Noname manuscript No.
(will be inserted by the editor)

Parallel Evidence Propagation on Multicore Processors

Yinglong Xia · Viktor K. Prasanna

Received: date / Accepted: date

Abstract We propose a parallel evidence propagation method on general-purpose multicore processors. Evidence propagation is a major step in exact inference, a key problem in exploring probabilistic graphical models. We explore the parallelism in evidence propagation at various levels. First, given an arbitrary junction tree, we construct a directed acyclic graph (DAG) with weighted nodes, each denoting a computation task for evidence propagation. Since the execution time of the tasks varies significantly, we develop a workload-aware scheduler to allocate the tasks to the cores of the processors. The scheduler monitors the workload of each core and dynamically allocates tasks to support load balance across the cores. In addition, we integrate a module in the scheduler to partition the tasks converted from cliques with large potential tables so as to achieve improved load balance. We implemented the proposed method using Pthreads on both AMD and Intel quadcore processors. For a representative set of junction trees, our method achieved almost linear speedup. The execution time of our method was around twice as fast as an OpenMP based implementation on both the platforms.

Keywords Exact inference · Multicore processor · Junction tree · Scheduling · DAG structured computation

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

Y. Xia
Computer Science Department
University of Southern California
Los Angeles, CA 90089, U.S.A.
E-mail: yinglonx@usc.edu

V.K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, U.S.A.
E-mail: prasanna@usc.edu

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 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 in a Bayesian network can be *exact* or *approximate*. Exact inference is NP hard [5]. The most popular exact inference algorithm for multiply connected networks was proposed by Lauritzen and Speigelhalter [1], which converts a Bayesian network into a *junction tree*, then performs exact inference in the junction tree. The complexity of 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.

In this paper, we study parallelization of evidence propagation on state-of-the-art multicore processors. We construct a task dependency graph with minimum critical path from a given junction tree. Since the execution time for tasks in exact inference can vary significantly, we propose an efficient scheduler to monitor the workload of each thread at runtime. The scheduler partitions tasks involving large potential tables into a set of small tasks so as to achieve improved load balance across the threads. We achieved speedup of 7.4 using 8 cores on state-of-the-art platforms. This speedup is much higher compared with an OpenMP based implementation. The proposed method can be extended for online scheduling of directed acyclic graph (DAG) structured computations.

The paper is organized as follows: In Section 2, we discuss the background of evidence propagation. Section 3 introduces related work. Section 4 defines computation tasks for evidence propagation. Section 5 presents our workload-aware scheduler for multicore processors. Experimental results are shown in Section 6. Section 7 concludes the paper.

2 Background

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to represent compactly a joint distribution. Figure 1 (a) shows a sample Bayesian network, where each node represents a random variable. The edges indicate the probabilistic dependence relationships between two random variables. The structure of a Bayesian network is a *directed acyclic* *graph* (DAG), 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 random variable in the Bayesian network has a *conditional probability table* $P(A_j|pa(A_j))$, where $pa(A_j)$ is the parents of A_j. Given the Bayesian network, a joint distribution is given by $P(\mathcal{V}) = \prod_{j=1}^{n} P(A_j|pa(A_j))$, where $A_j \in \mathcal{V}$ [1].

The *evidence* in a Bayesian network is the variables that have been instantiated, e.g. $\mathrm{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$, where A_{e_i} is a variable and a_{e_i} is the instantiated value. Evidence can be propagated to other variables in the Bayesian network using Bayes' Theorem. Propagating the evidence throughout a Bayesian network is called *inference*, which can be *exact* or *approximate*. Exact inference is proven to be NP hard [5]. The computational complexity of exact inference increases dramatically with the size of the Bayesian 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*. We illustrate a junction tree converted from the Bayesian network (Figure 1 (a)) in Figure 1 (b), where all undirected cycles in are eliminated. Each vertex in Figure 1 (b) contains multiple random variables from the Bayesian network. 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 set 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 having r states, the number of entries in $\mathcal{C}_i$ is r^w.

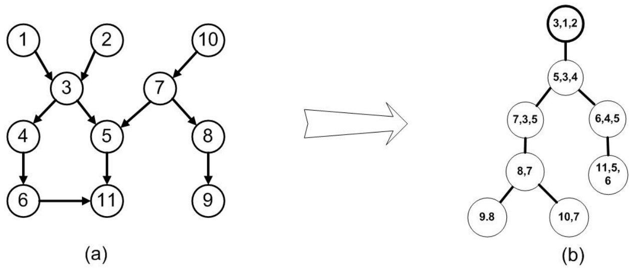

Fig. 1 (a) A sample Bayesian network and (b) corresponding junction tree.

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 variables of the clique. The evidence is then propagated from $\mathcal{C}_\mathcal{Y}$ to any adjacent cliques $\mathcal{C}_\mathcal{X}$. Let $\psi^*_\mathcal{Y}$ denote the potential table of $\mathcal{C}_\mathcal{Y}$ after E is absorbed, and $\psi_\mathcal{X}$ the potential table of $\mathcal{C}_\mathcal{X}$.

Mathematically, evidence propagation is represented as [1]:

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

where $\mathcal{S}$ is a separator between cliques $\mathcal{X}$ and $\mathcal{Y}$; $\psi_{\mathcal{S}}$ ($\psi_{\mathcal{S}}^{*}$) denotes the original (updated) potential table of $\mathcal{S}$; $\psi_{\mathcal{X}}^{*}$ is the updated potential table of $\mathcal{C}_{\mathcal{X}}$.

A two-stage method ensures that evidence at any cliques in a junction tree can be propagated to all the other cliques [1]. The first stage is called *evidence collection*, where evidence is propagated from leaf cliques to the root (bottom up); the second stage is called *evidence distribution*, where the evidence is propagated from the root to leaf cliques (top down). Figure 2 illustrates the first three steps of evidence collection and distribution in a sample junction tree.

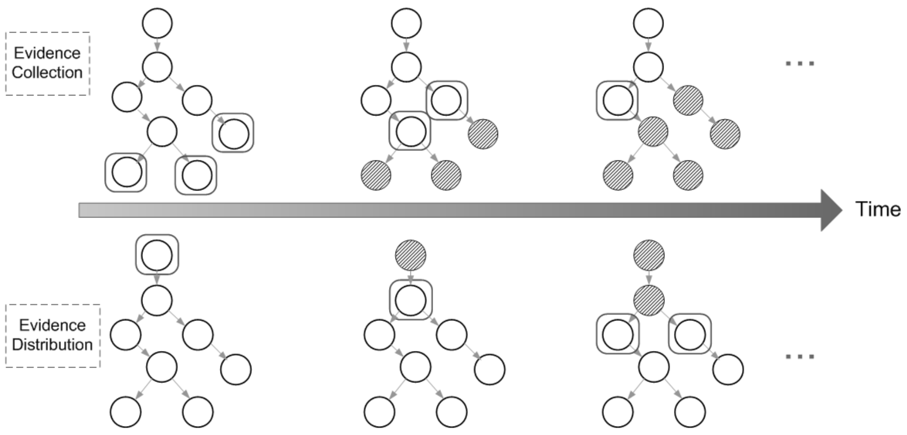

Fig. 2 Illustration of evidence collection and evidence distribution in a junction tree. The cliques in boxes are under processing. The shaded cliques have been processed.

3 Related Work

There are several works on parallel exact inference, such as Pennock [5], Kozlov and Singh [6] and Szolovits. However, some of those methods, such as [6], are dependent upon the structure of the Bayesian network. The performance is adversely affected if the structure of the input Bayesian network is changed. Some other methods, such as [5], exhibit limited performance for multiple evidence inputs. In [7], the node level primitives are parallelized using message passing on distributed memory platforms, which is not directly applicable in this paper, since the multicore platforms have shared memory. A centralized scheduler for exact inference is introduced in [8] on a heterogeneous multicore processor, but the multicore platforms studied in this paper are homogeneous.

Generic scheduling techniques have been proposed by several programming systems such as Cilk [9], Intel TBB [10], OpenMP [11], etc. These systems are not optimized specifically for exact inference. For example, OpenMP does not focus on task level parallelism. Cilk is based on task stealing, where task weights are not considered. In the case of exact inference, task execution time varies significantly from task to task. We deviate from the above approaches and explore workload-aware task scheduling techniques for exact inference.

4 Task Definition and Dependency Graph Construction

Evidence propagation consists of a series of computations called node level primitives. There are four types of node level primitives: *marginalization*, *extension*, *multiplication* and *division* [7]. In this paper, we define a *task* as the computation of a node level primitive. A property of the primitives is that the potential table of a clique can be partitioned into independent activities and processed in parallel. The results from each activity are concatenated (for extension, multiplication and division) or added (for marginalization) to obtain the final output. We use this property in Section 5.

4.1 Task Dependency Graph Construction

We construct a *task dependency graph* denoted G from a given junction tree $\mathbb{J}$ to describe the precedence constraints among the tasks. The task dependency graph is created in the following steps:

(I) *Junction tree rerooting*: The critical path of $\mathbb{J}$ is defined as a root-leaf path, where the execution time for processing the cliques on the path is the largest among all the root-leaf paths in $\mathbb{J}$. A junction tree can be rerooted at any clique and therefore leads to different critical paths [5]. Junction tree rerooting is to find a clique C as the new root that leads to the shortest critical path. We illustrate junction tree rerooting in Figure 3 (a) and (b), where we assume that processing each clique takes an unit time. After rerooting, the length of the critical path reduces from 5 to 4.

(II) *Construction of the clique updating graph*: In exact inference, evidence is firstly propagated from leaf cliques to the root and then from the root to the leaf cliques. Thus, the clique updating graph has two parts, both corresponding to $\mathbb{J}$ but the direction of edges is reversed. Figure 3 (c) shows a sample clique updating graph from the junction tree given in Figure 3 (b).

(III) *Construction of the task dependency graph*: Based on the clique updating graph, we construct *task dependency graph* G to describe the dependency relationship between the tasks defined in Section 4. Replacing each clique in the clique updating graph by the dependency among the tasks associated with the clique, we obtain the task dependency graph G for a junction tree $\mathbb{J}$. The tasks related to each clique are illustrated in Figure 3 (d).

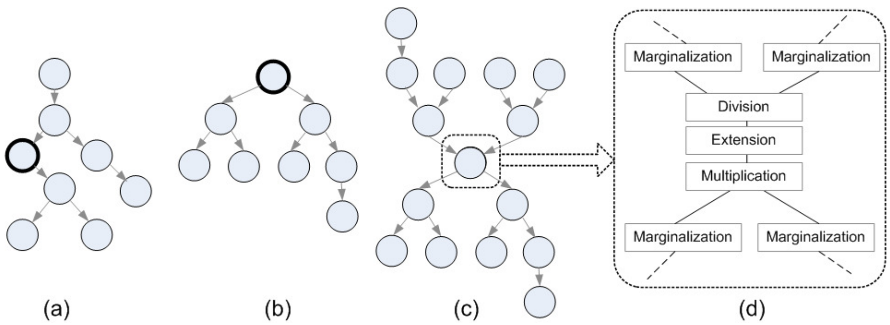

Fig. 3 (a) An arbitrary junction tree. (b) The rerooted junction tree at the bold clique. (c) An clique updating graph converted from the rerooted junction tree. (d) The task graph is obtained by replacing each clique in (c) with a set of tasks, each corresponding to a node level primitive.

4.2 Task Dependency Graph Modification due to Task Partitioning

As discussed in Section 2, the size of potential table increases dramatically even for moderate increase of clique width and the number of states of random variables in the clique. Thus, some tasks in the task dependency graph may involve large potential tables. Parallelizing the execution of these large tasks can accelerate exact inference, especially when the large tasks are in the critical path.

We partition large tasks dynamically by exploiting the property that node level primitives can process a portion of the potential table and the results can be combined to obtain the final output [7]. Once a task is partitioned, the task dependency graph must be updated accordingly to reflect the changes in dependency relationship among the tasks. Figure 4 illustrates task dependency graph modification in a partial task dependency graph, where the bold node is partitioned into a set of small tasks. The parents of the bold node become the preceding tasks of all the small tasks. We designate the last small task to be the successor of other small tasks, and let it inherit the successors of the bold node.

Note that such partitioning occurs locally and has no impact on other tasks. As shown in Figure 4, a large task is partitioned after all the preceding tasks complete. Thus, no change is needed for the completed tasks. For the successors of the large task, although one of their preceding tasks is replaced by the last small task, such replacement is transparent to the successors.

5 Workload-aware Scheduling for Evidence Propagation

5.1 Scheduler Organization

We assume that there are P threads in a system, each being bound to a separate core. The scheduler is shown in Figure 5, where the *global task list*

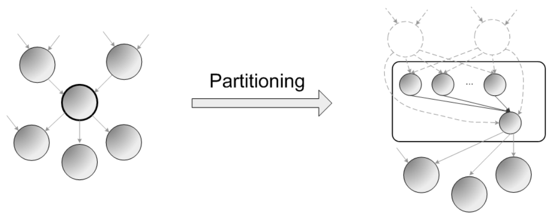

Fig. 4 Illustration of dynamic task dependency graph modification due to task partitioning. The task denoted by the bold node is partitioned when all the preceding tasks (gray dashed nodes) complete. All changes occur locally without affecting non-partitioned tasks.

(GL) is a list of tasks in a given task dependency graph G. Each element of GL stores a task and the related data, e.g., the dependency degree and the links to its succeeding tasks. Initially, the *dependency degree* of a task is the number of incoming edges of the task in G. Only the tasks with dependency degree equal to 0 can be processed. The GL is shared by all the threads and protected by locks.

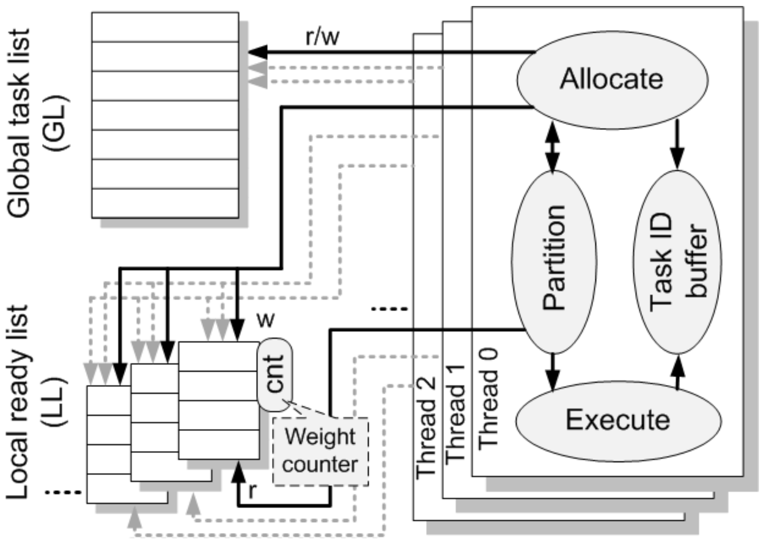

Fig. 5 Components of the scheduler for evidence propagation.

The *Allocate module* in each thread is in charge of decreasing task dependency degrees and allocating tasks to the threads. This module decreases the dependency degree of the *successors* of the tasks appearing in the *Task ID buffer*. Since each task ID correspond to the offset of a task in the GL, the module can locate a task in $O(1)$ time. Each module allocates tasks with dependency degree equal to 0 to the threads with the aim of load balancing.

Each thread has a *local ready list* (LL) to store the tasks allocated to the thread for execution. As shown in Figure 5, each LL can be written (insert tasks) by all the threads, but read (fetch tasks) by only one thread. Thus, the

LLs are actually global and locks are used to prevent concurrent write. Each LL has a *weight counter* to record the workload of the tasks currently in the LL.

The *Partition module* fetches tasks from the LL in the same thread and checks the weight of each fetched task. A fetched task T with heavy weight can be partitioned to subtasks $\hat{T}_0, \hat{T}_1, \cdots, \hat{T}_{n-1}$, each processing a part of the potential table associated to T (See Section 4.2). The module sends $\hat{T}_0$ to the Execute module and replaces T by $\hat{T}_{n-1}$ in GL. The dependency degree of $\hat{T}_{n-1}$ is set as $(n-1)$. The other subtasks are evenly distribute to LLs through the Allocate module.

Each thread also has a local *Execute module* where the primitive related to a task is performed. Once a task completes, the Execute module stores the ID of the task to the Task ID buffer. The Execute module also signals the Allocate module to take the next task, if any, from LL.

5.2 A Sample Implementation of the Scheduling Method

The workload-aware scheduling algorithm is shown in Algorithm 1, where the statements in boxes require synchronization with other threads. We use the following notations in the algorithm: GL is the global list. LL_i is the local ready list in Thread i. d_T and w_T denote the dependency degree and the weight of task T, respectively. W_i is the total weight of the tasks in LL_i. δ is a given threshold of the size of potential table. Any potential table larger than δ is partitioned.

Line 1 in Algorithm 1 initializes the Task ID buffers. Line 2 starts P threads to perform exact inference in parallel. As shown in Line 3, the scheduler continues until all the tasks are processed. Lines 4-10 correspond to the Allocate module, where the scheduler updates the dependency degree of the successors of the tasks in the Task ID buffer. Line 7 distributes ready-to-execute tasks to the LL with the lowest workload for load balancing. Thus, LL is actually shared by all the threads for appending new tasks. Line 11 is the Fetch module, where the tasks in LL_i are fetched by Thread_i only. Lines 12-18 correspond to the Partition module and Execute Module. Line 13 partitions tasks where the size of the corresponding potential table is larger than a given threshold δ. In our implementation, δ is set as the average size of the potential tables in the input junction tree. Line 15 modifies the task dependency graph to include new tasks due to task partitioning. (see Figure 4). Note that task partitioning is a local activity. In Algorithm 1, the statements in boxes require synchronization with other threads.

Algorithm 1 achieves load balancing by two means: First, the Allocate module ensures that the new tasks are allocated to the threads where the total workload of the tasks in its LL is the lowest. Second, the Partition module guarantees that each single large task can be processed in parallel.

Algorithm 1 Workload-aware Task Scheduling

```
1: $\forall\ T$ s.t. $d_T = 0$, evenly distribute $T$ to $\mathrm{LL}_i$ $(i = 0 \dots P-1)$
2: for Thread $i$ $(i = 0 \dots P-1)$ pardo
3:     while $\mathrm{GL} \cup \mathrm{LL}_i \neq \emptyset$ do
           {Allocate module}
4:         for $T \in \{$ successors of tasks in the $i$-th Task ID buffer $\}$ do
5:             $\boxed{d_T = d_T - 1}$
6:             if $d_T = 0$ then
7:                 $\boxed{\text{allocate } T \text{ to } \mathrm{LL}_j \text{ where } j = \arg_t \min(W_t), t = 0 \cdots P-1}$
8:                 $\boxed{W_j = W_j + w_T}$
9:             end if
10:        end for
           {Partition and Execute modules}
11:        fetch task $T'$ from $\mathrm{LL}_i$
12:        if the size of the potential table associated with $T' > \delta$ then
13:            partition $T'$ into $\hat{T'}_0, \hat{T'}_1, \cdots, \hat{T'}_{n-1}$, each smaller than $\delta$
14:            $\boxed{\text{replace } T' \text{ in GL with } \hat{T'}_{n-1}\text{, and allocate } \hat{T'}_0, \cdots, \hat{T'}_{n-2} \text{ as in Step 7}}$
15:            execute $\hat{T'}_0$ and place the task ID of $\hat{T'}_0$ into the $i$-th Task ID buffer
16:        else
17:            execute $T'$ and place the task ID of $T'$ into the $i$-th Task ID buffer
18:        end if
19:    end while
20: end for
```

5.3 Scheduling Optimization Techniques

Algorithm 1 is a sample implementation of the proposed method, aiming at demonstrating the ideas of task partitioning and thread collaboration for exact inference. Scheduling optimization techniques can be used for Algorithm 1 to improve the performance on multicore platforms. For example, in Lines 4-9, we access the shared data d_T, LL_j and W_j in batch mode to reduce synchronization overhead, since in batch mode we do not need to frequently request/release locks. In Line 7, we can exploit the affinity among tasks for allocation. In Line 11, we prioritize the tasks with more successors. The task allocation in Line 7 is a straightforward heuristic for load balancing. In addition to load balancing, we also consider affinity among tasks. That is, when the workload of the LLs is similar to each other, the new task is assigned to the local list where (some of) its preceding tasks were assigned, so that we can improve the data locality. In Line 11, we prioritize the tasks with more successors, so that we can shortly have more available tasks for scheduling. These optimization techniques adapt the proposed scheduler to general-purpose multicore platforms.

6 Experiments

We conducted experiments on Intel Xeon x86_64 E5335 and AMD Opteron 2347 platforms, each having two quadcore processors with 16 GB DDR2 memory. The Intel processor ran at 2.00 GHz with 4 MB L2 cache and the operating system was Red Hat Enterprise Linux WS release 4. The AMD processor ran at 1.9 GHz with 2 MB L2 cache and the operating system was Red Hat Linux CentOS version 5. We used GCC 4.1.2 compiler for both of the systems.

To evaluate the performance of junction tree rerooting, we used the template shown in Figure 6, a tree with $(b+1)$ branches rooted at R. We generated four junction trees by setting b as 1, 2, 4 and 8. Each tree had 512 cliques consisting of 15 binary variables. Thus, the serial complexity of performing exact inference on these junction trees is approximately equal. R' becomes the new root after rerooting. We disabled task partitioning, since it provides parallelism at data level.

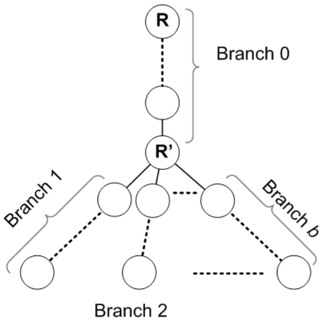

Fig. 6 Junction tree template for evaluating rerooting algorithm.

The results are shown in Figure 7. The speedup in Figure 7 was defined as $Sp = t_R/t_{R'}$, where t_R $(t_{R'})$ is the execution time of evidence propagation in the original (rerooted) junction tree. Note that when R is the root, (Branch 0 + Branch 1) is a critical path. When R' is the root, Branch 0 is a critical path. Thus, the maximum speedup is 2, if the number of concurrent threads P is larger than b. The observations in Figure 7 matched the analysis.

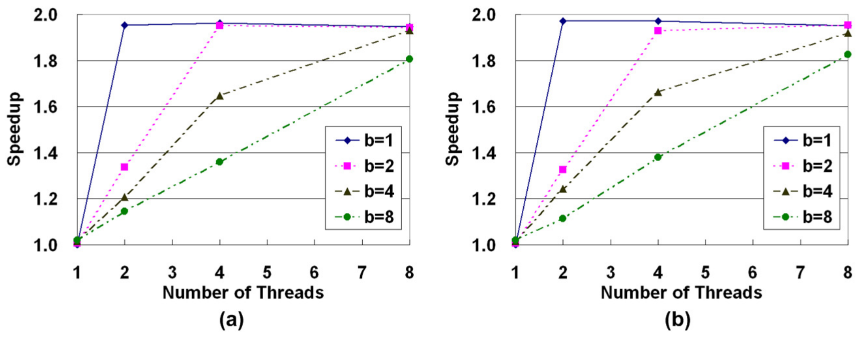

Fig. 7 Speedup due to rerooting on (a) Intel Xeon and (b) AMD Opteron. $(b+1)$ is the total number of branches in the junction tree.

	#cliques	clique width	clique degree	#states
Junction tree 1	512	20	4	2
Junction tree 2	256	15	4	3
Junction tree 3	128	10	2	3

Table 1 The parameters of the junction trees for evaluating the proposed method.

We generated junction trees of various sizes using the Bayes Net Toolbox [12] to evaluate the performance of the proposed evidence propagation method. The parameters of these junction trees are shown in Table 1. All the three junction trees were rerooted. Double precision floating point data were used in the potential tables. We first performed exact inference with respect to the three junction trees using Intel Open Source Probabilistic Network Library (PNL), a full function, free, open source, graphical model library with a parallel implementation [13]. The scalability of the results is shown in Figure 8. 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. So, PNL did not scale well for exact inference.

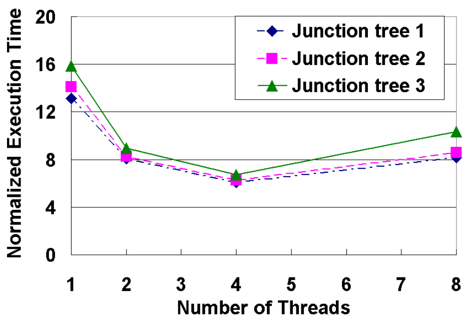

Fig. 8 Scalability of exact inference using PNL library for various junction trees.

We compared the proposed method with two parallel baselines on both Intel and AMD platforms in Figure 9. The first baseline used OpenMP intrinsic functions to parallelize the sequential code. The second baseline was called *data parallel method*, where we created multiple threads once a node level primitive was performed. The data parallel method is similar to the task partitioning mechanism in our workload-aware scheduler, but the overheads were large. We show the speedups in Figure 9. The results show that the proposed method exhibited almost linear speedup and was superior compared with the baseline methods on both the platforms. Performing the proposed method on 8 cores, we observed speedup of 7.4 on Intel Xeon and 7.1 on AMD Opteron. This speedup is approximately twice the speedup achieved by the OpenMP based method.

To show the load balance and the overhead of the workload-aware scheduler, we measured the computation time for each thread. By computation time, we mean the overall time taken by performing node level primitives for

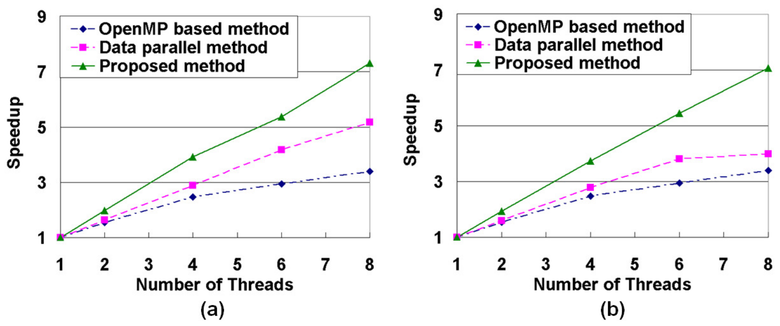

Fig. 9 Scalability of exact inference using various methods on (a) Intel Xeon and (b) AMD Opteron.

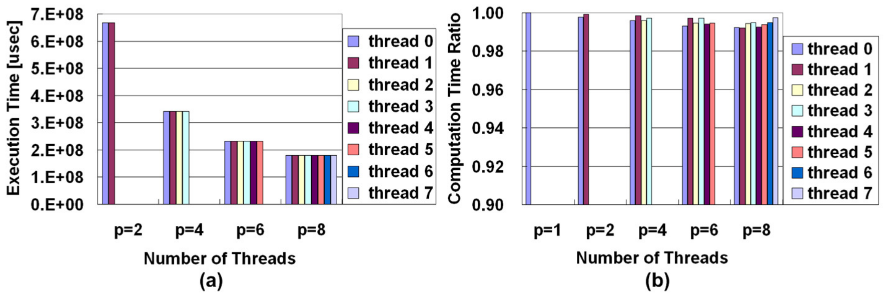

Fig. 10 (a) Load balance across the threads; (b) Computation time ratio for each thread.

each thread. Thus, the time taken to fetch and allocate tasks were not considered. We show the results obtained on Opteron using Junction tree 1 in Figure 10 (a). We observed similar results for Junction trees 2 and 3. We also calculated the ratio of the computation time over the overall execution time to illustrate the quality of the scheduler. From Figure 10 (b), we can see that, although the scheduling time increased a little bit as the number of threads increases, it was not exceeding 0.9% of the overall execution time.

Finally, we modified parameters of Junction tree 1 to observe the performance of our method in various situations. We varied the number of cliques N, clique width $w_{\mathcal{C}}$, number of states r and the average clique degree k. From Figure 11(a), we observed that the results exhibited almost linear speedup. In Figure 11(b) and (c), the results with $w_{\mathcal{C}} = 10$ and $r = 2$ showed lower speedup due to the small potential table. For $w_{\mathcal{C}} = 10$ and $r = 2$, the potential table size is about 1/1000 of that with $w_{\mathcal{C}} = 20$. Since N and the junction tree structures were the same, the scheduling requires approximately the same time. Thus, the scheduling overhead became relatively large and therefore affected the speedup. In Figure 11 (d), all the results had similar performance when k was varied, since k had limited impact on the parallelism in evidence propagation.

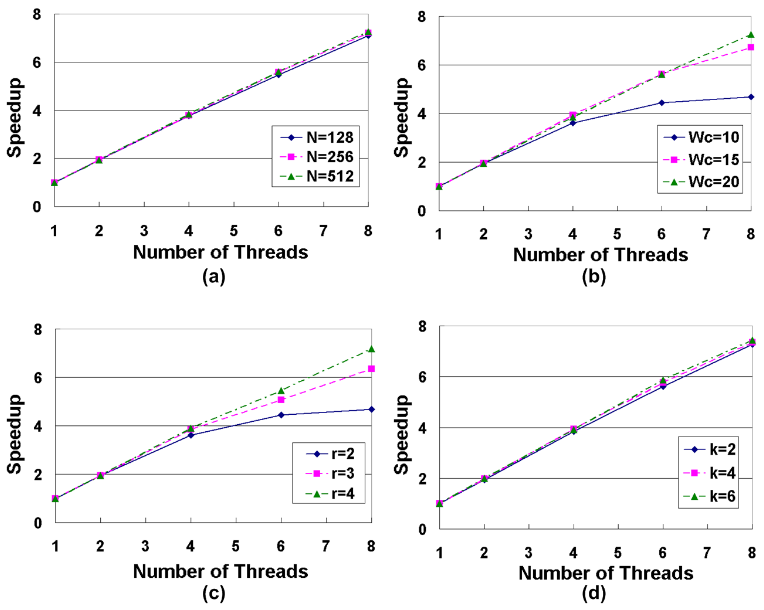

Fig. 11 Speedups of exact inference on multicore systems with respect to various junction tree parameters.

In addition to the above experiments on synthetic dataset, we conducted an experiment using a Bayesian network from a real application. The Bayesian network is called the Quick Medical Reference decision theoretic version (QMR-DT), which is used in a microcomputer-based decision support tool for diagnosis in internal medicine [14,15]. There were 1000 nodes in this 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 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.

The results shown in Figure 12 is consistent with that in Figure 11, where we can see that the proposed parallel exact inference method scaled well for datasets from both synthetic problems and real applications. Since the average clique width was 10 for this application, the speedup in Figure 12 (b) is close to that in Figure 11 (b) with the same clique width.

7 Conclusions

We explored parallelism at various levels for evidence propagation on general-purpose multicore processors. Our technique converted a given junction tree into a task dependency graph and scheduled the tasks using a workload-aware scheduler. The proposed scheduler monitored the workload dynamically and partitioned tasks involving large potential tables. Our implementation

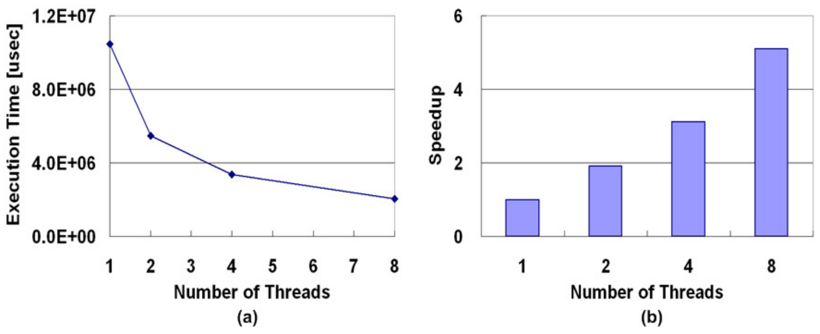

Fig. 12 (a) Execution time and the corresponding (b) speedup for exact inference in a junction tree converted from the QMR-DT network on multicore processors.

achieved higher performance compared with the baseline methods. In the future, we plan to investigate the overheads in the workload-aware scheduler on multicore systems. We intend to reduce such overhead to improve the scheduler for scheduling a class of DAG structured computations.

References

1. Lauritzen, S.L., Spiegelhalter, D.J.: Local computation with probabilities and graphical structures and their application to expert systems. J. Royal Statistical Society B **50** 157–224 (1988)
2. Heckerman, D.: Bayesian networks for data mining. In: Data Mining and Knowledge Discovery. (1997)
3. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach (2nd Edition). Prentice Hall (2002)
4. Segal, E., Taskar, B., Gasch, A., Friedman, N., Koller, D.: Rich probabilistic models for gene expression. In: 9th International Conference on Intelligent Systems for Molecular Biology. 243–252 (2001)
5. Pennock, D.: Logarithmic time parallel Bayesian inference. In: Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence. 431–438 (1998)
6. Kozlov, A.V., Singh, J.P.: A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In: Supercomputing. 320–329 (1994)
7. Xia, Y., Prasanna, V.K.: Node level primitives for parallel exact inference. In: Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing. 221–228 (2007)
8. Xia, Y., Prasanna, V.K.: Parallel exact inference on the cell broadband engine processor. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 1–12 (2008)
9. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Technical report, Cambridge (1996)
10. Intel Threading Building Blocks: (http://www.threadingbuldingblocks.org/)
11. OpenMP Application Programming Interface: (http://www.openmp.org/)
12. Murphy, K.: (http://www.cs.ubc.ca/∼murphyk/software/bnt/ bnt.html)
13. Intel Open Source Probabilistic Networks Library: (http:// www.intel.com/tech nology/computing/pnl/)
14. Middleton, B., S, M.S.M., Heckerman, D., D, H.L.M., Cooper, G.: Probabilistic diagnosis using a reformulation of the INTERNIST-1/QMR knowledge base. Medicine **30** 241–255 (1991)
15. Jaakkola, T.S., Jordan, M.I.: Variational probabilistic inference and the QMR-DT network. Journal of Artificial Intelligence Research **10**(1) 291–322 (1999)