

Collaborative Scheduling of DAG Structured Computations on Multicore Processors

Yinglong Xia
Computer Science Department
University of Southern California
Los Angeles, CA 90089, U.S.A.
yinglonx@usc.edu

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

ABSTRACT

Many computational solutions can be expressed as directed acyclic graphs (DAGs), in which the nodes represent tasks to be executed and edges represent precedence constraints among the tasks. A fundamental challenge in parallel computing is to schedule such DAGs onto multicore processors while preserving the precedence constraints. In this paper, we propose a lightweight scheduling method for DAG structured computations on multicore processors. We distribute the scheduling activities across the cores and let the schedulers collaborate with each other to balance the workload. In addition, we develop a lock-free local task list for the scheduler to reduce the scheduling overhead. We experimentally evaluated the proposed method by comparing with various baseline methods on state-of-the-art multicore processors. For a representative set of DAG structured computations from both synthetic and real problems, the proposed scheduler with lock-free local task lists achieved $15.12\times$ average speedup on a platform with four quadcore processors, compared to $8.77\times$ achieved by lock-based baseline methods. The observed overhead of the proposed scheduler was less than 1% of the overall execution time.

Categories and Subject Descriptors

F.1.2 [Modes of Computation]: Parallelism and concurrency; D.4.1 [Process Management]: Scheduling

General Terms

Performance

Keywords

DAG structured computations, collaborative scheduling, task sharing, lock free structures

1. INTRODUCTION

Given a program, we can represent the program as a *directed acyclic graph* (DAG) with weighted nodes, in which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'10, May 17–19, 2010, Bertinoro, Italy.

Copyright 2010 ACM 978-1-4503-0044-5/10/05 ...\$10.00.

the nodes represent code segments, and there is an edge from node v to node \tilde{v} if the output from the code segment performed 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. Such a DAG is called a *task dependency graph*. The computations that can be represented as task dependency graphs are called *DAG structured computations* [2, 15].

The objective of scheduling for DAG structured computations on multicore processors is to minimize the overall execution time by allocating the tasks to the cores, while preserving the precedence constraints [15]. Prior work has shown that system performance is sensitive to scheduling [7, 10, 20]. Efficient scheduling on typical multicore processors such as the AMD Opteron and Intel Xeon require balanced workload allocation and minimum scheduling overhead [3]. The *collaborative scheduling* is a type of work sharing based distributed scheduling methods, where we distribute the scheduling activities across the cores and let the schedulers dynamically collaborate with each other to balance the workload. Collaboration requires locks to prevent concurrent write to shared variables. However, locks can result in significant synchronization overhead. The traditional lock-free data structures are based on hardware specific atomic primitives, such as CAS2, which are not available in some modern systems [9]. Thus, we utilize a software lock-free data structure to improve the performance of the collaborative scheduler.

Our contributions in this paper include: (a) We propose a lightweight collaborative scheduling method for DAG structured computations. (b) We develop lock-free local task lists to reduce coordination overhead during scheduling. (c) We discuss the correctness of the proposed method. (d) We experimentally show the efficiency of the proposed method compared with various baseline methods.

The rest of the paper is organized as follows: In Section 2, we provide the background of task scheduling for DAG structured computations. Section 3 introduces related work on task scheduling and lock-free data structures. In Section 4, we present our collaborative scheduling method. Section 5 presents the lock-free data structure for proposed method. Experimental results are shown in Section 6. Section 7 concludes the paper.

2. BACKGROUND

In our context, the input to task scheduling is a directed acyclic graph (DAG), where each node represents a task and the edges correspond 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. A task can begin execution only if all its predecessors have completed execution [1]. Since we bind a separate thread to each core of a multicore processor in this paper, the task scheduling problem is to map the tasks in a given DAG onto the threads so as to minimize the overall execution time on parallel computing systems. Task scheduling is in general an *NP-complete* problem [11, 18].

Scheduling DAG structured computations can be *static* or *dynamic*. Unlike static scheduling which requires complete and accurate information on the DAG and platform to determine the schedule of tasks before execution begins, dynamic scheduling decides the mapping and scheduling of tasks on-the-fly. In addition, dynamic scheduling can tolerate error in estimated task weights [15]. We focus on dynamic scheduling in this paper. Dynamic scheduling methods can be *centralized* or *distributed*. Centralized scheduling dedicates a thread (or a core) to execute a scheduler, and uses the remaining threads to execute the tasks allocated by the scheduler. Distributed scheduling, however, integrates a scheduler into each thread. These schedulers cooperate with each other to achieve load balance across the threads. We consider distributed scheduling on homogeneous multicore processors where all the cores are identical.

Work sharing [19] and *work stealing* [6] are two generic approaches for task scheduling. In work sharing, when a task becomes ready, it is allocated to a thread to satisfying some objective function, for example, balance the workload across the threads. By workload of a thread, we mean the total weight of the tasks allocated to the thread but not yet been executed. In work stealing, however, every thread allocates new tasks to itself and the underutilized threads attempt to steal tasks from others [19]. Randomization is used for work stealing. This also reduces coordination overhead. When the task weights are not available and the input DAG has sufficient parallelism, work stealing shows encouraging performance [6]. When the task weights are known and the input DAGs are not limited to nested parallelism, work sharing can lead to improved load balance across the threads [19]. The main downside of work sharing approach is that a thread must know others' workload when allocating new tasks. This can increase coordination overhead in lock based implementations. Our proposed method is a type of work sharing scheduler. We use software lock-free data structures to reduce the overhead due to synchronization.

3. RELATED WORK

The scheduling problem has been extensively studied for several decades. Early algorithms optimized scheduling with respect to specific task dependency graphs, such as the fork-join graph [8], albeit general programs come in a variety of structures. Beaumont *et al.* compared a centralized scheduler to a distributed scheduler [5]. The comparison is based on multiple bag-of-task applications, which are different from DAG structured computations. Squillante and Nelson studied work sharing and stealing approaches in task scheduling, and pointed out that task migration in shared memory multiprocessor increased contention for the bus and the shared memory itself [19]. This implies the importance of lock-free data structure for schedulers. Dongarra *et al.* explored dynamic data-driven execution of tasks in dependency graphs with compute-intensive tasks interconnected

with a dense and complex dependencies, such as the tiled Cholesky, the tiled QR and the tiled LU factorizations, where the workload is divisible. In [22], Zhao discussed scheduling techniques for heterogeneous resources, unlike the platforms considered in this paper. Scheduling techniques have been utilized by several programming systems such as Cilk [6], Intel Threading Building Blocks (TBB) [12] and OpenMP [17]. All these systems rely on a set of extensions to common imperative programming languages, and involve a compilation stage and various runtime systems. TBB and Cilk employ work stealing for scheduling; they do not consider task weights. In contrast with these systems, we focus on scheduling methods for DAGs on general purpose multicore processors.

Almost all the existing lock-free data structures are based on atomic primitives. However, some atomic primitives used in these structures require special hardware support, e.g. Double-Word Compare-And-Swap (CAS2), which is not available in modern computer systems [4]. Unlike [16], we propose a software lock-free data structure without using atomic primitives. In our approach, we duplicate/partition shared data to ensure exclusive access to shared variables.

4. COLLABORATIVE SCHEDULING

4.1 Components

We modularize the collaborative scheduling method and show the components in Figure 1, where each thread host a scheduler consisting of several modules. The input task dependency graph is shared to all threads. The task dependency graph is represented by a list called the *global task list* (GL) on the left-hand side of Figure 1, which is accessed by all the threads shown on the right-hand side. Unlike the global task list, the other modules are hosted by each thread.

The *global task list* (GL) represents the given task dependency graph. Figure 2(a) shows a portion of the task dependency graph. Figure 2(b) shows the corresponding part of the GL. As shown in Figure 2(c), each element in the GL consists of task ID, dependency degree, task weight, successors 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. 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 transversing the global task list. In each element, there is *task meta data*, such as the task type and pointers to the data buffer of the task, etc. The GL is shared by all the threads and we use locks to protect dependency degree d_i , $0 \leq i < N$.

Every thread has a *Completed Task ID buffer* and an *Allocate module* (Figure 1). The Completed Task ID buffer in each thread stores the IDs of tasks that are processed by the thread. Initially, all the Completed Task ID buffers are empty. During scheduling, when a thread completes the execution of an assigned task, the ID of the task is inserted into the Completed Task ID buffer in that thread. For each task in the Completed Task ID buffer, the Allocate module decrements the task dependency degree of the successors of the

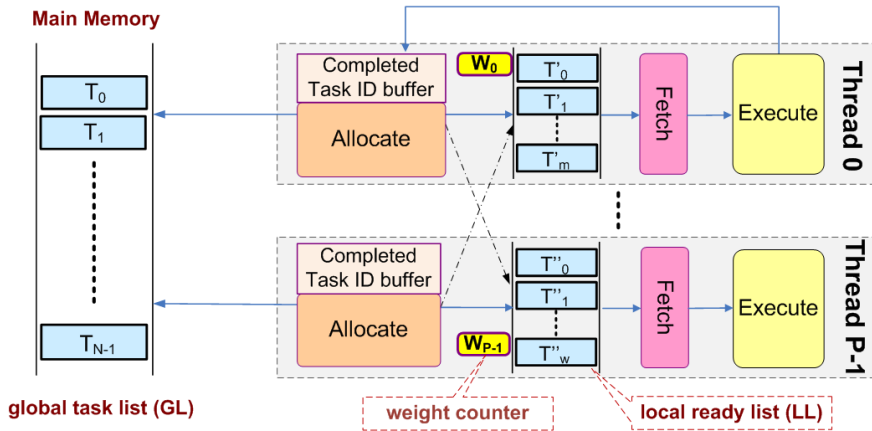


Figure 1: Components of the collaborative scheduler.

task. For example, if the ID of T_i in Figure 2 was fetched by Thread l from the Completed Task ID buffer, then Thread l locates the successors of T_i and decreases their task dependency degrees. If the task dependency degree of a successor T_j , becomes 0, the Allocate module allocates T_j to a thread. Heuristics can be used in task allocation to balance the workload across the threads. Our framework permits plug-and-play insertion of such modules. In this paper, we allocate a task to the thread with the smallest workload at the time of completion of the task execution.

The *local task list* (LL) in each thread stores the tasks allocated to the thread. For load balance, the Allocate module of a thread can allocate tasks to any thread. Thus, the LLs are actually *shared* by all threads. Each LL has a *weight counter* associated with it to record the total workload of the tasks currently in the LL. Once a task is inserted into (or fetched from) the LL, the weight counter is updated.

The *Fetch module* takes a task from the LL and sends it to the *Execute module* in the same thread for execution. Heuristics can be used by the Fetch module to select tasks from the LL. For example, tasks with more children can have higher priority for execution [15]. In this paper, we use a straightforward method, where the task at the head of the LL is selected by the Fetch module. Once the execution of the task is completed, the Execute module sends the ID of the task to the Completed Task ID buffer, so that the Allocate module can accordingly decrease the dependency degree of the successors of the task.

We emphasize that Figure 1 shows the *framework* of collaborative scheduling, where various heuristics can be used for the components. For example, genetic algorithms or randomization techniques can be used for the Allocate and Fetch modules [15]. In this paper, we focus on the reduction of overhead of collaborative scheduling with respect to simple implementations of the components.

4.2 An Implementation of Collaborative Scheduler

Based on the framework of collaborative scheduling in Section 4.1, we present a sample implementation of collaborative scheduling in Algorithm 1. We use the following notations in the algorithm: GL denotes the global task list. LL_i denotes the local task list in Thread i , $0 \leq i < P$. d_T and w_T denote the dependency degree and the weight of task

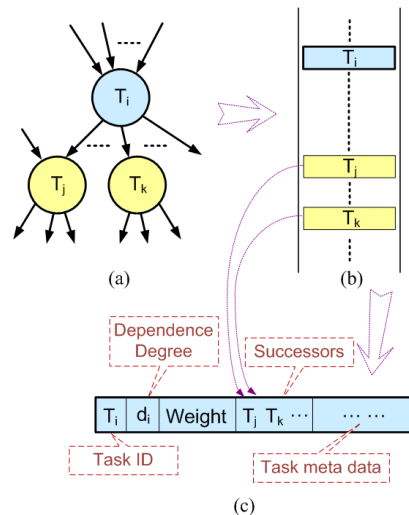


Figure 2: (a) A portion of a task dependency graph. (b) The corresponding representation of the global task list (GL). (c) The data of element T_i in the GL.

T , respectively. W_i is the weight counter of Thread i , i.e. the total weight (estimated execution time) of the tasks currently in LL_i . δ_M is a given threshold. The statements in boxes involve shared variable accesses.

Lines 1 and 2 in Algorithm 1 initialize the local task lists. In Lines 3-19, the algorithm performs task scheduling iteratively until all the tasks are processed. Lines 5-14 correspond to the Allocate module, where the algorithm decreases the dependency degree of the successors of tasks in the Completed Task ID buffer (Line 7), then allocates the successors with dependency degree equal to 0 into a target thread (Line 9). Line 5 determines when the Allocate module should work. When the number of tasks in the Completed Task ID buffer is greater than the threshold δ_M , the Allocate module fetches tasks from the GL. The motivation is that accessing the shared GL less frequently can reduce the lock overhead. We choose the target thread as the one with the smallest workload (Line 9), although alternative heuristics can be used. Line 15 corresponds to the Fetch module, where we fetch a task from the head of the local

Algorithm 1 A Sample Implementation of Collaborative Scheduling

```

{Initialization}
91: Let  $S = \{T_i | d_i = 0\}, 0 \leq i < P$ 
92: Evenly distribute tasks in  $S$  to  $LL_i, 0 \leq i < P$ 
{Scheduling}
93: for Thread  $i (i = 0 \dots P - 1)$  pardo
94:   while  $GL \cup LL_i \neq \phi$  do
     {Completed Task ID buffer & Allocate module}
95:   if  $|Completed\ Task\ ID\ buffer| > \delta_M$  or  $LL_i = \phi$ 
     then
96:     for all  $T \in \{successors\ of\ tasks\ in\ the\ Completed\ Task\ ID\ buffer\ of\ Thread\ i\}$  do
97:        $d_T = d_T - 1$ 
98:       if  $d_T = 0$  then
99:         fetch  $T$  from  $GL$  and append it to  $LL_j$ ,
          where  $j = \arg \min_{t=1 \dots P} (W_t)$ 
100:         $W_j = W_j + w_T$ 
101:       end if
102:     end for
103:     Completed Task ID buffer =  $\phi$ 
104:   end if
   {Fetch module}
105:   fetch a task  $T'$  from the head of  $LL_i$ 
106:    $W_i = W_i - w_{T'}$ 
   {Execute module}
107:   execute  $T'$  and place the task ID of  $T'$  into the
     Completed Task ID buffer of Thread  $i$ 
108:   end while
109: end for

```

task list. Line 16 updates W_i according to the fetched task. Finally, in Line 17, the fetched task T' is executed in the Execute module, and its ID is placed into the Completed Task ID buffer.

Due to the collaboration among threads (Line 9), all local task lists and weight counters are **shared** by all threads. Thus, in addition to the GL , we must avoid concurrent write to LL_i and $W_i, 0 \leq i < P$, in Lines 9, 10, 15, and 16.

5. LOCK-FREE STRUCTURES

5.1 Organization

In Algorithm 1, collaboration among the threads requires locks for the weight counters and local task lists to avoid concurrent writes. For example, without locks, multiple threads can concurrently insert tasks into local task list LL_j , for some $j, 0 \leq j < P$ (Line 9, Algorithm 1). In such a scenario, the tasks inserted by a thread can be overwritten by those inserted by another thread. Similarly, without locks, data race can occur to weight counter W_j (Line 10, Algorithm 1). Therefore, locks must be used for weight counters and local task lists. Locks serialize the execution and incur increasing overheads as P increases. Thus, we focus on eliminating the locks for local task lists and weight counters.

We propose a lock-free organization for the local task lists

and weight counters. We substitute P weight counters for each original weight counter and P circular lists for each local task list, so that each weight counter or circular list is updated *exclusively* during scheduling. Thus, there are P^2 lock-free weight counters and P^2 lock-free circular lists. The lock-free organization is shown in Figure 3, where the dashed box on the left-hand side shows the organization of local task list LL_i and weight counter W_i in Thread i for some $i, 0 \leq i < P$; the dashed circles on the right-hand side represent the P threads. Although there are more queues and counters than the organization in Figure 1, this organization does not stress the memory. Each queue can be maintained by three variables only: head, tail and counter. If each variable is an integer, then the total size of these variables is less than 1 KB, which is negligible compared with the cache size of almost all multicore platforms.

The lock-free local task list in Thread $i, 0 \leq i < P$, denoted by LL_i , consists of P circular lists $Q_i^0, Q_i^1, \dots, Q_i^{P-1}$. The j -th circular list $Q_i^j, 0 \leq j < P$, corresponds to the j -th portion of LL_i . Each circular list Q_i^j has two pointers, $head_i^j$ and $tail_i^j$, pointing to the first and last task, respectively. Tasks are fetched from (inserted into) Q_i^j at the location pointed to by $head_i^j$ ($tail_i^j$).

The solid arrows in Figure 3 connect the heads of the circular lists in LL_i to Thread i , which shows that Thread i can fetch tasks from the head of *any* circular list in LL_i . Corresponding to Line 15 of Algorithm 1, we let Thread i fetch tasks from the circular lists in LL_i in a round robin fashion. The dash-dotted arrows connecting Thread $j, 0 \leq j < P$, to the tail of Q_i^j imply that Thread j allocates tasks to LL_i by inserting the tasks at the tail of circular list Q_i^j . If Q_i^j is full, Thread j inserts tasks into $Q_k^j, 0 \leq k < P$ and $k \neq i$, in a round robin fashion. According to Figure 3, each head or tail is updated by one thread only.

We use P weight counters in Thread i to track W_i , the workload of LL_i . The P weight counters are denoted $W_i^0, W_i^1, \dots, W_i^{P-1}$ in Figure 3, and W_i is given by $W_i = \sum_{j=0}^{P-1} W_i^j$. Note that all the solid arrows pass through weight counter W_i^i in Figure 3, which means that Thread i must update W_i^i when it fetches a task from *any* circular list. The dash-dotted arrow passing through W_i^j for some $j, 0 \leq j < P$, indicates that Thread j must update W_i^j when it inserts a task into Q_i^j . Note that the value of W_i^j is *not* the total weight of tasks in Q_i^j . Therefore, each weight counter is updated by one thread only during scheduling.

5.1.1 Lock-free Weight Counters

The lock-free organization presented in Section 5.1 substitutes P weight counters for each weight counter used in Algorithm 1. Thus, there are P^2 weight counters overall. We prove that no lock is needed for the P^2 weight counters by analyzing Algorithm 1 with respect to the lock-free organization. According to Algorithm 1, the weight counters are accessed in the following *three* scenarios only:

(a) Thread i , for some $i, 0 \leq i < P$, allocates a task T with weight w_T to Thread $j, 0 \leq j < P$, by appending T to local task list LL_j . Such a scenario corresponds to Line 10 in Algorithm 1, where we update W_j after a new task is allocated to Thread j . According to the lock-free organization, in this scenario, Thread i updates $W_j^i = W_j^i + w_T$. If another thread, say Thread $k, k \neq i$, appends a

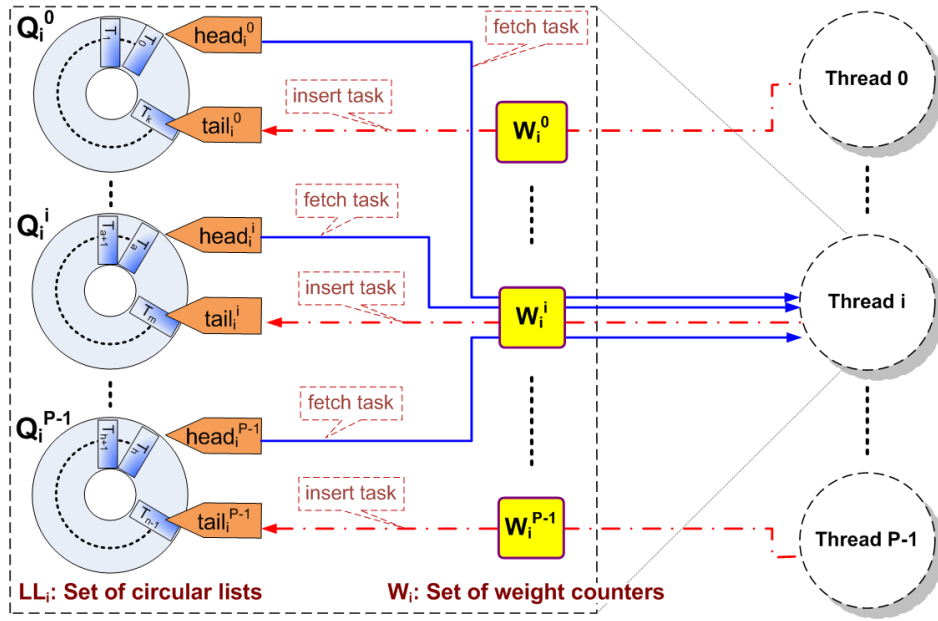


Figure 3: Lock-free organization of local task list (LL_i) and a set of weight counters in Thread i and their interaction with respect to each thread.

task to LL_j concurrently, a different weight counter W_j^k is updated. Thus, W_j^i is not accessed by any other thread.

(b) Thread i , for some i , $0 \leq i < P$, fetches a task T with weight w_T from LL_i . This scenario corresponds to Line 16 in Algorithm 1, where we fetch a task from the local task list for execution. Using the lock-free organization, Thread i updates $W_i^i = W_i^i - w_T$, when Thread i fetches a task from *any* circular list in LL_i . Since *only* Thread i can fetch tasks from LL_i , W_i^i is not accessed by any other thread.

(c) Thread i , for some i , $0 \leq i < P$, queries the workload of LL_j , $0 \leq j < P$. Such a scenario corresponds to Line 9 in Algorithm 1, where we select the thread with the smallest workload. In such a scenario, Thread i reads all the weight counters in Thread j , then computes $W_j = \sum_{k=0}^{P-1} W_j^k$. Other threads can query the workload of LL_j concurrently. Note that no write operation is performed.

According to the above scenarios, any weight counter can be read concurrently (Scenario c), but written exclusively (Scenarios a, b). Therefore, no lock is needed for weight counters to avoid concurrent write. A thread may read a stale value of W_j^i , if Thread i is concurrently updating the counter. This scenario is discussed in Section 5.2.

5.1.2 Lock-free Local Task Lists

The lock-free organization presented in Section 5.1 substitutes P circular lists for each local task list used in Algorithm 1. Thus, there are P^2 circular lists overall. We prove that no lock is needed for the P^2 circular lists. According to Algorithm 1, any circular list is accessed in the following *three* scenarios only:

(a) Thread i , for some i , $0 \leq i < P$, allocates a task T to Thread j , $0 \leq j < P$, by appending T to LL_j . This scenario corresponds to Line 9 in Algorithm 1, where we allocate a new task to the thread with the smallest load. According to the lock-free organization, Thread i inserts T into the location pointed to by $tail_j^i$ in Q_j^i , then updates $tail_j^i =$

$tail_j^i + 1$. If another thread, say Thread k , $k \neq i$, appends a task to LL_j concurrently, the tail of a different circular list Q_j^k is updated. $tail_j^i$ is not accessed by Thread k , $k \neq i$.

(b) Thread i , for some i , $0 \leq i < P$, fetches a task from LL_i for execution. This corresponds to Line 15 in Algorithm 1, where we fetch a task from the local task list for execution. In this scenario, Thread i finds a nonempty circular list Q_j^i , for some j , $0 \leq j < P$, from LL_i in a round robin fashion. Then, Thread i fetches a task from Q_j^i at the location pointed to by $head_j^i$ and updates $head_j^i = head_j^i - 1$.

(c) Thread i , for some i , $0 \leq i < P$, checks if Q_j^i in LL_j is empty (full). This is implicitly required by Line 15 (Line 9) in Algorithm 1. Q_j^i is empty (full) if $head_j^i = tail_j^i$ ($head_j^i = tail_j^i + 1$). In this scenario, Thread i reads $head_j^i$ and $tail_j^i$, and checks if $head_j^i = tail_j^i$ ($head_j^i = tail_j^i + 1$). Other threads can check Q_j^i concurrently.

According to the above scenarios, any $head_j^i$ and $tail_j^i$ can be read concurrently (Scenario c) but written by one thread only (Scenarios a, b). Therefore, no lock is needed for the circular lists to avoid concurrent write. Note that a thread may read a stale value of a head (tail), if another thread is concurrently updating the head (tail). This scenario is discussed in Section 5.2.

5.2 Correctness

When a thread is writing data to shared variables, e.g., the weight counters, the head and tail of the circular lists, another thread may read stale values of these shared variables if we eliminate locks. However, such a stale value does not impact the *correctness*. By correctness, we mean that all the precedence constraints are satisfied and each of the tasks in the DAG is scheduled onto some thread.

All the precedence constraints of a given DAG structured computation are satisfied because of the locks for the global task list (GL). For each task T in the GL, the task dependency degree d_T is protected by a lock to avoid concurrent

write. Thus, according to Lines 7 and 8 in Algorithm 1, T can be allocated only if all the precedence tasks of T are processed. Therefore, the precedence constraints are satisfied regardless of potentially stale values.

When a stale value of a weight counter W_i^j is read by a thread, an error is introduced into the workload estimate for LL_i , i.e. $W_i = \sum_{j=0}^{P-1} W_i^j$. Due to the inaccurate workload estimate, the thread may allocate a new task to a different thread, compared with the case for which the exact workload is known. This does not affect the correctness, but may lead to an unbalanced load. However, since the weight counters are checked in every iteration of scheduling (Lines 4-18, Algorithm 1), the updated value is taken into consideration in the next iteration of some thread. Therefore, except for temporal influence on load balance, the stale values of the weight counters have no impact on correctness.

We analyze the impact of the stale value of $head_j^i$, $0 \leq i, j < P$, by considering Line 9 in Algorithm 1, where Thread i allocates task T to Thread j by appending T to LL_j . To append T to the lock-free local task list LL_j , Thread i first checks if Q_j^i is full. If not, T is appended at $tail_j^i$ in Q_j^i . According to Scenario (c) in Section 5.1.2, a circular list Q_j^i is full if and only if $head_j^i = tail_j^i + 1$. Consider the following scenario: Assume Q_j^i is full and Thread j fetches a task from Q_j^i . Then, Thread j updates $head_j^i$ and Thread i allocates task T to Thread j . In such a scenario, T can be appended to Q_j^i , since Thread j has taken a task away. However, if Thread i reads the stale value of $head_j^i$, then Q_j^i still appears to be full. Thus, Thread i finds another circular list Q_k^i , $k \neq j$, in a round robin fashion, for task allocation, where Q_k^i is not full. Therefore, the stale value of $head_j^i$ does not affect the correctness, but may lead to an unbalanced load.

Similarly, we analyze the impact of the stale value of $tail_i^j$, $0 \leq i, j < P$, by considering Line 15 in Algorithm 1, where Thread i fetches task T' from LL_i . To fetch a task from a lock-free local task list LL_i , Thread i first checks if Q_i^j is empty, where j is chosen in a round robin fashion. If not empty, the task at $head_i^j$ in Q_i^j is fetched. Otherwise, Thread i fetches a task from the next nonempty circular list in LL_i . Consider the following scenario: Assume Q_i^j is empty and Thread j appends task T' to Q_i^j . Then, Thread j updates $tail_i^j$ and Thread i fetches a task from Q_i^j . In such a scenario, Thread i can fetch a task from Q_i^j , since Thread j has inserted T' into Q_i^j . However, if Thread i reads the stale value of $tail_i^j$, then Q_i^j still appears to be empty. Thus, Thread j fetches a task from another circular list. The updated value of $tail_i^j$ can be read in the next iteration of scheduling. Then Thread j can fetch T' for execution. Therefore, the stale value of $tail_i^j$ does not affect the correctness.

6. EXPERIMENTS

6.1 Computing Facilities

We conducted experiments on *three* state-of-the-art homogeneous multicore systems: (1) The dual Intel Xeon 5335 (Clovertown) quadcore platform contained two Intel Xeon x86_64 E5335 processors, each having four cores. The processors ran at 2.00 GHz with 4 MB L2 cache each and 16 GB DDR2 shared memory. The operating system was Red

Hat Enterprise Linux WS Release 4 (Nahant Update 7). We installed GCC version 4.1.2 compiler with streaming SIMD extensions 3 (SSE 3), also known as Prescott New Instructions. (2) The dual AMD Opteron 2335 (Barcelona) quad-core platform had dual AMD Opteron x86_64 2350 quadcore processors, running at 2.0 GHz. The system had 16 GB DDR2 memory, shared by all the cores, and the operating system was CentOS version 5 Linux. We also used the GCC 4.1.2 compiler on this platform. (3) The quad AMD Opteron 8358 (Barcelona) quadcore platform had four AMD Opteron x86_64 8358 quadcore processors, running at 1.2 GHz. The system had 64 GB shared memory and the operating system was Red Hat Enterprise Linux Server Release 5.3 (Tikanga). We also used GCC 4.1.2 compiler on this platform.

6.2 Baseline Schedulers

To evaluate the proposed method, we implemented *six* baseline methods and compared them along with the proposed one using the same input task dependency graphs on various multicore platforms.

(1) Centralized scheduling with shared core (**Cent shared**): This scheduling method consisted of a *scheduler thread* and several *execution threads*. Each execution thread was bound to a core, while the scheduler thread can be executed on any of these cores. In this scheduling method, the input DAG was *local* to the scheduler thread. Each execution thread has a ready task list *shared* with the scheduler thread. In addition, there was a Completed Task ID buffer *shared* by all the threads. The scheduler thread was in charge of all the activities related to scheduling, including updating task dependency degrees, fetching ready-to-execute tasks and evenly distributing tasks to the ready task lists. Each execution thread fetched tasks from its ready task list for execution. The IDs of completed tasks were inserted into the Completed Task ID buffer, so that the scheduler thread can fetch new ready-to-execute tasks from the successors of tasks in the Completed Task ID buffer. After a given number (i.e. δ_M in Algorithm 1) of tasks in the ready task list were processed, the execution thread invoked the scheduler thread and then went to sleep. When a task was allocated to the ready task list of a sleeping thread, the scheduler invoked the corresponding execution thread. Spinlocks were used for the ready task lists and Completed Task ID buffer.

(2) Centralized scheduling with dedicated core (**Cent ded**): This scheduling method was adapted from the centralized scheduling with shared core. The only difference is that a core was dedicated to the scheduler thread, i.e., each thread was bound to a separate core. Similar to **Cent shared**, the input DAG was *local* to the scheduler. Each execution thread had a ready task list *shared* with the scheduler thread. There was a Completed Task ID buffer *shared* by all the threads. The scheduler thread was also in charge of all the activities related to scheduling and the execution threads executed assigned tasks only. Spinlocks were used for the ready task lists and Completed Task ID buffer.

(3) Distributed scheduling with shared ready task list (**Dist shared**): In this method, we distributed the scheduling activities across the threads. This method had a *shared* global task list and a *shared* ready task list. Each thread had a *local* Completed Task ID buffer. The schedulers integrated into each thread fetched ready-to-execute tasks from the global task list, and inserted the tasks into the shared ready task list. If the ready task list was not empty, each thread fetched

tasks from the ready task list for execution. Each thread inserted the IDs of completed tasks into its Completed Task ID buffer. Then, the scheduler in each thread updated the dependency degree of the successors of tasks in the Completed Task ID buffer, and fetched the tasks with dependency degree equal to 0 for allocation. Pthreads spinlocks were used for the global task list and the ready task list.

(4) Collaborative scheduling with *lock-based* local task list (**ColSch lock**): This was the collaborative scheduling (Section 4) without any optimization discussed in Section 5. Rather than using the lock-free data structures, we used mutex locks to avoid concurrent write to the local task lists and weight counters. Spinlocks were used to protect task dependency degrees in the shared global task list.

(5) Collaborative scheduling with *lock-free* local task list (**ColSch lockfree**): This was *not* a baseline, but the proposed method (Section 4) with lock-free weight counters and local task lists (Section 5). We used spinlocks to protect task dependency degrees in the shared global task list.

(6) DAG Scheduling using the Cilk (**Cilk**): This baseline scheduler performed work stealing based scheduling using the Cilk runtime system. Unlike the above scheduling methods where we bound a thread to a core of a multicore processor and allocated tasks to the threads, we dynamically created a thread for each ready-to-execute task and then let the Cilk runtime system to schedule the threads onto cores. Although Cilk can generate a DAG dynamically, we used a given task dependency graph stored in a *shared* global list for the sake of fair comparison. Once a task completed, the corresponding thread reduced the dependency degree of the successors of the task and created new threads for the successors with resulting dependency degree equal to 0. We used spinlocks for the dependency degrees to prevent concurrent write.

(7) Our implementation of *work stealing* based scheduler (**Stealing**): Although the above baseline **Cilk** is also a work stealing scheduler, it used the Cilk runtime system to schedule the threads, each corresponding to a task. On the one hand, the Cilk runtime system has various additional optimizations; on the other hand, scheduling the threads onto cores incurs overhead due to context switching. Therefore, for the sake of fair comparison, we implemented the **Stealing** baseline; we distributed the scheduling activities across the threads, each having a *shared* ready task list. The global task list was *shared* by all the threads. If the ready task list of a thread was not empty, the thread fetched a task from it at the top for execution and upon completion updated the dependency degree of the successors of the task. Tasks with dependency degree equal to 0 were placed into the top of its ready task list by the thread. When a thread ran out of tasks to execute, it randomly chose a ready list to steal a task from its bottom, unless all tasks were completed. The data for randomization were generated offline to reduce possible overhead due to random number generator. Pthreads spinlocks were used for the ready task lists and global task list.

6.3 Datasets and Task Types

We used both synthetic and real datasets to evaluate the proposed scheduling methods. We built a task dependency graph for exact inference in a Bayesian network used by a real application called QuickMedical Reference decision theoretic version (QMR-DT), a microcomputer-based decision

support tool for diagnosis in internal medicine [13]. There were 1000 nodes in this network of 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. The resulting task dependency graph for exact inference had 228 tasks, each corresponding to a series of computations called node level primitives. These primitives consist of addition, multiplication and division operations among single precision floating point data in one or two tables [21]. Given the number of random variables involved in each task and the number of states of the random variables, denoted by W_c and r , respectively, the table size is $4r^{W_c}$ bytes. Each table entry was a 4-byte single precision floating point data. In this dataset, r was 2 and the average value for W_c was 10. The average number of successors d for each task was also 2. The task weights were estimated using W_c , r and d (see [21] for details).

We used extensive synthetic datasets to investigate the impact of various parameters of task dependency graphs on the scheduling performance. Our graph generator synthesized DAGs to simulate the structure of programs: Each node represented a code segment. The edges showed the data dependency between the code segments. We assumed that the loops in a program must either be inside a code segment or unrolled. Let N denote the number of nodes in the DAG and d the average node degree. Let $\alpha_0, \alpha_1, \dots, \alpha_{N-1}$ denote the nodes. We started with an empty set of edges. When node α_i was visited, the generator spawned $\max(0, (d - d_{in} + \delta))$ outgoing edges for α_i , where d_{in} is the number of incoming edges and $\delta \in \lfloor -d/2 \rfloor, \lceil d/2 \rceil$ is a random integer. The outgoing edges were connected to nodes selected randomly from α_{i+1} to α_{N-1} : The probability that $\alpha_j, i < j < N$, was selected is $P(\alpha_i \rightarrow \alpha_j) = e^{-1/(j-i)}$. That is, for a given node α_i , the nodes close to α_i had higher probability of selection. Finally, all the nodes with no parent, except for α_0 , were connected to α_0 . We generated DAGs with 10000 nodes. The average degree of the nodes was 8. The threshold for the Task ID buffer δ_M was 5.

We experimented with three types of tasks for *each node* of the generated DAGs in our experiments: (1) **Dummy task**: Dummy task was actually a timer, which allowed us to assign the execution time of a task. Dummy tasks also helped us analyze the scheduling overhead, since we could easily calculate the task execution time. (2) **Computation intensive task**: This simulated a number of real tasks. We performed matrix multiplication $Y = X^T X$, where X was a 256×256 matrix. (3) **Memory access intensive task**: This simulated another common type of real task, which performed irregular memory access. In our experiments, we updated each element $x[i]$ of an array with 128k entries by performing $x[i] = x[y[i]]$, where y was an index array for x .

6.4 Experimental Results

We first compared the proposed scheduling method (i.e. **ColSch lockfree**) with the two work stealing based methods **Stealing** and **Cilk** using the task dependency graph for exact inference in the QMR-DT Bayesian network on the platform with dual AMD Opteron 2335 quadcore processors. In each comparison, the input task dependency graphs were *identical* for all the methods. In Figure 4(a), we show the execution time for scheduling exact inference using the task dependency graph. The corresponding speedup is shown in

Figure 4(b). According to Figure 4(a), when the number of threads is less than four, almost no difference in execution time can be observed. However, as the number of threads increases, we can observe that the proposed method led to less execution time than the two baseline methods. The difference is clear in Figure 4(b), where we converted the execution time into speedup. A reason for the difference in the speedups is that the proposed scheduling method balanced the workload across the threads according to the task weights. However, both the baseline methods did not consider the task weights. Instead, they stole tasks from other threads to feed the underutilized threads. Since the DAG had 228 nodes and the number of threads was 8, the number of tasks per thread was 28.5 on the average. Thus, the number of available tasks for stealing at a particular time was even less. In this scenario, there is no guarantee for work stealing to achieve load balance across the threads. Note that, although the results of **Stealing** and **Cilk** were similar in Figure 4(b), the performance of **Stealing** was slightly better than **Cilk**. One reason for the performance difference was the overhead due to thread creation and the context switch. For **Cilk**, we dynamically created threads for the tasks. This incurs thread creation overheads. In addition, we let the runtime system to swap the threads in the cores, which resulted in context switch overhead. However, for the other methods, we bound the threads to separate cores and allocated tasks to the threads in user space, where no thread creation or swap in/out occurred.

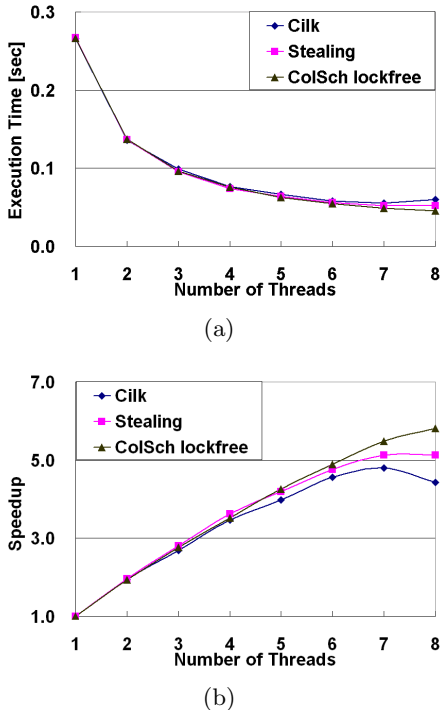


Figure 4: Comparison of the proposed scheduler with work stealing based schedulers.

We conducted the following *four* experiments using synthetic datasets on multicore platforms to investigate the impact of various parameters of input task dependency graphs on scheduling performance. The first experiment compared

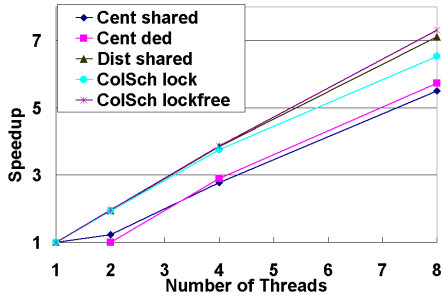
the proposed scheduling method and various baseline methods. We used our graph generator to construct a DAG with 10000 nodes, where each node was assigned a dummy task taking 50 microseconds. We executed the DAG using the five task sharing scheduling methods on a dual Intel Xeon 5335 (Clovertown) quadcore platform and a quad AMD Opteron 8358 (Barcelona) quadcore platform, where up to 8 and 16 concurrent threads were supported, respectively. Given various available threads, we measured the overall execution time on both platforms and converted it into speedups in Figures 5(a) and 6(a), where the speedup is defined as the serial execution time over parallel execution time.

The second experiment investigated the overhead due to the proposed lock-free collaborative scheduling method with respect to the number of threads and task size on various platforms. As in the above experiment, we used a DAG with 10000 nodes and dummy tasks. However, we varied the time delay for each dummy task. We show the ideal speedup and the experimental results with dummy tasks taking 50, 100, 500 and 1000 microseconds in Figures 5(b) and 6(b).

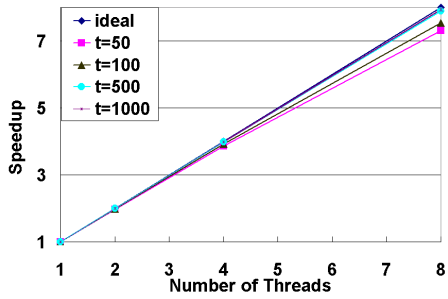
The third experiment showed the impact of the error in the estimated task execution time. We again used a DAG with 10000 dummy task nodes. However, instead of using the real execution time as the task weight, we used estimated execution time with errors. Let t denote the real task execution time and t' the estimated execution time. The real task execution time t was randomly selected from range $((1 - r/100)t', (1 + r/100)t')$, where r is the absolute percentage error in estimated task execution time. We conducted experiments with $r = 0, 1, 5, 10, 20$ and show the results Figures 5(c) and 6(c). Note that the definition of speedup in the above two figures is the serial execution time over 8-thread parallel execution time.

The last experiment examined the impact of task types. We used the lock-free collaborative scheduler to schedule a DAG with 10000 tasks. In the experiments, we used dummy tasks, computation intensive tasks and memory access intensive tasks, discussed in Section 6.3. The results are shown in Figures 5(d) and 6(d).

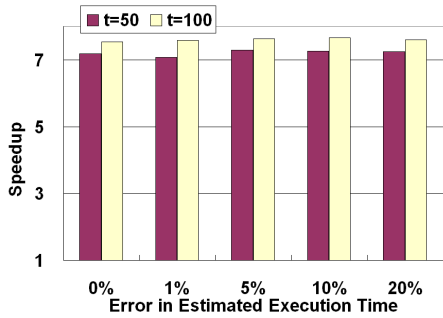
According to Figures 5 and 6, the proposed method showed superior performance compared with baseline methods on various platforms as number of cores increased. For example, in Figure 6(a), when 16 cores were used, the speedup achieved by our method ($15.12\times$) was significantly higher than others methods (up to $8.77\times$). This was because more cores lead to higher conflict in accessing the task lists. The proposed method allowed such concurrent access, but the baseline methods serialized the accesses. The baseline **Cent shared** showed relatively low performance due to the overhead of frequent context switch between the scheduler thread and the execution thread. **Cent ded** also showed limited speedup, since the computing capability of the core dedicated to the scheduler thread was not used efficiently. The proposed method exhibited good scalability for various task weights. The speedups were close to the ideal speedup for large task weights. The results showed that our technique was tolerant to the error in estimation of task weights. This was because that the errors in estimation of task weights could counteract each other. In addition, we observed that the scheduling overhead was less than 1% for the proposed method in all the experiments. Finally, the proposed method achieved almost linear speedup for various tasks, which im-



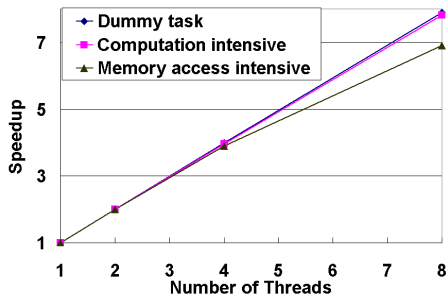
(a) Comparison with baseline scheduling methods.



(b) Impact of the size of tasks on the speedup achieved by ColSch lockfree.

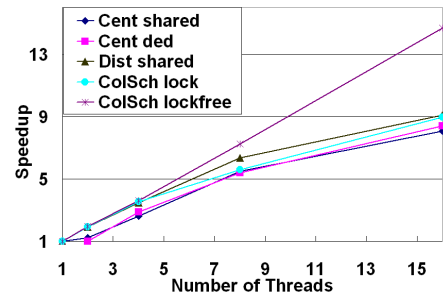


(c) Impact of error in the estimated task execution time on the speedup achieved by ColSch lockfree.

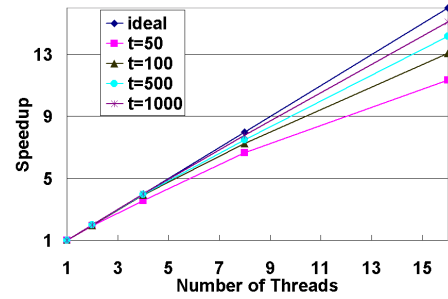


(d) Performance of ColSch lockfree with respect to task types.

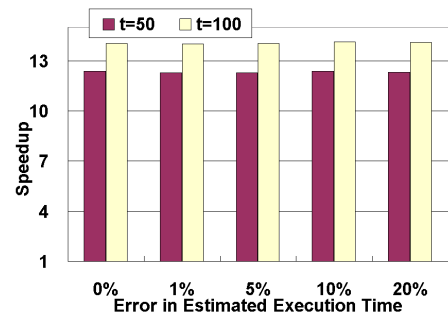
Figure 5: Experimental results on dual Intel Xeon 5335 (Clovertown) quadcore platform.



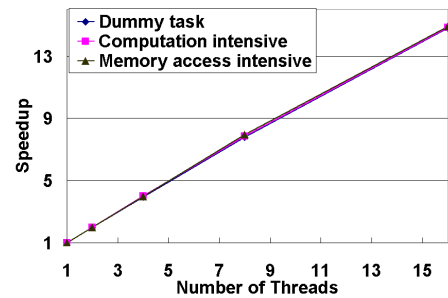
(a) Comparison with baseline scheduling methods.



(b) Impact of the size of tasks on the speedup achieved by ColSch lockfree.



(c) Impact of error in the estimated task execution time on the speedup achieved by ColSch lockfree.



(d) Performance of ColSch lockfree with respect to task types.

Figure 6: Experimental results on quad AMD Opteron 8358 (Barcelona) quadcore platform.

plied that the proposed scheduler can work well in real life scenarios.

7. CONCLUSIONS

We proposed a lightweight scheduling method called collaborative scheduling for DAG structured computations on general purpose multicore processors. The scheduler consists of several components; various heuristics could be used for optimization within each component. For example, randomization techniques could be used for the Allocate module [15]. We developed lock-free local task lists and weight counters. These differ from traditional lock-free data structures. In the future, we plan to reduce the number of variables used by the lock-free data structure. We would also like to explore the heuristics for each component in the proposed scheduler. For example, for the task fetch module, we plan to interleave computationally intensive tasks with memory access intensive tasks for the threads assigned to the same core to improve the overall performance.

8. ACKNOWLEDGMENTS

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 too. We would also like to thank Michail Giakkoupis and Nam Ma for useful discussion.

9. REFERENCES

- [1] 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*, pages 207–213, 1996.
- [2] 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.*, pages 1–6, 2008.
- [3] S. Alarm, R. Barrett, J. Kuehn, P. Roth, and J. Vetter. Characterization of scientific workloads on systems with multicore processors. In *IEEE International Symposium on Workload Characterization*, pages 225–236, 2006.
- [4] J. H. Anderson and S. Ramamurthy. Lock-free and practical doubly linked list-based dequeues using single-word compare-and-swap. In *Proceedings of the 8th International Conference On Principles Of Distributed Systems*, pages 240–255, 2005.
- [5] O. Beaumont, L. Carter, J. Ferrante, A. Legr, L. Marchal, and Y. Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2006.
- [6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. Technical report, Cambridge, 1996.
- [7] H.-L. Chen and C.-T. King. Eager scheduling with lazy retry in multiprocessors. *Future Generation Computer Systems*, 17(3):215–226, 2000.
- [8] E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, New York, NY, 1976.
- [9] G. Cong and D. A. Bader. Designing irregular parallel algorithms with mutual exclusion and lock-free protocols. *Journal of Parallel and Distributed Computing*, 66:854–866, 2006.
- [10] 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.*, pages 1–6, 2006.
- [11] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.
- [12] Intel Threading Building Blocks. <http://www.threadingbuildingblocks.org/>.
- [13] T. S. Jaakkola and M. I. Jordan. Variational probabilistic inference and the QMR-DT network. *Journal of Artificial Intelligence Research*, 10(1):291–322, 1999.
- [14] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrent Computing: Practice Experiment*, 22(1):15–44, 2010.
- [15] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.
- [16] J. Liu, D. M. Nicol, and K. Tan. Lock-free scheduling of logical processes in parallel simulation. In *Proceedings of the 2000 Parallel and Distributed Simulation Conference*, pages 22–31, 2001.
- [17] OpenMP Application Programming Interface. <http://www.openmp.org/>.
- [18] 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*, pages 510–513, 1988.
- [19] M. S. Squillante and R. D. Nelson. Analysis of task migration in shared memory multiprocessor scheduling. In *ACM Conf. on the Measurement and Modeling of Comp. Syst.*, pages 143–155, 1991.
- [20] X. Tang and G. R. Gao. Automatically partitioning threads for multithreaded architectures. *J. Parallel Distrib. Comput.*, 58(2):159–189, 1999.
- [21] Y. Xia and V. K. Prasanna. Scalable node-level computation kernels for parallel exact inference. *IEEE Trans. Comput.*, 59(1):103–115, 2010.
- [22] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2006.