Self-adaptive Evidence Propagation on Manycore Processors

Yinglong Xia
IBM T.J. Watson Research Center
1101 Kitchawan Rd.
Yorktown Heights, NY 10598, USA
Email: yxia@us.ibm.com

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

***Abstract*—Evidence propagation is a major step in exact inference, a key problem in exploring probabilistic graphical models. Evidence propagation is essentially a series of computations between the potential tables in cliques and separators of a given junction tree. In real applications, the size of the potential tables varies dramatically. Thus, to achieve scalability over dozens of threads remains a fundamental challenge for evidence propagation on manycore processors. In this paper, we propose a self-adaptive method for evidence propagation on manycore processors. Given an arbitrary junction tree, we convert evidence propagation in the junction tree into a task dependency graph. The proposed self-adaptive scheduler dynamically adjusts the number of threads for scheduling or executing tasks according to the task dependency graph. Such a self-adaptability prevents the schedulers being too idle or too busy during the scheduling process. We implemented the proposed method on the Sun UltraSPARC T2 (Niagara 2) platform that supports up to 64 hardware threads. Through a set of experiments, we show that the proposed method scales well with respect to various input junction trees and exhibits superior performance when compared with several baseline methods for evidence propagation.**

***Keywords*-exact inference; manycore processor; self-adaptive scheduling; graphical models;**

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 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. Thus, parallel techniques must be developed.

It is challenging to explore exact inference on manycore processors. The trend in architecture design is to integrate more and more cores onto a single chip to achieve higher performance. Examples of existing manycore processors include Sun UltraSPARC T1 (Niagara) and T2 (Niagara 2), which support up to 32 and 64 concurrent threads, respectively [6]. NVIDIA Tesla and Tilera TILE64 are also available. In addition, manycore processors are emerging soon, such as the Sun Rainbow Falls and IBM Cyclops64 [7]. Such processors are optimized to address how many tasks can be completed efficiently over a period of time rather than how quickly an individual task can be completed. The main challenges for parallel computing on manycore processors include achieving scalability over a large range and reducing possible overhead due to the synchronization among the cores.

Our contributions in this paper include: (a) We present a self-adaptive scheduler to allocate tasks to threads at runtime. By self-adaptive, we mean the scheduler dynamically adjusts the number of threads for scheduling or computation according to the input junction tree. To the best of our knowledge, this is the first general-purpose manycore solution for evidence propagation. (b) We implement the proposed method on the Sun UltraSPARC T2 (Niagara 2) platforms. (c) We conduct extensive experiments to validate the proposed method. When 64 threads were used, the speedup of the proposed method is 13.7 and 9.4 compared with the OpenMP and Charm++ based implementations, respectively.

The rest of the paper is organized as follows: In Section II, we review the background. Section III discusses related work. Section IV presents the data representation and self-adaptive scheduling scheme. We illustrate experimental results in Section V and address future research directions in Section VI.

II. Background

A. Exact Inference

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. Notice that these edges can *not* form directed cycles. Thus, 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. $\text{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$, 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*. 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 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 $\text{J} = (\mathbb{T}, \hat{\mathbb{P}})$, 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.

In a junction tree, exact inference proceeds as follows: Assuming evidence is $\text{E} = \{A_i = a\}$ 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]:

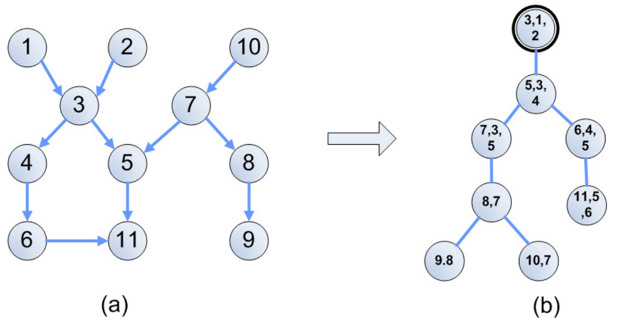


Figure 1. (a) A sample Bayesian network and (b) corresponding junction tree.

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

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

B. DAG Structured Computations on Manycore Processors

As many computational solutions, evidence propagation can be expressed as directed acyclic graphs (DAGs) with weighted nodes. The *weight* for a task is the estimated execution time of the task. A task can begin execution only if all of its predecessors have been completed [8].

Manycore processors incur challenges for scheduling DAGs. Directly utilizing traditional scheduling methods such as centralized or distributed scheduling can degrade the performance of DAG structured computations on manycore processors. Centralized scheduling has a single thread to allocate tasks, which may not be able to serve the rest of the threads in time. This leads to starvation of some threads, especially when the tasks complete quickly. On the other hand, distributed scheduling requires many threads to schedule tasks. This limits the resources for task execution. In addition, many schedulers accessing shared variables can result in costly synchronization overhead. Therefore, an efficient scheduling method on manycore processors must be able to adapt itself to input task dependency graphs.

III. Related Work

There are several works on parallel exact inference, such as Pennock [5], Kozlov and Singh [9] and Szolovits. However, some of those methods, such as [9], are dependent upon the structure of the Bayesian network. The performance is adversely affected if the structure of the input Bayesian network is changed. Our method can be used for Bayesian networks and junction trees with various structures. Some other methods, such as [5], exhibit limited performance for multiple evidence inputs. The performance of our method does not depend on the number of evidence cliques. In [10], the node level primitives are parallelized using message passing on distributed memory platforms. The optimization proposed in [10] is not applicable in this paper, since our platforms have shared memory. A centralized scheduler for exact inference is introduced in [11], which is implemented on Cell BE, a heterogeneous multicore processor with a PowerPC element (PPE) and 8 computing elements (SPEs). The centralized scheduler is hosted by the PPE and allocates tasks onto SPEs. However, the platforms studied in this paper are homogeneous and each individual core is relatively simple. Thus, we allow multiple cores to host the schedulers. In our proposed method, the number of schedulers is dynamically adjusted according to the input tasks. Although GPGPU is also a type of manycore processors, the architecture of GPGPU is significantly different from the manycore processors that we concerned in this paper, such as Sun Niagara T1/T2. For example, existing GPGPUs are lack of efficient synchronization mechanism among streaming multiprocessors (SMs), which prevents user-designed schedulers assigning tasks onto specific cores directly. Thus, we discuss the techniques for exact inference on GPGPUs in a separate paper [12]. In this paper, we deviate from the above approaches and device a novel task scheduling technique specifically for exact inference.

The scheduling problem has been extensively studied for several decades [13], [14], [15], [16]. The research on scheduling DAGs includes [17] where the authors studied the problem of scheduling more than one DAG simultaneously onto a set of heterogeneous resources. In [13], a game theory based scheduler on multicore processors for minimizing energy consumption was proposed. Dongarra *et al.* proposed dynamic schedulers optimized for a class of linear algebra problems on general-purpose multicore processors [16]. Scheduling techniques have been proposed by several emerging programming systems such as Cilk [18], Intel Threading Building Blocks (TBB) [19], OpenMP [20], Charm++ [21], etc. Although these techniques perform well in practice on general-purpose *multicore* processors, they are not specially designed for applications such as exact inference on general-purpose manycore systems. For example, increase in the number of cores leads to decreased probability of successfully stealing tasks when only a few threads have available tasks. In addition, to the best of our knowledge, the number of threads for scheduling or execution can not be dynamically adjusted in these techniques. Our proposed method addresses the above issues.

IV. Self-Adaptive Evidence Propagation

A. From Evidence Propagation to Task Dependency Graph

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* [10]. In this paper, we define a *task* as the computation of a node level primitive. The input to each task is one or two potential tables, depending on the specific primitive type. The output is an updated potential table. The details of the primitives are discussed in [10]. We illustrate the tasks related to clique $\mathcal{C}$ in Figure 2(b). A number within brackets corresponds to a task. The primitive type is given in Figure 2(c). The dashed arrows in Figure 2(b) illustrate whether the task works on the same potential table or between two potential tables. An edge in Figure 2(c) represents the precedence order of the execution of the tasks.

A property of the primitives is that a potential table can be partitioned into independent activities and processed in parallel. The results from each activity are combined (for extension, multiplication and division) or added (for marginalization) to obtain the final output.

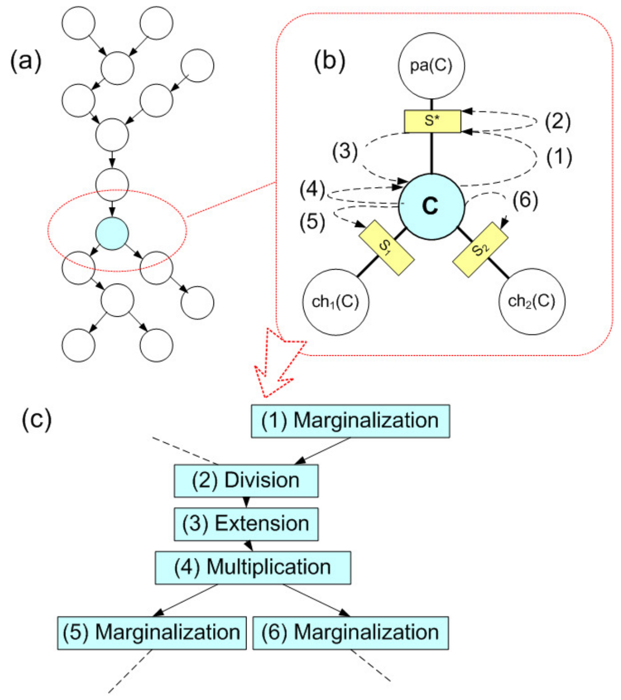


Figure 2. (a) Clique updating graph; (b) Primitives used to update a clique; (c) Local task dependency graph with respect to the clique in (b).

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

First, we construct a *clique updating graph* to describe the coarse grained dependency relationship between cliques in $\mathbb{J}$. In exact inference, $\mathbb{J}$ is updated twice [1]: (1) evidence is propagated from leaf cliques to the root; (2) evidence is then propagated from the root to the leaf cliques. Thus, the clique updating graph has two symmetric parts. In the first part, each clique depends on all its children in $\mathbb{J}$. In the second part, each clique depends on its parent in $\mathbb{J}$. Figure 2(a) shows a sample clique updating graph from the junction tree given in Figure 1(b).

Second, based on the clique updating graph, we construct *task dependency graph* G to describe the dependency relationship between the tasks. The tasks related to a clique $\mathcal{C}$ are shown in Figure 2(b). Considering the precedence order of the tasks, we obtain a small DAG called a *local task dependency graph* (see Figure 2(c)). In Figure 2(b), $\mathcal{S}^*$, $\mathcal{S}_1$ and $\mathcal{S}_2$ are separators between clique $\mathcal{C}$ and its parent $pa(\mathcal{C})$ and two children $ch_1(\mathcal{C}), ch_2(\mathcal{C})$, respectively. Each dashed arrow represents a primitive [10] between the potential tables of separators or cliques. For instance, the dashed arrow with number (1) represents a primitive called marginalization which marginalizes the potential table of $\mathcal{C}$ into the potential table of $\mathcal{S}^*$ (see [10] for details). Replacing each clique in Figure 2(a) with its corresponding local task dependency graph, we obtain the task dependency graph G for $\mathbb{J}$.

B. *Self-adaptive Scheduling*

The input task dependency graph is represented by a list called the *global task list* (GL). Figure 3(a) shows a portion of the task dependency graph. Figure 3(b) shows the corresponding part of the GL. As shown in Figure 3(c), each element in the GL consists of task ID, dependency degree, task weight, successors and the *task meta data*, i.e., potential tables related to the task. The *task ID* is the unique identity of a task. The *dependency degree* of a task is initially set as the number of incoming edges of the task. During the scheduling process, we decrease the dependency degree of a task once a predecessor of the task is processed. The *task weight* is the estimated execution time of the task. We keep the task IDs of the *successors* along with each task to preserve the precedence constraints of the task dependency graph. When we process a task T_i, we can locate its successors directly using the successor IDs, instead of traversing the entire list. The GL is shared by all the threads.

We illustrate the components of the self-adaptive scheduler in Figure 4. The boxes with rounded corners represent thread groups. Each group consists of a *manager* thread and several *worker* threads. The components inside a box are private to the group; while the components outside the boxes are shared by all groups.

The *global ready list* (GRL) in Figure 4 stores the IDs of tasks with current dependency degree equal to 0. These tasks are ready to be executed. During the scheduling process, a task is put into this list by a manager thread once the dependency degree of the task becomes 0.

The *local ready list* (LRL) in each group stores the IDs of tasks allocated to the group by the manager of the group. The workers in the group fetch tasks from LRL for execution. Each LRL is associated with a *workload indicator* (WI) to record the overall workload of the tasks currently in the LRL. Once a task is inserted into (or fetched from) the LRL, the indicator is updated.

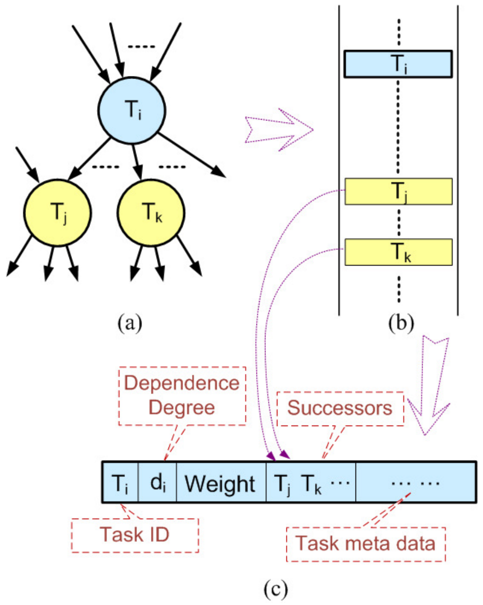


Figure 3. (a) A portion of a task dependency graph. (b) The corresponding representation of the global task list (GL). (c) The data of element T_i in the GL.

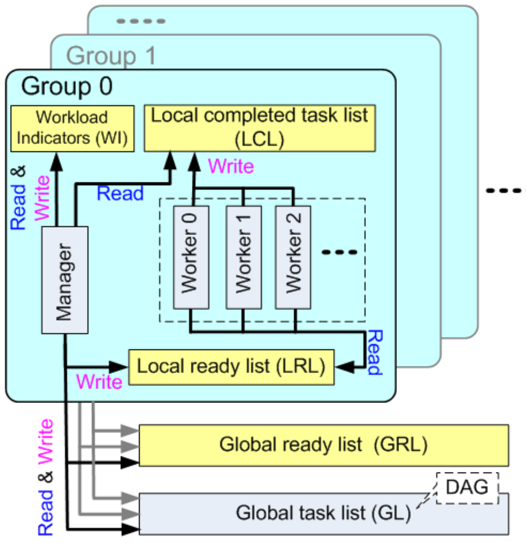


Figure 4. Components of the self-adaptive scheduler.

The *local completed task list* (LCL) in each group stores the IDs of tasks completed by a worker thread in the group. The list is read by the manager thread in the group for decreasing the dependency degree of the successors of the tasks in the list.

The arrows in Figure 4 illustrate how each thread accesses a component (read or write). As we can see, GL and GRL are shared by all the managers for both read and write. For each group, the LRL is write-only for the manager and read-only for the workers; while LCL is write-only for the workers and read-only for the manager. WI is local to the manager in the respective group only.

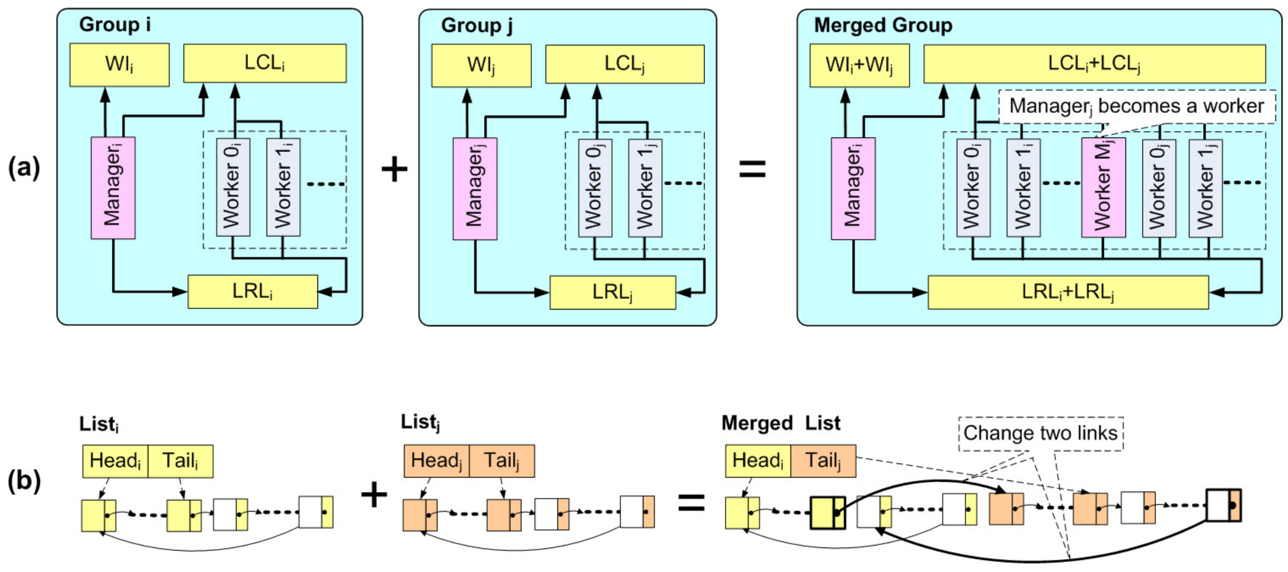


Figure 5. (a) Merge Group$_i$ and Group$_j$. (b) Merge circular lists $List_i$ and $List_j$. The head (tail) points to the first (last) tasks stored in the list. The blank elements have no task stored yet.

C. Dynamic Thread Grouping

The scheduler organization shown in Figure 4 supports dynamic thread grouping, which means that the number of threads in a group can be adjusted at runtime. We adjust groups by either merging two groups or partitioning a group. The proposed organization of the adaptive scheduler ensures efficient group merging and partitioning.

Figure 5(a) illustrates the merging of Group$_i$ and Group$_j$, $i < j$. The two groups are merged by converting all threads of Group$_j$ into the workers of Group$_i$ and merging WIs, LCLs and LRLs accordingly. Converting threads of Group$_j$ into the workers of Group$_i$ is straightforward: $Manager_j$ stops allocating tasks to Group$_j$, but performs self-scheduling as a worker thread. Then, all the threads in Group$_j$ access tasks from the merged LRL and LCL. To combine WI_i and WI_j, we add the value of WI_j to WI_i. Although WI_j is not used after merging, we still keep it updated for the sake of possible future group partitioning. Merging the lists i.e. LCLs and LRLs is efficient. Note that both LCL and LRL are circular lists, each having a head and a tail pointer to indicate the first and last tasks stored in the list, respectively. Figure 5(b) illustrates the approach to merge two circular lists. We need to update two links only, i.e. the bold arrows shown in Figure 5(b). None of the tasks stored in the lists are moved or duplicated. The head and tail of the merged list are $Head_i$ and $Tail_j$, respectively. Note that two merged groups can be further merged into a larger group.

Group$_i$ and Group$_j$ can be restored from the merged group by partitioning. As a reverse process of group merging, group partitioning is also straightforward and efficient. Due to space limitations, we do not elaborate it here.

D. Complete Algorithm

Based on the organization shown in Section IV-B, we show the proposed algorithm in Algorithms 1. Before discussing details of the algorithm, we give a brief overview: Algorithms 1 let each thread run in parallel. According to the thread ID (Line 10), a thread works as a scheduler or a worker. The schedulers maintain precedence constraints among the tasks and the workers executes the tasks fed by the scheduler. In addition, the first scheduler thread (Line 25) determines if two groups should be merged or not (Lines 27-32).

We use the following notations to describe the implementation: Assume there are P threads, each bound to a core. The threads are divided into groups consisting of a manager and several workers. GL and GRL denote the global task list and global ready list, respectively. LRL_i and LCL_i denote the local ready list and local completed task list of Group$_i$, $0 \le i < P$. d_T and w_T represent the dependency degree and the weight of task T, respectively. WI_i is the workload indicator of Thread$_i$. Parameters δ_M, δ_+ and δ_- are constant thresholds. Empirically, we set the three thresholds as $3\sum_{T\in GL} d_T/|GL|$, $1.25\sum_{T\in GL} d_T/|GL|$, $0.75\sum_{T\in GL} d_T/|GL|$, respectively. These thresholds can be tuned with respect to the platforms. We use a variable *rank* to control the merge or participation of the groups to maintain 2^{rank} groups during the scheduling. We let β denote the average number of successors of tasks.

The complete algorithm is shown in Algorithm 1. Lines 1-6 are initial steps. Lines 1-4 construct the task dependency graph from the given junction tree $\mathbb{J}$. Lines 5 and 6 initialize the scheduling process. Since the computation complexity of the initial steps is very low compared with the potential table

Algorithm 1 Self-adaptive evidence propagation

```
Input: Junction tree $\mathbb{J}$, Number of threads $P$
Output: Updated potential tables in $\mathbb{J}$
1: Let $\mathbb{J}' = \mathbb{J}$ with all edges in $\mathbb{J}'$ reversed
2: Combine $\mathbb{J}'$ and $\mathbb{J}$ by merging the roots
3: Replace each clique by tasks shown in Fig.2(c)
4: Store the tasks in $GL$ as shown in Figure 3
5: Distribute tasks $T_i \in GL$ which have $d_i = 0$ across $LRL_j$, $0 \le j < P$
6: $f_{exit}$ =false, $rank = 1$
7: for Thread $i = 0, 1, \cdots, P-1$ pardo
8:   while $f_{exit}$ =false do
9:     $Q = 2^{rank}$, $m = \lfloor i/Q \rfloor$
10:    if $i\%Q = 0$ then
11:      $\Delta W = \sum_{\tilde{T} \in LCL_m} w_{\tilde{T}}$, $WI_m = WI_m - \Delta W$
12:      for all $T \in \{$successors of $\tilde{T} \in LCL_i\}$ do
13:        $d_T = d_T - 1$
14:        if $d_T = 0$ then
15:          $GRL = GRL \cup \{T\}$; $GL = GL \backslash \{T\}$
16:        end if
17:      end for
18:      if $LRL_m$ is not full then
19:        $S' \Leftarrow$ fetch $(Q-1)$ tasks from $GRL$
20:        if $\sum_{T \in S'} w_T < \Delta W$ or $WI_m < \delta_M$ then
21:          Fetch more tasks from $GRL$ to $S'$ so that $\sum_{T \in S'} w_T \approx \Delta W + \delta_M$
22:        end if
23:        $LRL_m = LRL_m \cup \{S'\}$, $WI_m = WI_m + \sum_{T \in S'} w_T$
24:      end if
25:      if $i = 0$ then
26:        $r = \sum_{j=0}^{P/Q} (WI_j / (|LCL_j| \cdot \beta))$, $rank_{old} = rank$
27:        $rank = \begin{cases} \min(rank+1, \log P), & r > \delta_+ \\ \max(rank-1, 1), & r < \delta_- \end{cases}$
28:        if $rank_{old} < rank$ then
29:          Group$_j$ = Merge(Group$_{2j}$, Group$_{2j+1}$), $\forall 0 \le j < \frac{P}{2 \cdot Q} - 1$
30:        else if $rank_{old} > rank$ then
31:          (Group$_{2j}$, Group$_{2j+1}$) = Partition(Group$_j$), $j = \frac{P}{Q} - 1, \cdots, 1, 0$
32:        end if
33:      end if
34:      if $GL = \emptyset$, then let $f_{exit}$=true
35:    else
36:      if $LRL_m$ is not empty then
37:        Fetch $T$ from $LRL_m$
38:        Execute task $T$
39:        $LCL_m = LCL_m \cup \{T\}$
40:      end if
41:    end if
42:  end while
43: end for
```

update, we simply let a single thread to perform these steps. Lines 8-41 are the main body of the proposed algorithm. In Line 9, Q is the number of threads in a group and m is the group to which Thread i belongs. Lines 10 selects the first thread in each group to be the manager and the others be the workers. Lines 11-24 describe the work of the manager in Group m. Basically, the manager updates WI_m and d_T for all completed tasks. New ready-to-execute tasks are added to GRL (Lines 14-16) and part of the ready-to-execute tasks are added to LRL_m accordingly (Lines 18-24). The manager in Group 0 is also in charge of group merging/partitioning (Lines 25-34). Line 26 judges if the groups need to be merged (Line 29) or partitioned (Line 31), where β is the average number of successors for each task. Note that, in Line 26), WI_j gives the workload for the workers in Group j; while $|LCL_j| \cdot \beta$ shows the workload for the manager. Thus, by comparing with parameters δ_+ and δ_-, the ratio r hints if we need more managers or more workers (Line 27). Line 27 also ensures the range of $rank$, i.e., $1 \le rank \le \log P$. Group partitioning increases the number of managers; while group merging increases the number of workers. The two functions Merge$(\cdot,\cdot)$ and Partition$(\cdot)$ are discussed in Section IV-C. When all tasks are completed, the manager in Group 0 notifies all threads to quit by flipping f_{exit}. Lines 36-40 describe the algorithm for workers. Each worker fetches tasks allocated by the corresponding manager (Line 37) for execution. The completed tasks in Group m are added to LCL_m in Line 39, so that the manager can process them.

V. Experiments

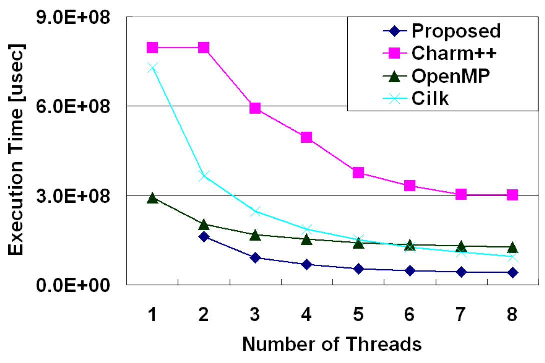
A. Computing Facilities

The Sun UltraSPARC T2 (Niagara 2) platform was a Sunfire T2000 server with a Sun UltraSPARC T2 multithreading processor [6]. UltraSPARC T2 has 8 hardware multithreading cores, each running at 1.4 GHz. In addition, each core supports up to 8 hardware threads with 2 shared pipelines. Thus, there are 64 hardware threads. Each core has its own L1 cache shared by the threads within a core. The L2 cache size is 4 MB, shared by all hardware threads. The platform had 32 GB DDR2 memory shared by all the cores. The operating system was Sun Solaris 11 and we used Sun Studio CC with Level 4 optimization (-xO4) to compile the code.

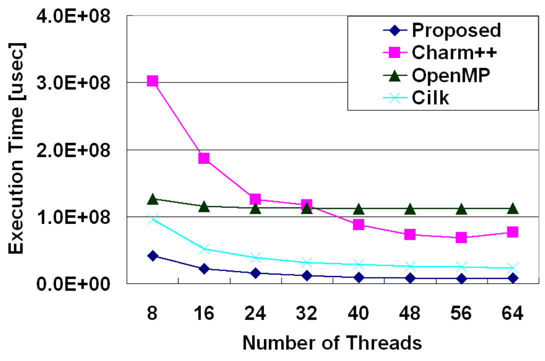
B. Datasets

In our experiments, we used various junction trees to analyze and evaluate the performance of our method. All the junction trees were generated using Bayes Net Toolbox [22]. The number of cliques N of these junction trees varied from 512 to 2048. The clique width w was selected from 8 to 20. The width of separator w_s was from 1 to 4. The number of states r for the random variables in the these junction trees varied from 2 to 4. The number of successors d of the cliques in these junction trees was selected from 2 to 16.

We stored the data of each junction tree as two parts in the memory. The first part was a list representing the structure of the junction tree (see Figure 3). The other part was an array, where each element was a structured variable consisting of all the data related to a clique, such as the clique potential table, separators between the clique and its successors. All the potential tables were represented using fixed point data, since the floating point units were shared by all the hardware threads in a core. C. Shi has studied the accuracy constraint of the fixed point data compared with floating point data [23].

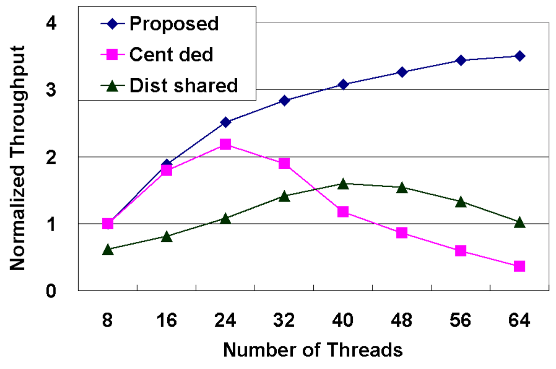


(a) Scalability with respect to 1-8 threads

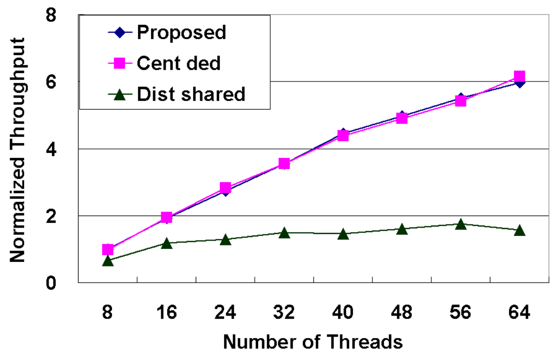


(b) Scalability with respect to 8-64 threads

Figure 6. Comparison with existing parallel programming systems.



(a) Speedup with respect to small POTs



(b) Speedup with respect to large POTs

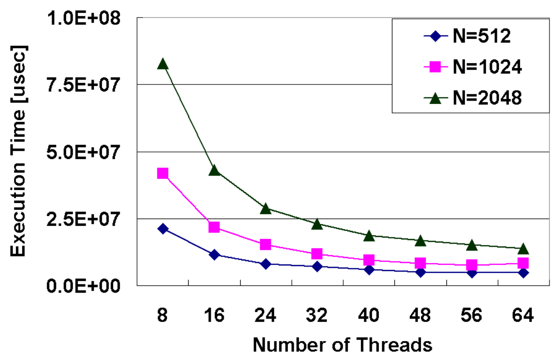
Figure 7. Comparison with baseline scheduling methods using junction trees of various potential table (POT) sizes.

C. Baseline Implementations

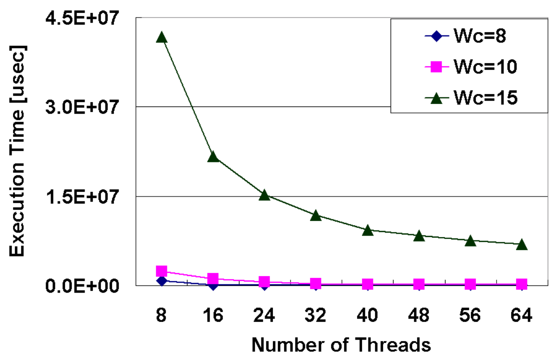
In order to evaluate the performance of the proposed algorithm, we implemented evidence propagation using five baseline methods, i.e., (1) `Charm++` [21], (2) `OpenMP` [20] and (3) `Cilk` [18]. (4) `Cent ded` dedicated a thread as the manager to monitor all the rest of the threads. The manager constructed the task dependency graph from a given junction tree, and allocated ready-to-execute tasks to the threads. The dedicated thread was also in charge of maintaining the precedence of the tasks. (5) `Dist shared` let each thread allocate tasks for itself and put all ready-to-execute tasks into a shared list. Each thread locked the shared list to fetch tasks and add new ready-to-execute tasks. We used mutex lock in the implementation. We evaluated the baseline methods along with the proposed scheduler using the same input task dependency graphs.

D. Results

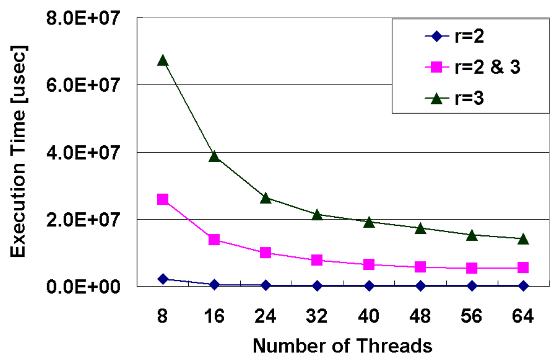
Figure 6 compares performance of the proposed method with three state-of-the-art parallel runtime systems, i.e., OpenMP, Charm++ and Cilk. The input junction tree was a junction tree with 1024 cliques, each consisting of 15 binary variables. The average number of successors for the cliques was 2 and the separator width was 4. We observed similar results using other junction trees. In Figure 6(a), all the methods exhibited scalability, although Charm++ implementation showed relatively large overhead due to its message passing based synchronization mechanism. Charm++ had similar execution time when using one or two threads, since no message passing was involved for the scenario with one thread. For Cilk, we generated the DAG offline and then dynamically spawned a thread for each task. The runtime system of Cilk scheduled the threads. This ensures that Cilk used the same DAG for scheduling. Cilk implementation



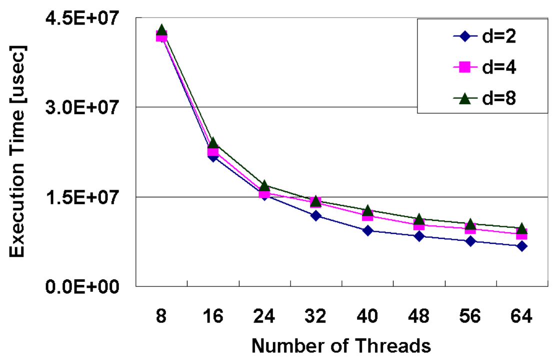
(a) Execution time with respect to number of cliques



(b) Execution time with respect to clique width



(c) Execution time with respect to number of states of random variables



(d) Execution time with respect to number of successors for each clique

Figure 8. Scalability of the proposed method with respect to various parameters of junction trees.

scaled well, since the randomization technique used for work stealing in Cilk reduces synchronization overhead. The absolute execution time of Cilk implementation was higher compared with the proposed method. In Figure 6(b), the proposed method still showed the best performance. The OpenMP based implementation exhibited limited scalability, since it mainly parallelized the loops in node level primitives (see [10]) during exact inference. When 64 threads were used, the speedup of the proposed method is 13.7 and 9.4 compared with the OpenMP and Charm++ based implementations, respectively.

We compared the proposed method with two typical schedulers for exact inference, i.e., a centralized scheduler and a distributed scheduler, as discussed in Section V-C. We used the same junction tree as in the previous experiment, but the clique width was set as 8 and 15 for Figure 7(a) and (b), respectively, to show the impact of task sizes. Thus, the size of each potential table (POT) in Figure 7(b) was 128 times as large as that in Figure 7(a). The results were *inconsistent* for the two baseline methods. `Cent ded` achieved better performance than `Dist shared` for junction tree with large POTs, but poor performance for junction tree with small POTs. An explanation to this observation is that large POTs provides more parallelism, but `Dist shared` dedicates many threads to scheduling. Thus, the number of threads for processing POTs was limited. When scheduling small POTs, many threads completed the assigned POTs quickly, but the single scheduler of `Cent ded` was too busy to process the completed POTs and allocate new POTs in time. The proposed method dynamically adjusted the number of threads for scheduling or execution. Thus, it achieved superior performance.

We modified the parameters of the junction tree used in the previous two experiments to observe the performance of our method in various situations. We varied the number of cliques N, clique width W_C, number of states of the random variables r and the average number of children d for the cliques. For each set of parameters, we repeated the experiment five times. The average of the results is shown in Figure 8. The standard deviation was within 5% of the execution time. In Figure 8(a), the execution time was propotional to the number of cliques N. Regardless of N, the execution time scaled well for all the junction trees. In Figure 8(b), the execution time with respect to $W_C = 15$ was much higher than the other two situations, since the potential table size increases exponentially with respect to W_C. In

Figure 8(c), we reduced W_C to 10 to evaluate the impact of r, since the junction tree with 1024 cliques is too large to be stored when $r = 3$ and $W_C = 15$. In this figure, the label $r = 2\&3$ corresponds to the junction tree where half of the random variables were binary and the others are trinary. As shown in Figure 8(d), the impact of d on execution time was minor. Based on the observations in Figure 8, we conclude that the proposed method scales well in a large range for junction trees having various parameters.

Finally, we investigated the efficiency of the proposed scheduler for exact inference on manycore processors. We measured the execution time of each thread to check if the workload was evenly distributed, and normalized the execution time of each thread for the sake of comparison. We also limited the number of available cores in this experiment to observe the load balance in various scenarios. As the number of cores increases, although there was a minor imbalance across the threads, the percentage of the imbalanced work was very small compared with the total execution time. In Figure 9, we show one of the results where we used a junction tree with $N = 1024, W_C = 15, r = 2$ and $d = 4$. We had consistent results for the junction trees given in Section V-B.

VI. Conclusion

In this paper, we proposed a self-adaptive algorithm for evidence propagation on manycore processors. We developed a self-adaptive scheduler to allocate the task dependency graph for evidence propagation to the cores on general-purpose manycore processors. Evidence propagation is a typical DAG structured computation. Unlike scheduling DAG structured computations on multicore processors, a centralized scheduler on manycore processors can become a bottleneck in assigning tasks to the execution threads in time; a completely distributed scheduler can lead to high synchronization overheads. The proposed scheduler dynamically adjusts the number of managers for scheduling and number of workers for task execution to improve the overall performance. To the best of our knowledge, no existing scheduler addresses such an issue for exact inference or any other DAG structured computations. The experimental results show the efficiency of the proposed method. As part of our future work, we intend to investigate exact inference on various types of manycore processor, for example, GPGPU. Since most of the GPGPUs can perform a single kernel function only at a time, we will explore how to process multiple tasks by a single kernel function without incurring branch overhead.

Acknowledgment

This research was partially supported by the National Science Foundation under grant number CCF-702784. NSF equipment grant CNS-0454407 is gratefully acknowledged.

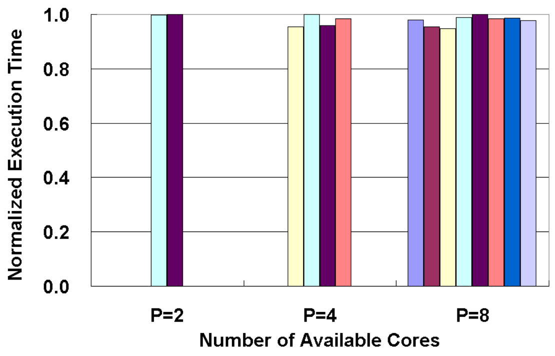


Figure 9. Load balance with respect to number of available cores.

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, December 2002. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20&path=ASIN/0137903952

[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. [Online]. Available: citeseer.ist.psu.edu/segal01rich.html

[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] D. Sheahan, "Developing and tuning applications on UltraSPARC T1 chip multithreading systems," Tech. Rep., 2007.

[7] G. Tan, V. C. Sreedhar, and G. R. Gao, "Analysis and performance results of computing betwenness centrality on ibm cyclops64," *Journa of Supercomputing*, 2009.

[8] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors," in *Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks*, 1996, pp. 207–213.

[9] A. V. Kozlov and J. P. Singh, "A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference," in *Supercomputing*, 1994, pp. 320–329. [Online]. Available: citeseer.ist.psu.edu/kozlov94parallel.html

[10] 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*, October 2007.

[11] ——, “Parallel exact inference on the cell broadband engine processor,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008, pp. 1–12.

[12] H. Jeon, Y. Xia, and V. K. Prasanna, “Parallel exact inference on a CPU-GPGPU heterogeneous system,” in *International Conference on Parallel Processing (ICPP)*, 2010, pp. 1–10.

[13] I. Ahmad, S. Ranka, and S. Khan, “Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy,” in *Intl. Sym. on Parallel Dist. Proc.*, 2008, pp. 1–6.

[14] A. Benoit, M. Hakem, and Y. Robert, “Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems,” *Parallel Computing*, vol. 35, no. 2, pp. 83–108, 2009.

[15] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[16] F. Song, A. YarKhan, and J. Dongarra, “Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems,” in *International Conference for Hight Performance Computing, Networking Storage and Analysis*, 2009.

[17] H. Zhao and R. Sakellariou, “Scheduling multiple DAGs onto heterogeneous systems,” in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2006, pp. 1–12.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” Cambridge, Tech. Rep., 1996.

[19] Intel Threading Building Blocks, “http://www.threadingbuldingblocks.org/.”

[20] OpenMP Application Programming Interface, “http://www.openmp.org/.”

[21] Charm++ programming system, “http://charm.cs.uiuc.edu/research/charm/.”

[22] B. N. T. Kevin Murphy, “http://www.cs.ubc.ca/∼murphyk/software/bnt/bnt.html.”

[23] C. Shi, “Floating-point to fixed-point conversion,” Ph.D. dissertation, Berkeley, CA, USA, 2004, chair-Brodersen, Robert W.

[24] D. Bader, “High-performance algorithm engineering for large-scale graph problems and computational biology,” in *4th International Workshop on Efficient and Experimental Algorithms*, 2005, pp. 16–21.

[25] M. Vikram, A. Baczewski, B. Shanker, and S. Aluru, “Parallel accelerated Cartesian expansions for particle dynamics simulations,” in *Intl. Sym. on Parallel Dist. Proc.*, 2009, pp. 1–6.

[26] K. Liu, J. Chen, Y. Yang, and H. Jin, “A throughput maximization strategy for scheduling transaction-intensive workflows on swindew-g,” *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 15, pp. 1807–1820, 2008.

[27] E. S. Santos, J. T. Wilkinson, and E. E. Santos, “Bayesian knowledge fusion,” in *FLAIRS Conference*, 2009.

[28] K. Madduri, D. Bader, J. Berry, J. Crobak, and B. Hendrickson, “Multithreaded algorithms for processing massive graphs,” in *Petascale Computing: Algorithms and Applications*, 2008, ch. 12, pp. 237–262.