Exploring Weak Dependencies in DAG Scheduling

Nam Ma
Computer Science Department
University of Southern California
Los Angeles, CA 90089
Email: namma@usc.edu

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

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

***Abstract*—Many computational solutions can be expressed as directed acyclic graphs (DAGs) with weighted nodes. In parallel computing, a fundamental challenge is to efficiently map computing resources to the tasks, while preserving the precedence constraints among the tasks. Traditionally, such constraints are preserved by starting a task after all its preceding tasks are completed. However, for a class of DAG structured computations, a task can be partially executed with respect to each preceding task. We define such relationship between the tasks as weak dependency. In this paper, we adapt a traditional DAG scheduling scheme to exploit weak dependencies in a DAG. We perform experiments to study the impact of weak dependency based scheduling method on the execution time using a representative set of task graphs for exact inference in junction trees. For a class of task graphs, on a state-of-the-art general-purpose multicore system, the weak dependency based scheduler runs 4x faster than a baseline scheduler that is based on the traditional scheduling method.**

I. Introduction

General-purpose multicore processors such as the AMD Opteron and Intel Xeon have been prevalent in servers, workstations, and even personal computers. Parallel programs executed on those platforms can be represented as a task dependency graph, i.e., a *directed acyclic graph* (DAG) with weighted nodes. Nodes represent code segments. An edge from node v to node $\tilde{v}$ denotes that the output from the code segment at v is an input to the code segment at $\tilde{v}$. The weight of a node represents the (estimated) execution time of the corresponding code segment. Computations that can be represented as task dependency graphs are called *DAG structured computations* [1], [2].

Performing a task $\mathcal{T}$ in a DAG requires computations on (1) data Y associated with $\mathcal{T}$ and (2) data $X_i, 0 \leq i < d$, where d is the number of immediate preceding tasks of $\mathcal{T}$, and each X_i corresponds to the result of an immediate preceding task of $\mathcal{T}$. The dependency between $\mathcal{T}$ and its immediate preceding tasks is called *weak dependency*, if the computations at $\mathcal{T}$ satisfy the following properties:

1) $\mathcal{T}$ completes when Y has been updated with respect to all $X_i, 0 \leq i < d$.
2) Updating Y with any X_i is independent of the data or the computations performed using $X_j, 0 \leq j < d$ and $j \neq i$.

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.

Note that the order of updating Y using X_i can be arbitrary. Also, in some computations, Y can be updated concurrently by more than one X_i. Our definition of weak dependency allows data Y to be updated by any X_i as soon as X_i becomes available even if $X_j, j \neq i$ is not available.

Prior work has shown that the overall execution time of a program is sensitive to scheduling [3], [4], [5]. Efficient scheduling on multicore processors requires balanced workload allocation and minimum scheduling overhead [6]. It remains a fundamental challenge in parallel computing to dynamically schedule tasks in a task dependency graph. The traditional approach for scheduling tasks in a DAG is to process a task after all its preceding tasks complete, so as to preserve precedence constraints among tasks. However, as observed above, weak dependency allows a task to start execution when any of its preceding tasks completes. A typical example of DAG structured computations with weak dependency is exact inference in junction trees [7].

Our contributions in this paper include:

- We discuss the impact of weak dependency on scheduling DAG structured computations on parallel computing platforms. We show that a scheduling method exploiting weak dependency can explore more parallelism and can reduce the overall execution time.
- We propose a scheduler that can take advantage of weak dependency. We discuss the scheduler in a modular fashion, so that it can be easily adapted to various DAG structured computations by using plug-and-play modules specific to an application.
- We implement the proposed scheduling method on a general-purpose multicore platform. We use exact inference, a classic AI technique, to evaluate the scheduling method discussed in this paper. We show that the proposed scheduler outperforms the baseline for various task graphs, especially for those having limited structural parallelism and tasks with large indegree.

The rest of the paper is organized as follows. In Section II, we provide the background and related work. Section III introduces an example of DAG structured computations that can employ the weak dependency based scheduler. In Section IV, we present our proposed scheduling method. Experimental setup and results are discussed in Sections V and VI respectively. Section VII concludes the paper.

II. Background and Related Work

The input to task scheduling is a directed acyclic graph (DAG), where each node represents a task and each edge corresponds to precedence constraints among the tasks. Each task in the DAG is associated with a *weight*, which is the estimated execution time of the task. Traditionally, a task can start execution only if *all* of its predecessors have been completed [8], [9]. The task scheduling problem is to map the tasks to the threads in order to minimize the overall execution time on parallel computing systems. Task scheduling is in general an *NP-complete* problem [10], [11]. We consider scheduling an arbitrary DAG with given task weights and perform the mapping and scheduling of tasks on-the-fly. The goal of such dynamic scheduling includes not only the minimization of the overall execution time, but also the minimization of the scheduling overhead [2].

The scheduling problem has been extensively studied for several decades [1], [12], [2], [13], [14]. Early algorithms optimized scheduling with respect to the specific structure of task dependency graphs [15], such as a tree or a fork-join graph. In general, however, programs come in a variety of structures [2]. Karamcheti and Chien studied hierarchical load balancing framework for multithreaded computations for employing various scheduling policies for a system [16]. Recent 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, and [1] where Ahmad proposed a game theory based scheduler on multicore processors for minimizing energy consumption. Dongarra *et al.* proposed dynamic schedulers optimized for a class of linear algebra problems on general-purpose multicore processors [14]. Scheduling techniques have been proposed by several emerging programming systems such as Cilk [18], Intel Threading Building Blocks (TBB) [19], OpenMP [20], Charm++ [21] and MPI micro-tasking [22], etc. All these systems rely on a set of extensions to common imperative programming languages, and involve a compilation stage and runtime libraries. These systems are not optimized specifically for scheduling DAGs on manycore processors. For example, Dongarra *et al.* showed that Cilk is not efficient for scheduling workloads in dense linear algebra problems on multicore platforms [23]. In contrast with these systems, we focus on scheduling tasks in DAGs with weak dependencies on general purpose multicore processors.

III. A Motivating Example

A. Exact Inference

A *Bayesian network* is a probabilistic graphical model that exploits conditional independence to represent compactly a joint probability distribution. The most popular exact inference algorithm proposed by Lauritzen and Spiegelhalter [7] converts Bayesian network into a junction tree, then performs exact inference in the 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* (POT). The POT 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.

Exact inference in a junction tree requires updating the junction tree by propagating evidence at some cliques to all the other cliques. Mathematically, evidence propagation from clique $\mathcal{Y}$ to clique $\mathcal{X}$ can be represented as [7]:

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

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

Equation 1 implies three types of node level primitives: *Multiplication*(i.e. $\psi_{\mathcal{X}} \frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}}$) and *Division*(i.e. $\frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}}$) between two POTs and *Marginalization*(i.e. $\sum_{\mathcal{Y}\setminus\mathcal{S}} \psi^*_{\mathcal{Y}}$) that obtain the separator POT using clique POT [24]. Hereafter, we use updating a clique or a separator as a brief of updating a clique POT or separator POT.

A junction tree is updated using the above evidence propagation in two passes: *evidence collection* and *evidence distribution*. In evidence collection, evidence is propagated from the leaves to the root of the junction tree. A clique $\mathcal{C}$ is ready to propagate evidence to its parent when $\mathcal{C}$ has been fully updated from all its children. Evidence distribution is the same as evidence collection, except that the evidence propagation direction is from the root to the leaves. In this paper, we only consider evidence collection where a clique requires multiple updates from its children. Note that a clique can be updated from its children separately in any order. Thus, there exist weak dependencies between a clique and its children in evidence collection.

B. Impact of Weak Dependency

Consider evidence collection in a junction tree with 9 cliques as given in Figure 1. Note that in evidence collection, the evidence is propagated from the leaves $(C_1, C_3, C_4, C_6, C_7, C_8)$ to the root (C_0). Nodes marked with red boxes represent the being-updated cliques. Nodes marked with dark shadow represent the already-updated cliques.

Using the traditional scheduling methods, evidence collection in the junction tree proceeds as illustrated in Figure1(a). Accordingly, clique C_2 needs to wait until C_4, C_5, and C_6 complete execution. Then, C_2 is updated sequentially from inputs C_4, C_5, and C_6. Similarly, C_0 needs to wait for and then updated from C_1, C_2, C_3.

Using the weak dependency based scheduling method, evidence collection in junction tree proceeds as illustrated in Figure1(b). Accordingly, C_2 does not have to wait until C_4, C_5, and C_6 complete; it can start to be updated right after any of them completes execution. Similarly, C_0 can start to be updated when any of C_1, C_2, and C_3 completes. Thus,

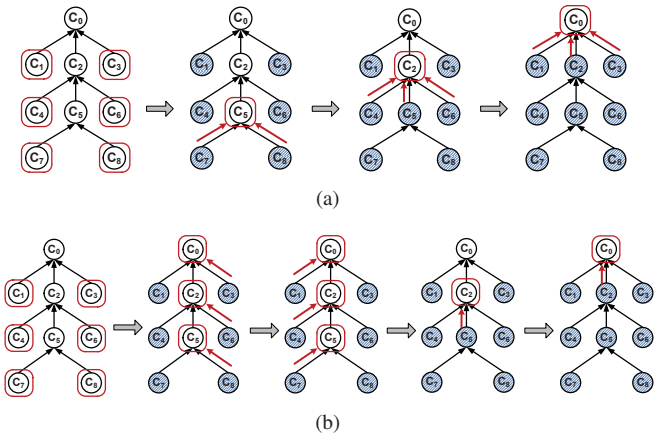


Fig. 1. Evidence collection using (a) the traditional scheduling method, (b) the weak dependency based scheduling method.

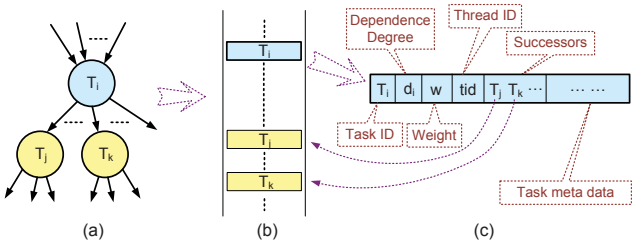


Fig. 2. (a) A portion of a task dependency graph, (b) The corresponding representation of the global task list (GL), (c) The data of task T_i in the GL.

during the execution, C_0, C_2, C_5 can be updated in parallel using C_3, C_6, C_8 and then C_1, C_4, C_7 respectively.

Assume that updating a leaf clique or updating a clique using another clique takes unit time. On a parallel random access machine (PRAM) with sufficient number of processors, the traditional scheduling method requires unit time to update leaf cliques $\{C_1, C_3, C_4, C_6, C_7, C_8\}$, two units to update C_5 from $\{C_7, C_8\}$, three units to update C_2 from $\{C_4, C_5, C_6\}$, and three units to update C_0 from $\{C_1, C_2, C_3\}$, resulting in nine units of time in total; the weak dependency based scheduling method requires unit time to update $\{C_1, C_3, C_4, C_6, C_7, C_8\}$, a unit to update $\{C_0, C_2, C_5\}$ from $\{C_3, C_6, C_8\}$, a unit to update $\{C_0, C_2, C_5\}$ from $\{C_1, C_4, C_7\}$, a unit to update C_2 from C_5, and a unit to update C_0 from C_2, resulting in five units of time in total.

IV. DAG Scheduling with Weak Dependencies

A. Components of The Scheduler

We exploit weak dependencies in DAGs using a straightforward scheduling scheme that consists of a single *task allocator* allocating ready tasks to threads for execution. Each thread is bound to a core and persists over the entire program execution. Locks are used to protect shared data. Detailed modules for scheduling are discussed as the following:

The input task dependency graph is represented by a list called the *global task list* (GL), accessible by all the threads. Figure 2 shows a portion of the task dependency graph, the corresponding part of the GL, and the data of element T_i in the GL. Each element in the GL consists of task ID, dependency degree, task weight, successors, thread ID, and the task meta data (e.g. application specific parameters). 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. The *task weight* is the estimated execution time of the task. We keep the *successors* that are the pointers to the elements of succeeding tasks along with each task. The *thread ID* shows the thread in which the task is executing. The thread ID is used for allocating task copies as described below. In each element, there is *task meta data*, such as the task type and pointers to the data buffer of the task, etc.

Each thread has a *local task list* (LL) for tasks that are ready to be processed. However, to exploit weak dependencies in a task graph, each element in the LL is not the real task as in the GL but a *copy* of the task. Each copy of a task consists of a pointer to the corresponding task in the GL and a pointer to the preceding task in the GL. The number of copies of a task is equal to the number of its incoming edges (node degree). When a copy of a task completes execution, the dependency degree of the corresponding task in the GL is decreased by 1. Thus, when all the copies of a task finish execution, its dependency degree becomes zero, and its successors become ready to execute. Then, a copy of each successor of the task is created and inserted into one of the LLs. Each LL has a *weight counter* associated with it to record the total workload of the tasks currently in the LL. Once the first copy of a task is inserted into an LL or the last copy of a task is fetched from an LL, the weight counter is updated.

The *Allocate module* of the scheduler creates a copy of a task and inserts it into one of the LLs. The *Completed Task ID buffer* of the scheduler stores the IDs of tasks that have completed execution. Initially, all the Completed Task ID buffers are empty. For each task in the Completed Task ID buffer, the Allocate module creates a copy for each of its successors and insert that copy into an LL. The preceding task in each copy will be updated by that task. Since all the copies of a task need to be processed one after another, we assigned all of them to the same thread for data locality. Accordingly, when the first copy of a task is assigned to a thread, the thread ID will be updated in the corresponding task. The later copies of the task will be assigned to the thread based on the available thread ID. Heuristics can be used for task allocation to balance the workload across the threads. In this paper, we allocate the first copy of a task to the thread with the smallest workload at the time of completion of the task execution.

Each thread has a *Fetch module* that takes a copy of a task from its LL and sends the task-copy to the *Execute module* in the same thread for execution. Heuristics can be used by the Fetch module to select elements from the LL. For example, tasks with more children can have higher priority for execution [2]. In this paper, we use a straightforward method, where the element at the head of the LL is selected by the Fetch module. Once a task-copy is fetched for execution, the dependency degree of the corresponding task is decremented.

Once the dependency degree becomes 0, i.e. the last copy of the task has been processed, the Fetch module sends the ID of the task to the Completed Task ID buffer, so that the Allocate module can accordingly create and schedule copies of the successors of the task.

B. A Weak Dependency Based Scheduler

Based on the framework of scheduling in Section IV-A, we present a sample implementation of the weak dependency based scheduling method in Algorithm 1. The notations used in the algorithm are given as follows. N denotes the total number of tasks in GL. LL_t denotes the local task list in Thread t, $0 \leq t < P$. tid_T, d_T and w_T denote the thread ID, dependency degree and the weight of task T, respectively. T_i^j denotes a copy of task T_i that will be executed based on the preceding task T_j. W_t is the weight counter of Thread t, i.e. the total weight (estimated execution time) of the tasks currently in LL_t. $\mathcal{C}_i$ denotes the clique corresponding to task T_i.

Algorithm 1 An Implementation of Weak Dependency Based Scheduling for Evidence Collection in Junction Tree

```
    {Initialization}
 1: let tid_i = NIL, 0 ≤ i < N;
 2: let S = {T_i | d_i = 0}, 0 ≤ i < N;
 3: evenly distribute a task-copy T_i^NIL of each T_i in S to LL_t, 0 ≤ t < P;
    {Scheduling}
 4: for Thread t (t = 0 ··· P − 1) pardo
 5:   while GL ≠ φ or LL_t ≠ φ do
        {Fetch module}
 6:     fetch a task-copy T_i^p from the head of LL_t;
 7:     d_{T_i} = d_{T_i} − 1;
 8:     if d_{T_i} = 0 then
 9:       remove task T_i from GL;
10:       add the ID of T_i to the Completed Task ID buffer;
11:       W_t = W_t − w_{T_i};
12:     end if
        {Execute module}
13:     execute T_i^p: updating C_i from C_p according to Eq. 1;
        {Allocate module at Thread 0}
14:     if t = 0 then
15:       for all T_i ∈ Completed Task ID buffer do
16:         for all T_s ∈ {successors of T_i} do
17:           create a task-copy T_s^i of T_s;
18:           if tid_{T_s} = NIL then
19:             let j = arg min_{t=1···P}(W_t);
20:             append T_s^i to LL_j;
21:             let tid_{T_s} = j;
22:             W_j = W_j + w_{T_s};
23:           else
24:             append T_s^i to LL_{tid_{T_s}};
25:           end if
26:         end for
27:       end for
28:       Completed Task ID buffer = φ;
29:     end if
30:   end while
31: end for
```

Lines 1-3 in Algorithm 1 initialize the thread ID of the tasks and the local task lists. The thread ID of a task is initialized as NIL, and will be updated when the first copy of the task is allocated. The later copies of the task will be allocated to the same thread. In Lines 4-31, the algorithm performs task scheduling iteratively until all the tasks are processed. Lines 6-12 correspond to the Fetch module, where it fetches a task-copy from the head of the local task list and decrements the dependency degree of the corresponding task (also the number of unprocessed copies of the task). Lines 9-11 add the ID of the fetched task to the Completed Task ID buffer and updates W_t according to the fetched task when the last copy of the task is processed. Lines 14-29 correspond to the Allocate module, which creates copies of the tasks in the Completed Task ID buffer (Line 17), then allocates those copies into a target thread (Line 20 or 24). Only thread 0 is responsible for allocating tasks to all the threads (Line 14). We choose the target thread for the first copy of a task as the one with the smallest workload (Line 21), although alternate heuristics can be used.

Line 13 corresponds to the Execute module, where the fetched task-copy T_i^p is executed. We include the execution of the evidence collection as an illustrated application. In the execution of T_i^p for evidence collection, clique $\mathcal{C}_i$ is updated from its preceding clique $\mathcal{C}_p$ according to Equation 1. Other DAG structured computations can be easily plugged in.

V. Experimental Setup

A. Platforms

We conducted experiments on an 8-core AMD Opteron based system. The system consists of the Dual AMD Opteron 2350 (Barcelona) Quad-Core processors running at 2.0 GHz, and 16 GB DDR2 memory shared by all the cores. The operating system was CentOS version 5 Linux. We used the GCC 4.1.2 compiler on this platform. Since AMD Opteron does not support more than one hardware thread per core, we mostly conducted experiments using one thread in each core.

B. Baseline Implementation

So as to compare with the proposed scheduler, we implemented a baseline scheduler using the traditional scheduling approach. The baseline scheduler is similar to the proposed scheduler, except for handling the precedence constraints in the DAG. The baseline scheduler allowed a task to be executed only when all its preceding tasks completed. Thus, there is no need for task copy. The local task lists contained the elements of task data as in the global task list. The Allocate module must keep track of the number of completed preceding tasks of a task. When a task completed execution, the Allocate module decremented the dependency degree of each successor of that task. When the dependency degree of a task became 0, the Allocate module assigned the task to the LL with lightest workload and updated the corresponding weight counter. The Fetch module fetched a task from the LL, updated the weight counter, and placed the ID of the task into the Completed Task ID buffer. In the Execute module, a task updated its clique from all its preceding cliques sequentially. Spinlocks were used for the ready task lists and Completed Task ID buffer.

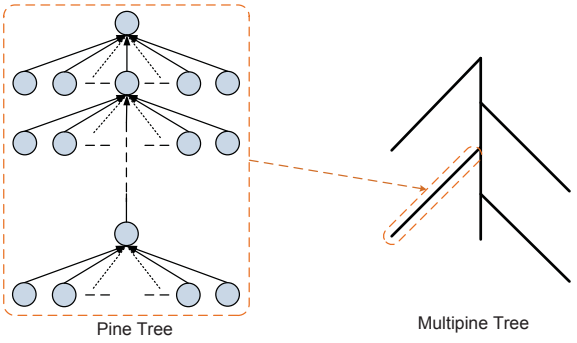


Fig. 3. A Pine Tree and a Multipine Tree.

C. Datasets

We evaluated our proposed scheduling method using evidence collection for various junction trees. They can be grouped into three types of trees: Pine Tree, Arbitrary Tree, and Balanced Tree.

- A *Pine Tree*, as illustrated in Figure 3, was formed by first creating a chain of cliques, then each clique on this chain was connected to $(d-1)$ cliques. Thus, nodes on the principal chain have the same node degree (d). As shown in Section III, this type of graph can significantly limit the performance of the baseline scheduler since the root node has to wait for a long time for the cliques on the principal chain to complete. In contrast, the weak dependency based scheduler can still execute cliques on the principal chain in parallel. A *Multipine Tree* was formed by connecting b Pine Trees together to obtain more parallelism. One Pine Tree was used as the main branch, in which some leaf nodes became the roots of $(b-1)$ remaining Pine Trees, distributed evenly on the main branch.
- An *Arbitrary Tree* was generated based on the parametric random graph described in [13]. Parameters used to specify an arbitrary tree include: (1) number of nodes (N), (2) maximum node degree (D_m), (3) maximum tree height (H_m). A generated tree had an average node degree (d) and tree height (H). If H is high, the tree may have a very low average node degree, resulting in the limited amount of parallelism. Depending on D_m, the tree could be more balanced (with low D_m) or unbalanced (with high D_m).
- A *Balanced Tree* was generated as a traditional balanced tree: every node, except leaf nodes, had an identical node degree (d). Balanced Trees offered sufficient parallelism to achieve high speedup. A high node degree may limit the performance of the baseline. Binary Balanced Tree, with the lowest node degree $(d=2)$, is considered as the best case for the baseline scheduler.

The specific parameters of the junction trees, including the clique size, are given in Section VI. In all the experiments, the random variables were set as binary, i.e. $r=2$.

D. Performance metrics

We used execution time and speedup to present our experimental results. We calculated the speedup on p cores of each scheduler as the ratio of the execution time on 1 core to the execution time on p cores.

VI. Evaluation Results

A. Illustration of the Execution Time of Tasks

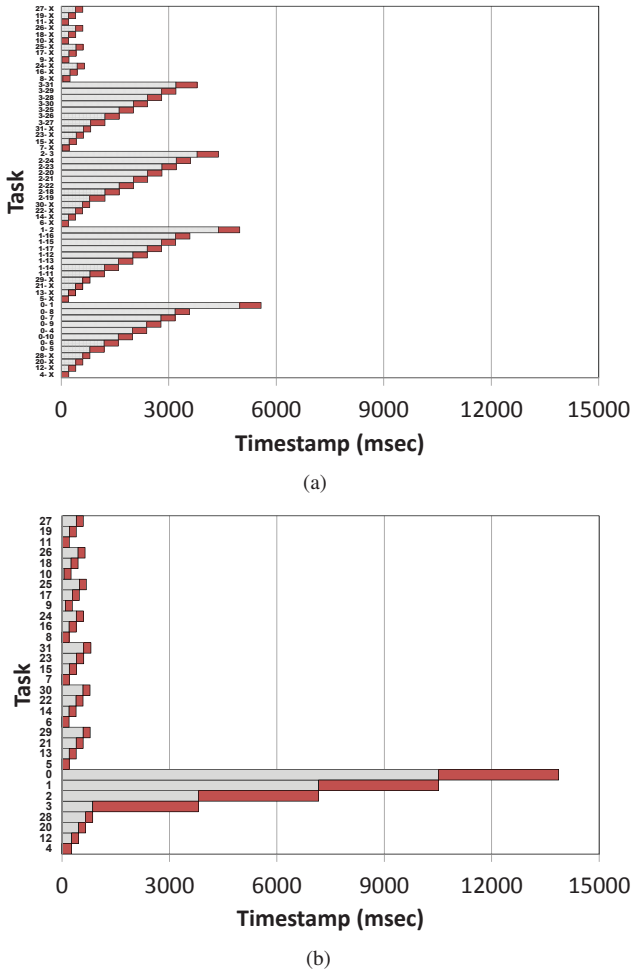


Fig. 4. Execution timestamps on 8 cores for a Pine Tree with $N=32$ nodes and node degree $d=8$ using (a) the proposed scheduler and (b) the baseline.

For the sake of illustrating how the weak dependency based scheduling method improved the performance of evidence collection, we recorded the execution timestamp of all the tasks for a small Pine Tree that had $N=32$ cliques and node degree $d=8$. Clique size was set at 18, resulting in the size of the POT of each clique to be 2^{18} (Section III-A). We used 8 cores with one thread in each core. The results of the evidence collection in this junction tree are given in Figure 4. Each entry on the vertical axis represents a task (TaskID) for the baseline, and a task-copy (TaskID - Preceding TaskID) for the proposed scheduler.

We observe that 28 leaf nodes of the tree were distributed evenly to 8 cores in both of the schedulers. However, as

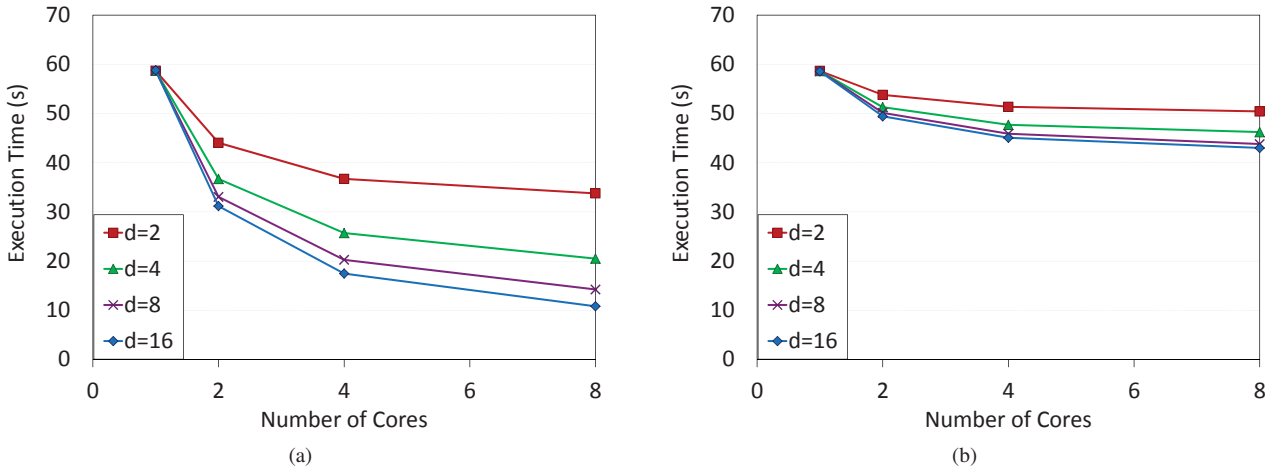


Fig. 5. Execution time for a Pine Tree with $N = 1024$ nodes and various node degree d using (a) the proposed scheduler, and (b) the baseline.

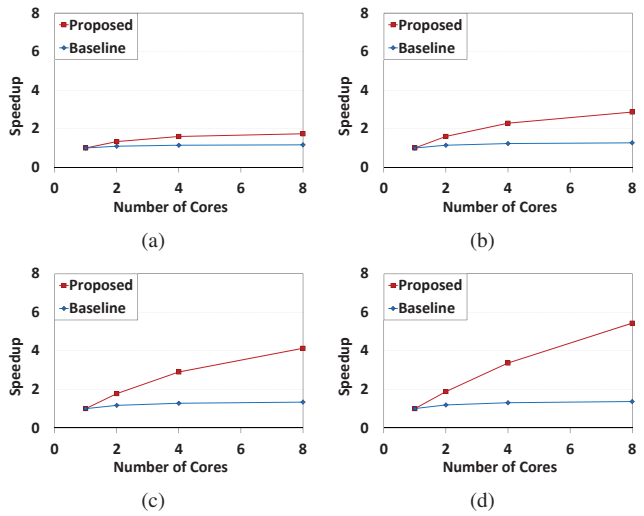


Fig. 6. Speedups of the two schedulers for a Pine Tree with $N = 1024$ nodes and node degree (a) $d = 2$, (b) $d = 4$, (c) $d = 8$, (d) $d = 16$.

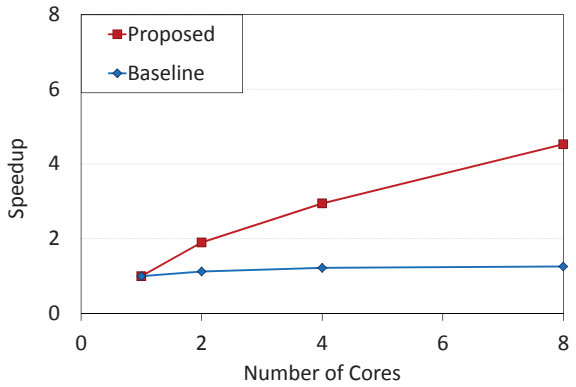


Fig. 7. Speedup of the two schedulers using two threads per core for a Pine Tree with $N = 1024$ nodes and node degree $d = 16$.

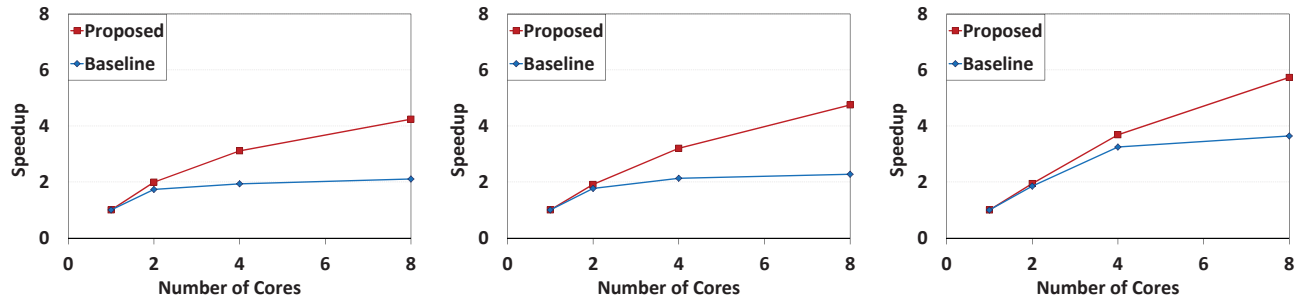
expected, the two schedulers had different behaviors with the 4 nodes of the principal path. In the baseline, these nodes were executed one after another; each node was processed from the preceding nodes sequentially. In the proposed scheduler, these 4 nodes were processed in parallel; each from a corresponding preceding node. Thus, the amount of parallelism increased, and the execution time was significantly reduced.

B. Pine Tree

Pine Tree is considered as the special case where the baseline performs worst comparing with the proposed scheduler. We used Pine Trees consisting of $N = 1024$ cliques, with various node degrees (d) to observe the impact of node degree on the performance of the two schedulers. Clique size was set at 15. We used one thread in each core. The results including the execution time and the speedup for various numbers of cores are given in Figures 5 and 6, respectively. For any d, the proposed scheduler ran much faster than the baseline. When d increased, the speedup of the proposed scheduler was significantly improved while speedup of the baseline just slightly increased. We observed that when $d = 16$, using 8 cores, the speedup of the proposed scheduler was about 4 times higher than that of the baseline (5.43 vs. 1.37). This speedup can be higher by using more processors or by increasing the node degree.

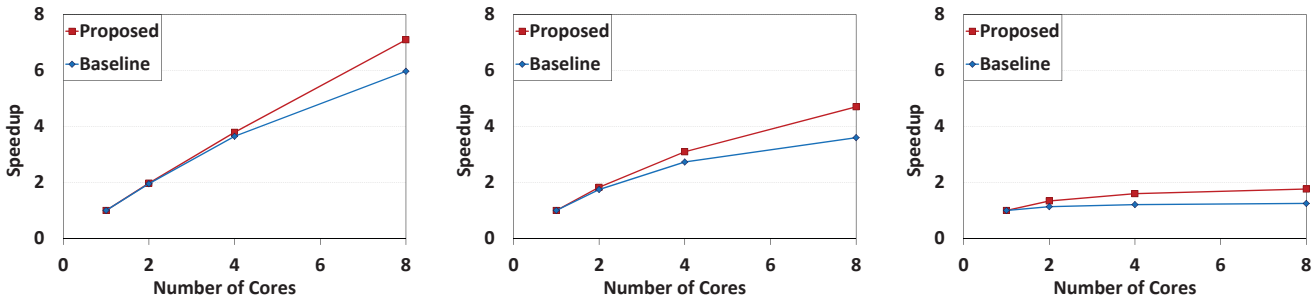
We also conducted our experiments using more than one thread on each core. Figure 7 shows the speedup of the two schedulers using two threads per core for a Pine Tree with $N = 1024$ and $d = 16$, the same dataset used in Figure 6(d). The proposed scheduler still achieved far higher speedup than baseline. However, compared with the results using one thread per core given in Figure 6(d), the speedup achieved using two threads per core was slightly decreased. Note that the allocator is centralized. Thus we incur higher overheads as the number of threads increases. In all the experiments presented hereafter, we used one thread per core.

As the generalization of the Pine Tree, a Multipine Tree was created by connecting multiple branches of Pine Tree. The more number of Pine Tree branches, the higher the amount of parallelism was provided in the input junction tree. Therefore, the speedups of the two schedulers increased with the number of branches b, as shown in Figure 8. However, the speedup of



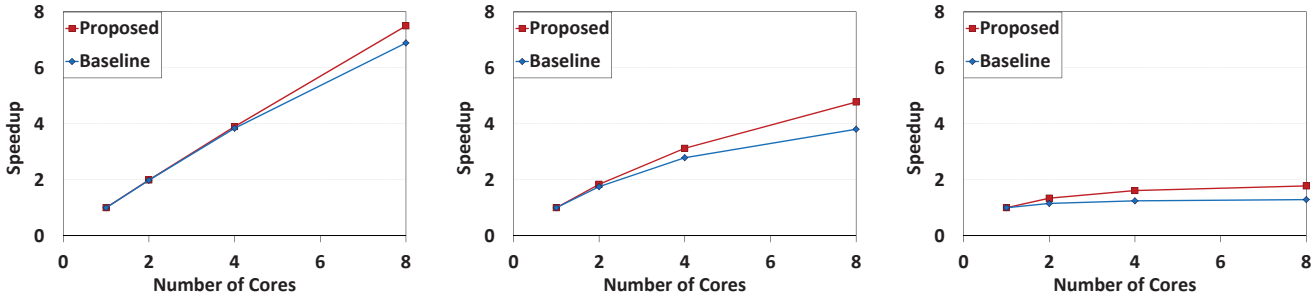
(a) (b) (c)

Fig. 8. Speedups of the two schedulers for a Multipine Tree with $N = 1024$ nodes and node degree $d = 4$ and the number of branches (a) $b = 2$, (b) $b = 4$, (c) $b = 8$.



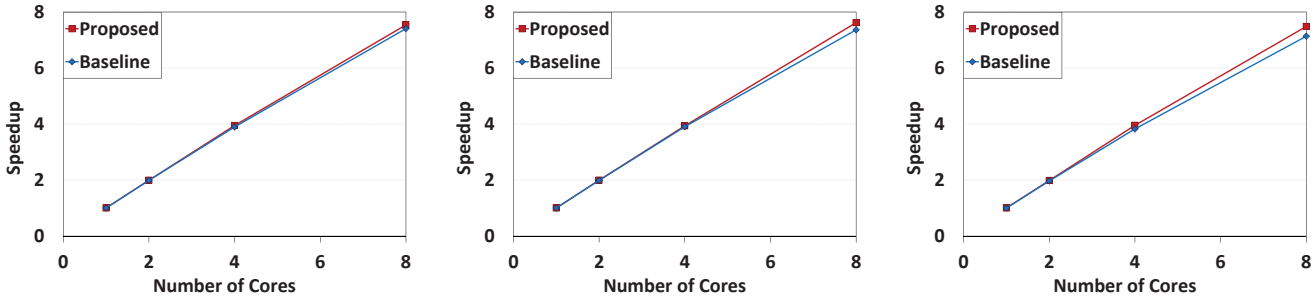
(a) (b) (c)

Fig. 9. Speedups of the two schedulers for arbitrary trees with $N = 1024$ nodes, maximum node degree $D_m = 16$, and tree height (a) $H = 10$, (b) $H = 100$, (c) $H = 500$.



(a) (b) (c)

Fig. 10. Speedups of the two schedulers for arbitrary trees with $N = 1024$ nodes, maximum node degree $D_m = 6$, and tree height (a) $H = 10$, (b) $H = 100$, (c) $H = 500$.



(a) (b) (c)

Fig. 11. Speedups of the two schedulers for balanced trees with $N = 1024$ nodes and node degree (a) $d = 2$, (a) $d = 4$, (a) $d = 8$.

the proposed scheduler was still significantly higher than the baseline.

C. Arbitrary Tree

We generated arbitrary trees of 1024 cliques using the maximum node degrees $D_m = 16$ (less balanced trees) and $D_m = 6$ (more balanced trees). For each D_m, we set the tree height $H = 10$ (very "bushy" tree), $H = 100$, and $H = 500$ (very slim tree). Clique size was set randomly between 14 to 16. The speedups of the two schedulers are given in Figures 9 and 10. As expected, the two schedulers achieved high speedups when $H = 10$, and low speedups when $H = 500$. In any case, the proposed scheduler consistently outperformed the baseline scheduler with the average improvement of 30%. For lower D_m, the gap between the two schedulers became smaller. This is because with smaller D_m the execution time of the preceding tasks became more identical, and the impact of exploiting weak dependencies became less.

D. Balanced Tree

Balanced trees provide plenty of parallelism for the two schedulers. For balanced trees, the impact of the weak dependency based scheduling method is minimal, since all the preceding tasks of a task finish their execution almost at the same time. The binary balanced tree is the best case for the baseline, where the node degree is low – only 2. Figure 11 shows the speedup of the two schedulers for balanced trees with $N = 1024$ cliques and various node degrees d. Clique size was set at 15in this experiment.

Since the parallelism was sufficient, we achieved almost linear speedup with both schedulers. In addition, we observed that the proposed scheduler continued having better speedups compared with the baseline. Even for the best case of the baseline – the binary balanced tree – the proposed scheduler still slightly outperformed the baseline.

VII. Conclusions

In this paper, we exploited weak dependencies in DAG scheduling to improve the performance of scheduling such computations on multicore processors. We designed and implemented a scheduling method by adapting a traditional scheduling scheme, so as to allow task execution with respect to its preceding tasks separately. We used a typical example called evidence collection in junction trees to evaluate the proposed scheduling method. We showed that the proposed scheduling method significantly improved the performance compared with the baseline for a wide variety of junction trees. For arbitrary trees, on an 8-core multicore system, the proposed scheduler ran faster than the baseline by 30% on the average. For some types of junction trees, the proposed method can run 4x faster than the baseline. We plan to generalize the scheduling method and investigate the method for other DAG structured computations in the future. We would also like to investigate the scalability of the proposed technique, so that it can be utilized on other parallel computing platforms, such as manycore processors and clusters.

References

[1] 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.

[2] 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.

[3] H.-L. Chen and C.-T. King, "Eager scheduling with lazy retry in multiprocessors," *Future Gener. Comput. Syst.*, vol. 17, no. 3, pp. 215–226, 2000.

[4] M. De Vuyst, R. Kumar, and D. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors," in *Intl. Sym. on Parallel Dist. Proc.*, 2006, pp. 1–6.

[5] X. Tang and G. R. Gao, "Automatically partitioning threads for multithreaded architectures," *J. Parallel Distrib. Comput.*, vol. 58, no. 2, pp. 159–189, 1999.

[6] S. Alarm, R. Barrett, J. Kuehn, P. Roth, and J. Vetter, "Characterization of scientific workloads on systems with multi-core processors," in *IEEE International Symposium on Workload Characterization*, 2006, pp. 225–236.

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

[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] O. Sinnen, *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[11] C. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," in *Proceedings of the twentieth annual ACM symposium on Theory of computing*, 1988, pp. 510–513.

[12] 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.

[13] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260–274, March 2002.

[14] 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.

[15] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*. New York, NY: John Wiley and Sons, 1976.

[16] V. Karamcheti and A. Chien, "A hierarchical load-balancing framework for dynamic multithreaded computations," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1998, pp. 1–17.

[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] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "Mpi microtask for programming the cell broadband enginetm processor," *IBM Systems Journal*, vol. 45, no. 1, pp. 85–102, 2006.

[23] J. Kurzak and J. Dongarra, "Fully dynamic scheduler for numerical computing on multicore processors," Tech. Rep., 2009.

[24] 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, pp. 221–228.