for online scheduling of directed acyclic graph (DAG) structured computations.**

I. 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 size 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 on 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 Speigelhalter [1], which converts a Bayesian network into a *junction tree*, then performs exact inference on the junction tree. The complexity of the exact inference algorithms increases dramatically with the density of the network, the width of the cliques and the number of states of the random variables in the cliques. 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 and implementations of exact inference have been presented, such as Pennock [5], Kozlov and Singh [6]. In [7], [8], authors proposed parallel exact inference algorithms using message passing. 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 (SPE). 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. Some recent research provides insight to parallel algorithms design for the Cell [9], [10]. 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 present an efficient *scheduler* to maintain and schedule 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 to optimize data transfer between local stores and the main memory. We implement efficient *node level primitives* and *computation kernels*. The proposed scheduler and data representation can be ported to other parallel computing systems for the online scheduling of directed acyclic graph (DAG) structured computations.

The paper is organized as follows: In Section II, we discuss the background of Bayesian networks and junction trees. Section III discusses related work on parallel exact inference. In Section IV, we presents the exact inference algorithm for the Cell BE processor. Experimental results are shown in Section V. Section VI concludes the paper.

II. Background

A. Exact inference in Bayesian Networks and Junction Trees

A *Bayesian network* is a probabilistic graphic model that exploits conditional independence to represent compactly a joint distribution. Figure 1 (a) shows a sample Bayesian network. In Figure 1 (a), each node represents a random variables. The edges indicate the probabilistic dependence relationships 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 $\mathrm{B} = (\mathbb{G}, \mathbb{P})$, where $\mathbb{G}$ is a *directed acyclic graph* (DAG) and $\mathbb{P}$ is the parameter of the network. The graph $\mathbb{G}$ is denoted $\mathbb{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{A_1, A_2, \ldots, A_n\}$ 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. $\mathrm{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$. 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 are probabilistic dependent upon some 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. To inquiry the updated distribution of variables is 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 (b). All undirected cycles in Figure 1 (a) are eliminated in Figure 1 (a). Each vertex in Figure 1 (b) contains multiple random variables. The numbers in each vertex indicate of which random variables in the Bayesian network the vertex consists. Notice that adjacent vertices always share one or more random variables. All junction trees satisfy the *running intersection property* (RIP) [1]. The RIP property 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 property ensures the evidence observed at any random variables 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 $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$, 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.

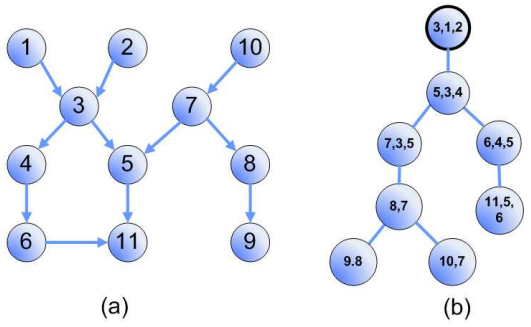

Fig. 1. (a) A sample Bayesian network and (b) corresponding junction tree.

In a junction tree, exact inference proceeds as follows: Assuming evidence is $\mathrm{E} = \{A_i = a\}$ and $A_i \in \mathcal{C}_\mathcal{Y}$, E is *absorbed* at $\mathcal{C}_\mathcal{Y}$ by instantiating the variable A_i and renormalizing the remaining constituents of the clique. The evidence is then propagated from $\mathcal{C}_\mathcal{Y}$ to all other cliques. Mathematically, the evidence propagation is represented as [1]:

$$\psi^*_\mathcal{S} = \sum_{\mathcal{Y} \setminus \mathcal{S}} \psi^*_\mathcal{Y}, \quad \psi^*_\mathcal{X} = \psi_\mathcal{X} \frac{\psi^*_\mathcal{S}}{\psi_\mathcal{S}} \tag{1}$$

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 $\psi_{\mathcal{C}_y}$.

B. 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. 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 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 [11], [12], [13].

III. Related Work on Parallel Exact Inference

There are several works on parallel exact inference, such as Pennock [5], Kozlov and Singh [6] and Szolovits [14]. However, some of those methods, such as [6], are dependent upon the structure of the Bayesian network. The performance of the method also depends upon the structure of the network. Others, such as [5], exhibit limited performance for multiple evidence inputs, since the evidence is assumed to be in the root of the junction tree. In [8], the authors discuss the structure conversion of Bayesian networks. In [7] the node level primitives are parallelized using message passing. All the above algorithms assume a homogeneous machine model. Several recent studies on the Cell provide insight into parallel computing on heterogeneous multicore processors. For example, Bader studied FFT, list ranking etc. on the Cell [9], [12]. Buehrer discussed scientific computing using the Cell [15]. Petrini et al. have developed parallel breadth-first search (BFS) algorithm on the Cell [13]. 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.

IV. Exact Inference for the Cell BE processor

A. Sequential Exact Inference

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

Algorithm 1 Sequential Exact Inference

Input: Junction tree J $= (\mathbb{T}, \hat{\mathbb{P}})$, evidence E and query variables Q, BFS order of cliques $\alpha = (\alpha_1, \cdots, \alpha_N)$.
Output: Probability distribution of Q

1: Absorb evidence: $\psi_{\mathcal{C}_i} = \psi_{\mathcal{C}_i}\delta(E = e)$, $\forall \mathcal{C}_i$
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 $\mathcal{C}_{\alpha_i}$ using $\psi_{\mathcal{C}_{\alpha_i}} = \psi_{\mathcal{C}_{\alpha_i}}\psi^*_{\mathcal{S}}/\psi_{\mathcal{S}}$ where $\psi_{\mathcal{S}} = \sum_{\mathcal{C}_j \setminus \mathcal{C}_{\alpha_i}} \psi_{\mathcal{C}_{\alpha_i}}$
6: **end for**
7: **end for**
8: **for** $i = 2$ to N **do**
9: Compute separator potential table $\psi^*_{\mathcal{S}} = \sum_{\mathcal{C}_{\alpha_i} \setminus \mathcal{C}_{pa(\alpha_i)}} \psi_{\mathcal{C}_{pa(\alpha_i)}}$ 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 $\psi_{\mathcal{S}} = \sum_{\mathcal{C}_{\alpha_i} \setminus \mathcal{C}_{pa(\alpha_i)}} \psi_{\mathcal{C}_{\alpha_i}}$
11: **end for**
12: Compute query Q: $p(Q) = \frac{1}{Z}\sum_{\mathcal{C}_i \setminus Q} \psi_{\mathcal{C}_i}$ where $Q \cap \mathcal{C}_i \neq \emptyset$

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

B. Dynamic Scheduling of Cliques

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 3 (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

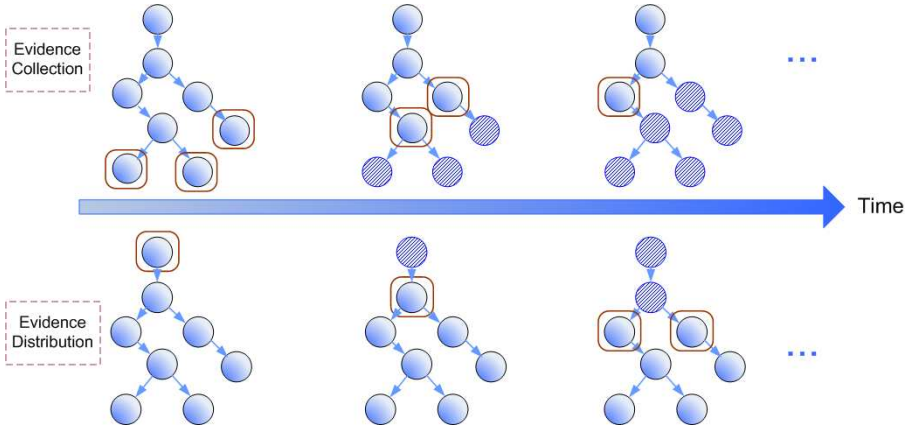

Fig. 2. Illustration of evidence collection and evidence distribution in a junction tree. The cliques in boxes are under processing. The shaded cliques have been processed.

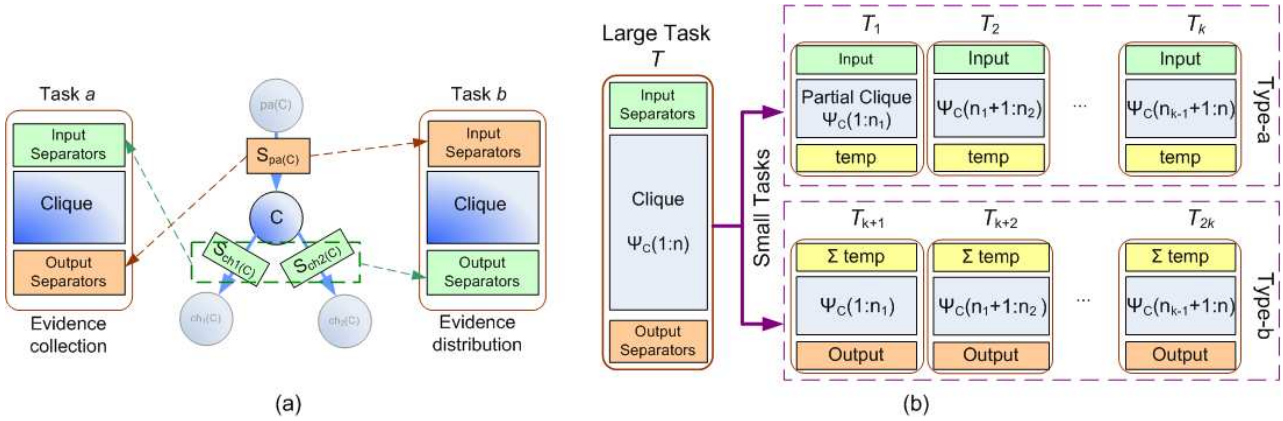

Fig. 3. (a) Illustration of the relationship between a clique $\mathcal{C}$ and the two tasks related to $\mathcal{C}$; (b) Partitioning of a large task to a series of small tasks.

separators are switched, as shown in Figure 3.

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 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 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 IV-C. 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 $\psi_{\mathcal{C}}$, where n is size of $\psi_{\mathcal{C}}$. Denote the $2k$ small tasks $T_1, T_2, \cdots, T_{2k}$. We call $T_1, T_2, \cdots, T_k$ *type-a* tasks of the *regular task* T; $T_{k+1}, T_{k+2}, \cdots, T_{2k}$ are called *type-b* tasks of T (see Figure 3 (b)). The input separators of *type-a* tasks are identical to those of the regular task T. The output separators of *type-a* tasks are called *temporary separators*. The temporary separators of all *type-a* tasks are *accumulated*, and the result is used as the input separators for each *type-b* task. Thus, no *type-b* task can be scheduled for execution until all *type-a* tasks are processed. Accumulating separators is defined as summing up corresponding entries of all input separators. The output separators of all *type-b* tasks are also *accumulated* as the final output separators (the output separators of the original task T). Figure 3 (b) shows a sample large task T and $2k$ small tasks partitioned from T.

2) Task Dependency Graph: For the sake of unifying clique scheduling in both evidence collection and evidence distribution, we create a *task dependency graph* G according to the given junction tree (see the left side figure in Figure 4 (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 4 (a): The upper subgraph

is the given junction tree with all edges directed 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 in the junction tree (except the root) is related to two nodes in the task dependency graph. Note that we do not duplicate potential tables in constructing task dependency graph, though each clique in the junction tree except the root corresponds to two tasks. We need only to store the location of the potential table in a task as a link.

As some tasks involving large potential tables are partitioned to a series of tasks, the task dependency graph G is modified accordingly. The scheduler is in charge of task partition and task dependency graph modification at runtime. We assume the bold nodes in G (see the left side figure in Figure 4 (a)) involve large potential tables. In task partitioning, the bold nodes are replaced by sets of small nodes shown in the dashed boxes (see the right side figure in Figure 4 (a)). Each node in a dashed box denotes a small task partitioned from the original task. All *type-a* tasks are connected to the parent of the regular task. Each *type-b* task depends on all *type-a* tasks. The children of the regular task depend on all *type-b* tasks.

3) Dynamic Partitioning and Scheduling: Scheduling task dependency graphs has been extensively studied (for example, see [16]). In this paper, we use an intuitive 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 we discussed in Section IV-B2, it is straightforward to construct G by using the structure of the given junction tree.

The components of the scheduler are shown in Figure 4 (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 tasks that are not ready to be processed. The *partitioner* is a crucial component of the proposed scheduler, as it dynamically explores parallelism in a finer granularity. The function of the partitioner is to check and partition tasks. In Figure 4 (b), $P_1, P_2, \cdots, P_P$ 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 4 (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 parents 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 [17], [18]. 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 a task T, it loads relevant data 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 $\tilde{T}$, if necessary, to a set of smaller tasks. If $\tilde{T}$ is partitioned, all *type-a* tasks generated from $\tilde{T}$ are assigned to S_I while *type-b* tasks of T 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 number of tasks in 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 SPEs busy. Idle SPEs adversely affect the performance. In Figure 4 (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.

C. Potential Table Organization and Efficient Primitives

For the sake of enhancing the performance of computations on potential tables, we carefully organize the potential tables. 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 determinate the order later in this section. For a clique $\mathcal{C}$, the 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 $s_i \in \{0, 1, \cdots, r-1\}$ 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, it can also be converted 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 5 (a) shows a sample clique $\mathcal{C}$ with binary variable vector (a, b, c, d, e, f). The potential table is given in Figure 5 (b). Notice that we need only to store potential table $\psi_{\mathcal{C}}$ in memory.

Assigning a proper order to 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 for the separator between $\mathcal{C}$ and its parent, and $V_{\mathcal{C} \setminus S_{pa}(\mathcal{C})}$ consists of the remaining

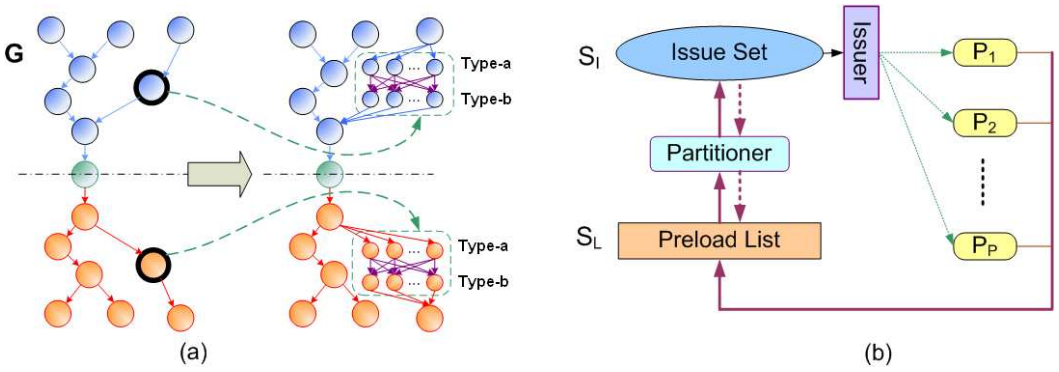

Fig. 4. (a) An example of a task dependency graph for the junction tree given in Figure 1 and the task dependency graph with partitioned tasks. (b) Scheduling scheme for exact inference.

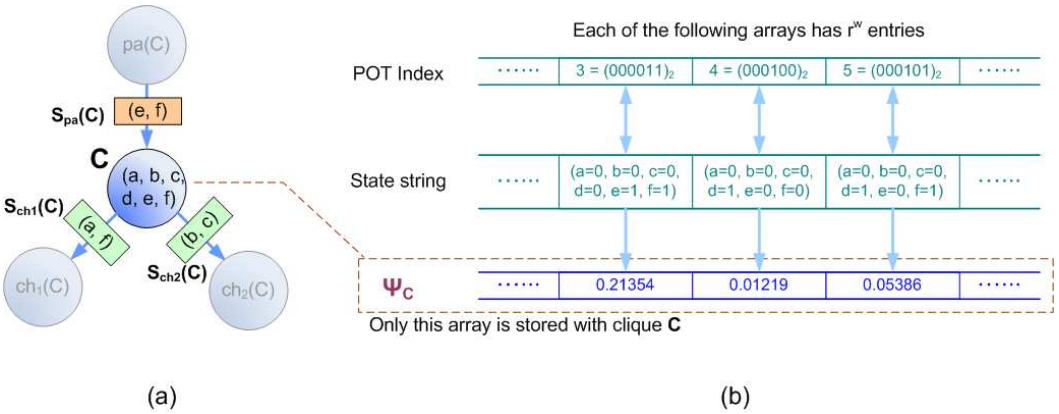

Fig. 5. (a) A sample clique and its variable vector; (b) The relationships among array index, state string and potential table. For the sake of illustration, we assume all random variables are binary variables.

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 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 [7]. In this section, we propose an efficient implementation for the node level primitives by using vectorized algebraic operations. Marginalization is used to obtain separator potential tables from a clique potential table. For example, in Figure 5 (a), we marginalize $\psi_{\mathcal{C}}$ to obtain $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$. Since $V_{S_{pa(\mathcal{C})}}$ is the lower part of $V_{\mathcal{C}}$, the relationship between the entry in $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$ and the entry in $\psi_{\mathcal{C}}$ is straightforward (see Figure 6). Segment i of $\psi_{\mathcal{C}}$ is denoted $\psi_{\mathcal{C}}\left((i-1)|\psi_{pa(\mathcal{C})}| : (i|\psi_{pa(\mathcal{C})}|-1)\right)$, i.e. an array consists of entries from the $(i-1)|\psi_{pa(\mathcal{C})}|^{th}$ to the $(i|\psi_{pa(\mathcal{C})}|-1)^{th}$ of $\psi_{\mathcal{C}}$. Thus, this marginalization can be implemented by accumulating all 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 $\psi_{\mathcal{C}}$ and $\psi^{\dagger}_{\mathcal{S}_{pa(\mathcal{C})}}$, where $\psi^{\dagger}_{\mathcal{S}_{pa(\mathcal{C})}} = \frac{\psi^{*}_{\mathcal{S}_{pa(\mathcal{C})}}}{\psi_{\mathcal{S}_{pa(\mathcal{C})}}}$. Since $\psi^{\dagger}_{\mathcal{S}_{pa(\mathcal{C})}}$ has the same state strings as $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$, the relationship between $\psi^{\dagger}_{\mathcal{S}_{pa(\mathcal{C})}}$ and $\mathcal{S}_{pa(\mathcal{C})}$ is very similar to that in Figure 6. Thus, multiplication can be implemented by multiplying each entry of $\psi^{\dagger}_{\mathcal{S}_{pa(\mathcal{C})}}$ into the corresponding entries in all segments.

However, marginalization to obtain $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$ - the separator potential table for the i^{th} child - requires more computation than that to obtain $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$. The reason is we need to identify the mapping relationship between $V_{\mathcal{S}_{ch_i(\mathcal{C})}}$ and $V_{\mathcal{C}}$. We define the *mapping vector* to represent the mapping relationship from $V_{\mathcal{C}}$ to $V_{\mathcal{S}_{ch_i(\mathcal{C})}}$. The mapping vector is defined as $M_{ch_i(\mathcal{C})} = (m_1 m_2 \cdots m_w | m_j \in \{0, 1, \cdots w_{\mathcal{S}_{ch_i}}\})$, where w is the width of clique $\mathcal{C}$ and $w_{\mathcal{S}_{ch_i}}$ is the length of $V_{ch_i(\mathcal{C})}$. m_j is defined as that the j^{th} variable in $V_{\mathcal{C}}$ mapped to the m_j^{th} variable in $V_{\mathcal{S}_{ch_i(\mathcal{C})}}$. $m_j = 0$ if the j^{th} variable in $V_{\mathcal{C}}$ is not in $V_{\mathcal{S}_{ch_i(\mathcal{C})}}$. Using the mapping vector $M_{ch_i(\mathcal{C})}$, we can identify the relationship between $\psi_{\mathcal{C}}$ and $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$. Given an entry $\psi_{\mathcal{C}}(t)$, 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, $\psi_{\mathcal{C}}(t)$ corresponds to $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}(\tilde{t})$. To compute $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}$ from $\psi_{\mathcal{C}}$, we just need to identify the relationship for each t and accumulate $\psi_{\mathcal{C}}(t)$ to $\psi_{\mathcal{S}_{ch_i(\mathcal{C})}}(\tilde{t})$.

Clique potential table size increases dramatically with 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 IV-B 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 which 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 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 performed.

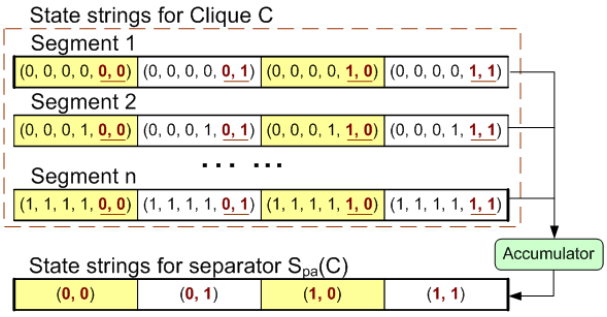

Fig. 6. The relationships between entries of $\psi_{\mathcal{C}}$ and entries of $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$ in Figure 5 (a). Each entry of $\psi_{\mathcal{S}_{pa(\mathcal{C})}}$ is the sum of the corresponding entries in all segments.

D. Data Layout for Cell Optimization

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 should help data transfer between main memory and local stores, and the 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 7 (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 $S_{ch1}(\mathcal{C})$ and $S_{ch2}(\mathcal{C})$ respectively. Then, clique $\mathcal{C}$ is updated using $S_{ch1}(\mathcal{C})$ and $S_{ch2}(\mathcal{C})$. Lastly, $\mathcal{C}$ updates $S_{pa}(\mathcal{C})$. Notice that, according to Eq. (1), updating clique potential table ψ_C needs both ψ_C and the updated separator, while updating separator potential table $\psi_{S_{pa}(\mathcal{C})}$ needs only the updated ψ_C. In evidence distribution, $\psi_{\mathcal{C}}$ is updated using $\psi_{S_{pa}(\mathcal{C})}$ and then updated $\psi_{\mathcal{C}}$ is used to propagate evidence to $S_{ch1}(\mathcal{C})$ and $S_{ch2}(\mathcal{C})$.

We store the junction tree data in the main memory (Figure 7). 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 an SPE. If all 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 data. However, note that a separator is shared by two cliques. If all data needed for processing a clique are stored in continuous locations, separators must be duplicated. In addition, to maintain 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 separators between it and its children, but leave the parent separator to the parent of the clique. Each clique corresponds to a block in Figure 7 (a), while each block consists of a clique potential table, child separators, the properties of the clique and child separators (see Figure 7 (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. The alignment benefits the computation in SPEs [11]. When the PPE schedules a clique $\mathcal{C}$ to an SPE, the PPE sends the starting address of the data block of clique $\mathcal{C}$ and the starting address of the parent separator to the SPE.

E. Complete Algorithm for Parallel Exact Inference

The parallel exact inference algorithm proposed in this paper consists of task dependency graph scheduling (Section IV-B) and potential table updating (Sections IV-C). For the Cell, PPE is in charge of the task scheduling, and SPEs perform the potential table updating.

Using the scheduling definition and notations in Section IV-B, we present the algorithm for task dependency graph scheduling in Algorithm 2. Lines 1 and 2 in Algorithm 2 are initial steps. Since all tasks in S_I are ready to be processed, Lines 4-7 issue the tasks to available SPEs. Lines 8-12 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 V, 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 accomplished tasks 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 17-22). In Line 17, 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 20-21).

Using the data layout in Section IV-D, we present the algorithm for updating potential tables (Algorithm 3). Two

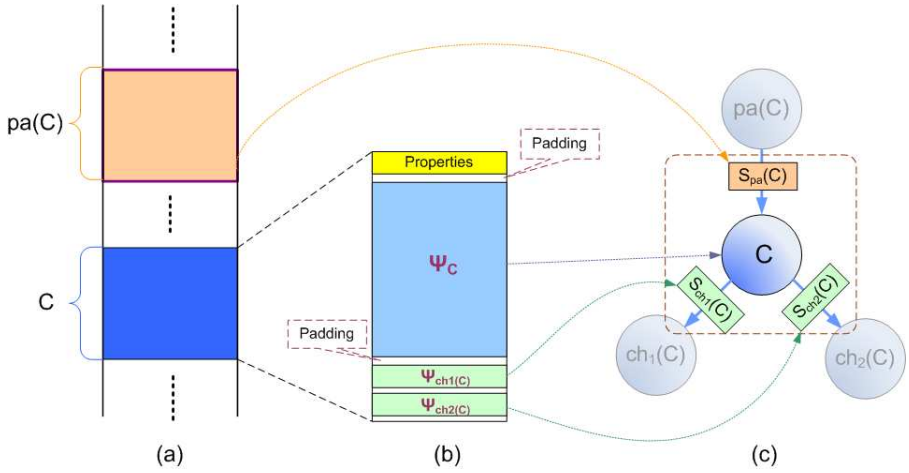

Fig. 7. (a) Data layout for junction tree in main memory; (b) Data layout for each clique; (c) Relationship between the layout and partial junction tree structure.

Algorithm 2 PPE: Task dependency graph scheduling

Input: Junction tree J, size of clique potential tables and separators, thresholds δ_T, δ_S, number of SPEs P
Output: Scheduling sequence for exact inference

Obtain task dependency graph G from J
$S_I = \emptyset; S_L = \{\text{all tasks in } G\}$
while $S_I \cup S_L \neq \emptyset$ **do**
  for i = 1 to $\min(|S_I|, P)$ **do**
    $T =$ get a task in S_I
    assign T to SPE_i and let $S_I = S_I \backslash T$ if SPE_i is idling
  end for
  if $|S_I| < \delta_S$ and S_I contains regular tasks **then**
    $T =$ the largest regular task in S_I
    partition T to small task sets T_{type-a}, T_{type-b}
    $S_I = S_I \cup T_{type-a}$; $S_L = S_L \cup T_{type-b}$
  end if
  for $T \in \{$accomplished tasks in all SPEs$\}$ **do**
    for $\tilde{T} \in \{$children of $T\}$ **do**
      decrease the dependency degree of $\tilde{T}$ by 1
      if dependency degree of $\tilde{T} = 0$ **then**
        if $|\tilde{T}| < \delta_T$ **then**
          $S_L = S_L \backslash \{\tilde{T}\}$; $S_I = S_I \cup \{\tilde{T}\}$
        else
          partition $\tilde{T}$ to small task sets $\tilde{T}_{type-a}, \tilde{T}_{type-b}$
          $S_I = S_I \cup \tilde{T}_{type-a}$; $S_L = S_L \cup \tilde{T}_{type-b}$
        end if
      end if
    end for
  end for
end while

Algorithm 3 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 ψ_{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 ψ_{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 ψ_{out} using computation kernel of evidence collection (Algorithm 4)
16: **else**
17: update $\psi_{\mathcal{C}}$ and ψ_{out} using computation kernel of evidence distribution (Algorithm 5)
18: **end if**
19: **end if**
20: store updated $\psi_{\mathcal{C}}$ and ψ_{out} to main memory
21: notify PPE that task T is done
22: let $T = \tilde{T}$
23: **end while**

computation kernels for potential table updating are shown in Algorithms 4 and 5. Double buffering is used in Algorithm 3 for the sake of overlapping computation and data transfer. While loading relevant data for $\tilde{T}$, we perform computations on task T. Lines 5-7 compute ψ_{temp} for regular tasks and *type-a* tasks. ψ_{temp} is the output for *type-a* tasks, but it is

Algorithm 4 Computation kernel of evidence collection

Input: input separators ψ_{in_i}, (partial) clique potential table $\psi_{\mathcal{C}}$, output separator ψ_{out}, temporary separator ψ_{temp}, index offset t_δ, mapping vector M_{in}
Output: updated $\psi_{\mathcal{C}}$ 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 $\tilde{S} = (\tilde{s}_1 \tilde{s}_2 \cdots \tilde{s}_{|in|})$ from S using mapping vector M_{in_i}
6: Convert $\tilde{S}$ to $\tilde{t}$
7: $\psi_{\mathcal{C}}(\tilde{t}) = \psi_{\mathcal{C}}(\tilde{t}) * \psi_{in_i}(t)$
8: **end for**
9: **end for**
10: $\psi_{out}(1 : |\psi_{out}|) = \vec{0}$
11: **for** $i = 1$ to $|\psi_{\mathcal{C}}|$ step $|\psi_{out}|$ **do**
12: $\psi_{out}(0 : |\psi_{out}|) = \psi_{out}(0 : |\psi_{out}|) + \psi_{\mathcal{C}}(i : i + |\psi_{out}|)$
13: **end for**

Algorithm 5 Computation kernel of evidence distribution

Input: input separator ψ_{in}, (partial) clique potential table $\psi_{\mathcal{C}}$, output separators ψ_{out_i}, temporary separator ψ_{temp}, index offset t_δ, mapping vector M_{out_i}
Output: updated $\psi_{\mathcal{C}}$ 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: $\psi_{\mathcal{C}}(i : i + |\psi_{in}|) = \psi_{\mathcal{C}}(i : i + |\psi_{in}|) * \psi_{in}(1 : |\psi_{in}|)$
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 $|\psi_{\mathcal{C}}| - 1$ **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: $\psi_{out_i}(\tilde{t}) = \psi_{out_i}(\tilde{t}) + \psi_{\mathcal{C}}(t)$
12: **end for**
13: **end for**

an intermediate result for regular tasks (see Lines 8-12 in Algorithm 3). Lines 13-19 process T by using one of two computation kernels, depending on the status of clique $\mathcal{C}$. Line 21 notifies PPE to prepare a new task for this SPE.

Two computation kernels (Algorithms 4 and 5) are used for evidence collection and evidence distribution respectively. The two kernels apply Eq. 1 to the given task. Lines 1-9 in Algorithm 4 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 IV-C). Lines 10-13 marginalize the updated $\psi_{\mathcal{C}}$. Benefiting from the data organization presented in Section IV-C, potential table division (Line 2) and marginalization (Line 12) are simplified to vectorized algebraic operations, which perform efficiently on SPEs and other SIMD machines. Algorithm 5 is similar to Algorithm 4. However, in Algorithm 5, potential table division (Line 1) and multiplication (Line 3) are simplified to vectorized algebraic operations. Mapping vectors are utilized to implement potential table marginalization (Lines 6-12).

V. Experiments

We conducted experiments on a Sony PlayStation 3, where the CPU is a 3.2 GHz Cell BE processor with 512 KB Level 2 cache, 256 MB XDR memory. 6 out of 8 SPEs are available [19]. The PlayStation 3 was installed with Yellow Dog Linux and the Cell BE SDK 3.0 Developer package. We compiled the code using the gcc compiler provided with the SDK, with level 3 optimization.

As a comparison to our experimental results, we show the scalability of exact inference using Intel's Open-Source Probabilistic Networks Library (PNL) [20]. The PNL is a full function, free, open source, graphical model library released under a Berkeley Software Distribution (BSD) style license. The PNL provides an implementation for junction tree inference with discrete parameters. The parallel version of the PNL is now available [20]. The scalability of exact inference using PNL is shown in Figure 8. Note that the Cell was not supported by the PNL at the time, so the results shown in Figure 8 were obtained on standard processors [21]. We can see from Figure 8 that, for all the three junction trees, the execution time increased when the number of processors was greater than 4.

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 [22]. The first junction trees 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 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 had 1024 cliques. Each clique contained quarternary 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.

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 structure (AoS) where each element was a structured variable consisting of a clique potential table, child separators and auxiliary variables for the clique (see Section IV-D). The data layout in the local store contained two buffers, each corresponding to an element of the array of structure. 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 separators for computation.

We implemented the algorithms in Section IV-E 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 the performance.

In Figure 9, we measured the execution time for exact inference on the Cell BE processor. 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 9, we calculated the speedup of exact inference for all the input junction trees. The real speedup and the ideal speedup are given in Figure 10. Since the results of speedup for various junction trees were very similar, the speedup curves were closed to each other. The ideal speedup given in Figure 10 was linearly increased with respect to the number of SPEs. It was the theoretical upperbound of the real speedup.

For the sake of illustrating the load balancing of the proposed algorithm, we measured the workload of each SPE (Figure 11). Different from Figure 9, where each bar represents normalized execution time, each bar in Figure 11 indicates the the normalized workload. The first subfigure in Figure 11 shows the result with load balancing (see the last paragraph in Section IV-B3). The second subfigure shows the result without load balancing. In each subfigure, for the sake of comparison, we normalized the heaviest workload to be 1. In our implementation, as shown in Figure 12, the time taken for scheduling was much less than that for potential table computation. The scheduling algorithm was executed in PPE, while the potential table computation was performed in one or more SPEs. Notice that the time taken for scheduling was partially overlapped with that for potential table computation, since PPE and SPEs worked in parallel. Thus, the overhead of scheduler was small and we could focus on the parallelization of the potential table computation.

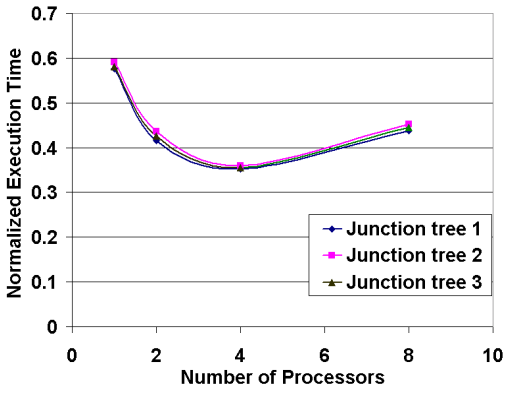

Fig. 8. The scalability of exact inference in various junction trees using PNL library.

For the sake of illustrating the capability of the Cell for exact inference, we implemented a sequential code for exact inference using the same potential table organization and data layout mentioned in Section IV-D. We compiled the code using gcc with level 3 optimization as well, and executed the code

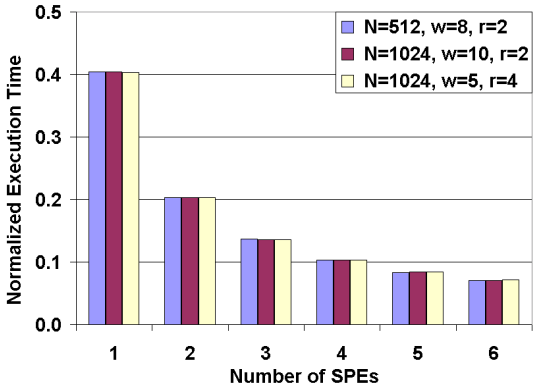

Fig. 9. Execution time of exact inference for various junction trees using various numbers of SPEs.

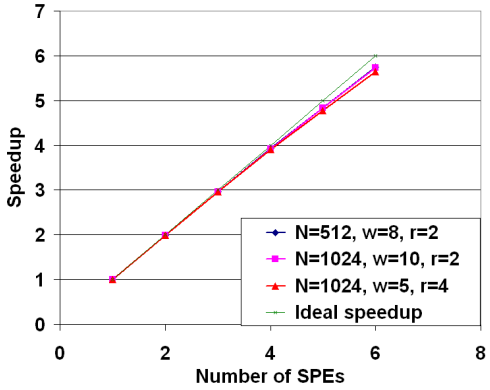

Fig. 10. Observed speedup of exact inference on Cell for various junction trees.

on various platforms, including the AMD Opteron 270 (2.0 GHz, L1 64 KB, L2 1 MB), Intel Pentium 4 (3.0 GHz, L1 16 KB L2 2 MB), Intel Xeon (2.0 GHz, L1 128 KB, L2 8 MB) and IBM Power 4 (1.5 GHz, L1 128 KB + 64 KB, L2 1.4 MB, L3 32 MB). The results are shown in Figure 13.

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 8, our experimental results exhibited linear speedup and a larger scalability range. From Table I and Figure 9, we observe that, even though we used junction trees with various parameters, the experimental results exhibited very stable speedup for all kinds of input. We can see from Figure 10 that, for various number of SPEs used in the experiments,

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

TABLE I
SPEEDUP WITH RESPECT TO VARIOUS NUMBERS OF SPES

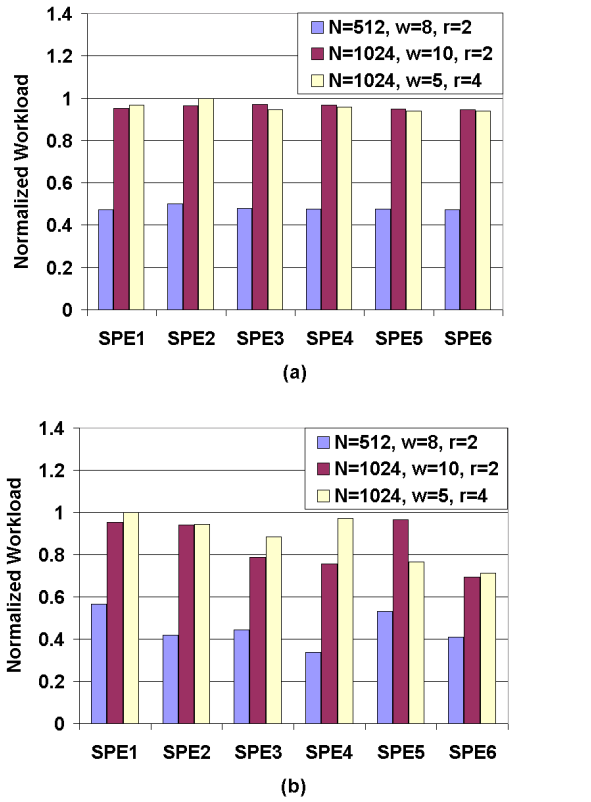

Fig. 11. The workload of each SPE with respect to various junction trees (a) with load balancing. (b) without load balancing.

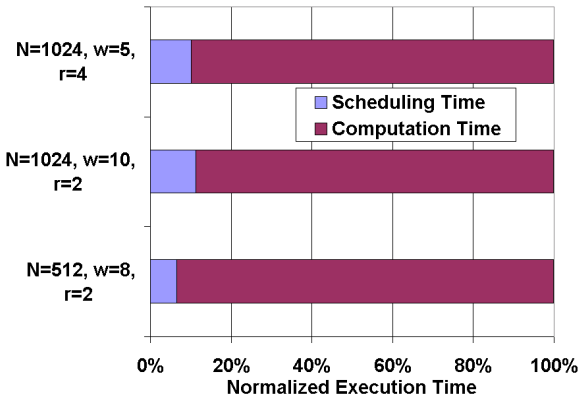

Fig. 12. Comparison of the time for scheduling and the computation time for various types of junction trees.

the speedups achieved in our experimental results were very similar. It suggests that the speedup we 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 10 are quite close to the ideal speedup which is the theoretical upperbound of the real speedup. When the number of SPEs used in experiments was smaller than 5, the real speedup was almost overlapped to the ideal speedup. When 6 SPEs were used, the speedup was still very close to the upperbound. The stability in experimental results shows that the proposed method is useful for exact inference for arbitrary junction trees.

The experimental results in Figure 11 show that the dynamic scheduler proposed in Section IV-B evenly distributed workload among all the SPEs, regardless of number of cliques in the junction tree and the size of the potential tables of the cliques. The scheduler dynamically exploits parallelism at multiple granularity levels, leading to load balancing in our experiments. Thus, the SPE resources were sufficiently used. In the first subfigure of Figure 11, the workloads were very similar for all inputs. The length of bar 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 11 (b), the scheduler randomly distributed the workload among all available SPEs. Therefore, it did not consider the current workload for the SPEs. As a result, the workload was not evenly distributed, compared to that in the first subfigure. Although the scheduler needed to keep track of several data, Figure 12 shows that the proposed scheduler was efficient. The time taken for scheduling, e.g. checking task size and partitioning large tasks, took a small fraction of the entire execution time. Notice that the scheduling algorithm was performance in PPE, while the potential table computation was executed in SPEs. The potential table computation, which took a large portion of the total execution time, was parallelized. The time taken for scheduling, although not be parallelized, was partially overlapped with the potential table computation.

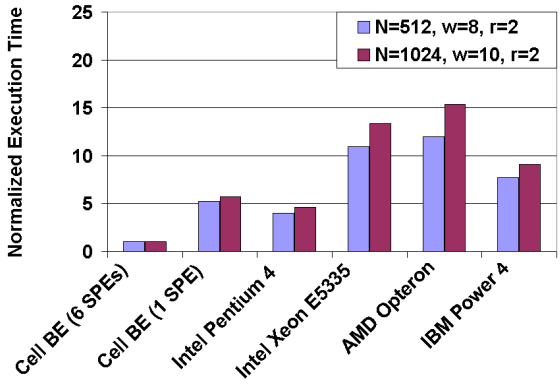

Fig. 13. Execution time of the Cell versus that of other processors.

Note that, although there are 8 SPEs in the Cell, only 6 of them can actually be used by users. Using 6 of the 8 SPEs in the Cell, the exact inference on the Cell achieved speedups of 16, 4, 13 and 6 over Opteron, Pentium 4, Xeon and Power 4, respectively (see Figure 13). For the sake of comparison, we also show the normalized execution time of the proposed method using only one SPE. In Figure 13, the execution time of exact inference on the Cell with 6 SPEs was the smallest. The deviations in execution time among the

standard processors were caused by several reasons such as the clock rate, cache configuration and processor architecture. We addressed the clock rates and cache configurations of these standard processors in the early part of this section. For example, the execution time on Intel Pentinum 4 was shorter than on Xeon because the Pentinum processor we used had a higher clock frequency. Although the IBM Power 4 processor had a limited clock frequency, it was equipped with 3 levels of cache. The size of L3 cache was 32 MB, which is much larger than the cache size. Such variety leads to the deviations in execution time.

VI. Conclusion

In this paper, we presented a design and implementation of a parallel exact inference algorithm on the Cell BE processor. 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 online scheduling of DAG structured computations. The optimized data organization and efficient primitives can also be used in other studies involving probabilistic computation such as particle filtering [23] and MCMC simulation [24]. As part of our future work, we intend to investigate the optimization of the scheduling scheme and theoretical analysis of the proposed parallel algorithm. We will study the optimized 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 efficiently exploit the parallelism in exact inference at multiple levels.

Acknowledgment

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

References

[1] S. L. Lauritzen and D. J. Spiegelhalter, "Local computation with probabilities and graphical structures and their application to expert systems," *J. Royal Statistical Society B*, vol. 50, pp. 157–224, 1988.

[2] D. Heckerman, "Bayesian networks for data mining," in *In Data Mining and Knowledge Discovery*, 1997.

[3] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.

[4] E. Segal, B. Taskar, A. Gasch, N. Friedman, and 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 and J. P. Singh, "A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference," in *Supercomputing*, 1994, pp. 320–329.

[7] Y. Xia and 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.

[8] ——, "Parallel exact inference," in *Parallel Computing: Architectures, Algorithms and Applications*, vol. 38, 2007, pp. 185–192.

[9] D. Bader and 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.

[10] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and 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.

[11] "IBM Cell BE programming tutorial." [Online]. Available: http://www.ibm.com/developer/power/cell

[12] D. Bader, V. Agarwal, K. Madduri, and S. Kang, "High performance combinatorial algorithm design on the Cell Broadband Engine processor," *Parallel Computing*, vol. 33, pp. 720–740, 2007.

[13] F. Petrini, O. Villa, and D. Scarpazza, "Efficient breadth-first search on the Cell BE processor," *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[14] R. D. Shachter, S. K. Andersen, and P. Szolovits, "Global conditioning for probabilistic inference in belief networks," in *Proceedings of the Tenth Conference on Uncertainty in Articial Intelligence*, 1994, pp. 514–522.

[15] G. Buehrer and S. Parthasarathy, "The potential of the Cell Broadband Engine for data mining," Department of Computer Science and Engineering, Ohio State University, Tech. Rep. TR-2007-22, 2007.

[16] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, MA: USA: Addison-Wesley, 1992.

[17] P. Bellens, J. Perez, R. Badia, and J. Labarta, "CellSs: a programming model for the cell be architecture," in *Proceedings of the ACM/IEEE Supercomputing 2006 Conference*, 2006, pp. 5–5.

[18] M. Iverson and 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.

[19] M. Linklater, "Optimizing Cell core," *Game Developer Magazine*, pp. 15–18, 2007.

[20] "Intel Open Source Probabilistic Networks Library." [Online]. Available: http://www.intel.com/technology/computing/pnl/

[21] V. K. Namasivayam and V. K. Prasanna, "Scalable parallel implementation of exact inference in Bayesian networks," in *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, 2006, pp. 143–150.

[22] K. Murphy, "Bayes net toolbox." [Online]. Available: http://www.cs.ubc.ca/∼murphyk/Software/BNT/bnt.html

[23] V. Teulire and O. Brun, "Parallelisation of the particle filtering technique and application to doppler-bearing tracking of maneuvering sources," *Parallel Computing*, vol. 29, no. 8, pp. 1069–1090, 2003.

[24] J. Corander, M. Gyllenberg, and T. Koski, "Bayesian model learning based on a parallel MCMC strategy," *Statistics and Computing*, vol. 16, no. 4, pp. 355–362, 2006.