Hierarchical Scheduling of DAG Structured Computations on Manycore Processors with Dynamic Thread Grouping*

Yinglong Xia[1], Viktor K. Prasanna[2,1] and James Li[2]

Department of Computer Science[1]
Ming Hsieh Department of Electrical Engineering[2]
University of Southern California, Los Angeles, CA 90089, U.S.A.
{yinglonx,prasanna,jamesyli}@usc.edu

Abstract. Many computational solutions can be expressed as directed acyclic graphs (DAGs) with weighted nodes. In parallel computing, scheduling such DAGs onto manycore processors remains a fundamental challenge, since synchronization across dozens of threads and preserving precedence constraints can dramatically degrade the performance. In order to improve scheduling performance on manycore processors, we propose a hierarchical scheduling method with dynamic thread grouping, which schedules DAG structured computations at three different levels. At the top level, a supermanager separates threads into groups, each consisting of a manager thread and several worker threads. The supermanager dynamically merges and partitions the groups to adapt the scheduler to the input task dependency graphs. Through group merging and partitioning, the proposed scheduler can dynamically adjust to become a centralized scheduler, a distributed scheduler or somewhere in between, depending on the input graph. At the group level, managers collaboratively schedule tasks for their workers. At the within-group level, workers perform self-scheduling within their respective groups and execute tasks. We evaluate the proposed scheduler on the Sun UltraSPARC T2 (Niagara 2) platform that supports up to 64 hardware threads. With respect to various input task dependency graphs, the proposed scheduler exhibits superior performance when compared with other various baseline methods, including typical centralized and distributed schedulers.

Key words: Manycore processor, hierarchical scheduling, thread grouping

1 Introduction

Given a program, we can represent the program as a *directed acyclic graph* (DAG) with weighted nodes, in which the nodes represent code segments, and edges represent dependencies among the segments. An edge exists 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.

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

Such a DAG is called a *task dependency graph*, and the computations that can be represented as task dependency graphs are called *DAG structured computations* [2, 12]. The *objective* of task scheduling for DAG structured computations on manycore processors is to minimize the overall execution time by proper allocation of the tasks to concurrent threads, while preserving the precedence constraints among the tasks [12, 21].

Scheduling DAG structured computations on manycore processors is a fundamental challenge in parallel computing nowadays. The trend in architecture design is to integrate more and more cores onto a single chip to achieve higher performance. Such architectures are known as manycore processors. Examples of existing manycore processors include the Sun UltraSPARC T1 (Niagara) and T2 (Niagara 2), which support up to 32 and 64 concurrent threads, respectively [16]. The Nvidia Tesla and Tilera TILE64 are also available. More manycore processors are emerging soon, such as the Sun Rainbow Falls, IBM Cyclops64 and Intel Larrabee [18]. Such processors are more interested in how many tasks from a DAG can be completed efficiently over a period of time rather than how quickly an individual task can be completed.

Our contributions in this paper include: (a) We propose a hierarchical scheduling method which schedules DAG structured computations at three different levels on manycore systems. (b) We propose a dynamic thread grouping technique to merge or partition the thread groups at run time, so that the proposed scheduler can dynamically adjust to become a centralized scheduler, a distributed scheduler or somewhere in between, depending on the input graph. (c) We implement the hierarchical scheduling method on the Sun UltraSPARC T2 (Niagara 2) platform. (d) We conduct extensive experiments to validate the proposed method.

The rest of the paper is organized as follows: In Section 2, we review the background and related work. Section 3 presents the hierarchical scheduling scheme. We illustrate experimental results in Section 4 and address the future research in Section 5.

2 Background and Related Work

In this paper, 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. A task can begin execution only if all of its predecessors have been completed [1]. 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 [8, 15]. We consider scheduling an arbitrary DAG with given task weights and decide 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 [12].

The scheduling problem has been extensively studied for several decades [2, 4, 12, 17]. Early algorithms optimized scheduling with respect to the specific structure of task dependency graphs [7], such as a tree or a fork-join graph. In general, however, programs come in a variety of structures [12]. Karamcheti and Chien studied hierarchical load balancing framework for multithreaded computations for employing various scheduling policies for a system [10]. Recent research on scheduling DAGs in-

cludes [20] where the authors studied the problem of scheduling more than one DAG simultaneously onto a set of heterogeneous resources, and [2] where Ahmad proposed a game theory based scheduler on multicore processors for minimizing energy consumption. Dongarra *et al.* proposed dynamic schedulers optimized for some linear algebra problems on general-purpose multicore processors [17]. Scheduling techniques have been proposed by several emerging programming systems such as Cilk [5], Intel Threading Building Blocks (TBB) [9], OpenMP [14], Charm++ [6] and MPI microtasking [13], 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 [11]. In contrast with these systems, we focus on scheduling for DAGs on manycore processors.

To design an efficient scheduler we must take into account the architectural characteristics of processors. Almost all the existing manycore processors have relatively simple cores, compared with general-purpose multicore processors, e.g., AMD Opteron and Intel Xeon. For example, the pipeline of the UltraSPARC T2 does not support out of order (OoO) execution and therefore results in a longer delay. However, the fast context switch of such processors overlaps such delays with the execution of another thread. For this reason, the UltraSPARC generally shows higher throughput when enough parallel tasks are available [16].

Directly utilizing traditional scheduling methods such as centralized or distributed scheduling can degrade the performance of DAG structured computations on manycore processors. Centralized scheduling has a single thread to allocate tasks, which may not be able to serve the rest of the threads in time. This leads to starvation of some threads, especially when the tasks can be completed quickly. On the other hand, distributed scheduling requires many threads to schedule tasks. This limits the resources for task execution. In addition, many schedulers accessing shared variables can result in costly synchronization overhead. Therefore, an efficient scheduling method on manycore processors must be able to adapt itself to input task dependency graphs. To the best of our knowledge, no scheduling algorithm for DAG structured computations has been proposed specifically on manycore processors such as the UltraSPARC T2.

3 Hierarchical Scheduling

3.1 Organization

The input graph is represented by a list called the *global task list* (GL). Figure 1(a) shows a portion of the task dependency graph. Figure 1(b) shows the corresponding part of the GL. As shown in Figure 1(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 traversing the entire list. In each element, we have *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.

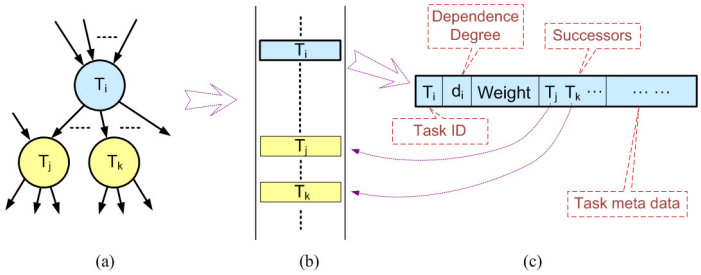


Fig. 1. (a) A portion of a task dependency graph. (b) The corresponding representation of the global task list (GL). (c) The data of element T_i in the GL.

We illustrate the components of the hierarchical scheduler in Figure 2. The boxes with rounded corners represent thread groups. Each group consists of a *manager* thread and several *worker* threads. The manager in Group_0 is also the *supermanager*. The components inside of a box are private to the group; while the components out of the boxes are shared by all groups.

The *global ready list* (GRL) in Figure 2 stores the IDs of tasks with dependency degree equal to 0. These tasks are ready to be executed. During the scheduling process, a task is put into this list by a manager thread once the dependency degree of the task becomes to 0.

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

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

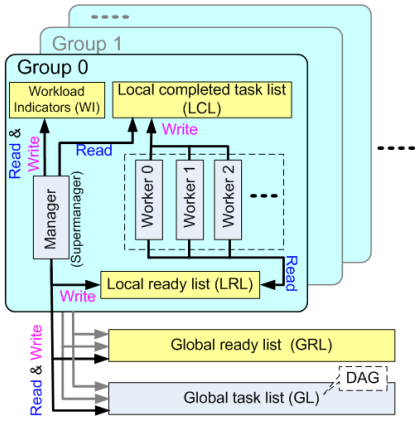


Fig. 2. Components of the hierarchical scheduler.

The arrows in Figure 2 illustrate how each thread accesses a component (read or write). As we can see, GL and GRL are shared by all the managers for both read and

write. For each group, the LRL is write-only for the manager and read-only for the workers; while LCL is write-only for the workers and read-only for the manager. WI is local to the manager in the respective group only.

3.2 Dynamic Thread Grouping

The scheduler organization shown in Figure 2 supports dynamic thread grouping, which means that the number of threads in a group can be adjusted at runtime. We adjust groups by either merging two groups or partitioning a group. The proposed organization ensures efficient group merging and partitioning.

Figure 3(a) illustrates the merging of Group$_i$ and Group$_j$, $i < j$. The two groups are merged by converting all threads of Group$_j$ into the workers of Group$_i$ and merging WIs, LCLs and LRLs accordingly. Converting threads of Group$_j$ into the workers of Group$_i$ is straightforward: $Manager_j$ stops allocating tasks to Group$_j$, but performs self-scheduling as a worker thread. Then, all the threads in Group$_j$ access tasks from the merged LRL and LCL. To combine WI_i and WI_j, we add the value of WI_j to WI_i. Although WI_j is not used after merging, we still keep it updated for the sake of possible future group partitioning. Merging the lists i.e. LCLs and LRLs is efficient. Note that both LCL and LRL are circular lists, each having a head and a tail pointer to indicate the first and last tasks stored in the list, respectively. Figure 3(b) illustrates the approach to merge two circular lists. We need to update two links only, i.e. the bold arrows shown in Figure 3(b). None of the tasks stored in the lists are moved or duplicated. The head and tail of the merged list are $Head_i$ and $Tail_j$, respectively. Note that two merged groups can be further merged into a larger group.

We summarize the procedure in Algorithm 1. Since the queues and weight indicators are shared by several threads, locks must be used to avoid concurrent write. For example, we lock LRL_i and LRL_j immediately before Line 1 and unlock them after Line 3. Algorithm 2 does not explicitly assign the threads in Group$_i$ and Group$_j$ to Group$_k$, since this algorithm is executed only by the supermanager. Each thread dynamically updates its group information and decides if it should be a manager or worker (see Algorithm 2).

Group$_i$ and Group$_j$ can be restored from the merged group by partitioning. As a reverse process of group merging, group partitioning is also straightforward and efficient. Due to space limitations, we do not elaborate it here. Group merging and partitioning can be used for groups with an arbitrary number of threads. We assume the number of threads per group is a power of two hereinafter for the sake of simplicity.

3.3 Hierarchical Scheduling

Using the proposed data organization, we schedule a given DAG structured computation at three levels. The top level is called the *meta-level*, where we have a supermanager to control group merging/partitioning. At this level, we are not scheduling tasks directly, but reconfiguring the scheduler according to the characteristics of the input tasks. Such a process is called *meta-scheduling*. The supermanager is hosted along with the manager of Group$_0$ by Thread$_0$. Note that $Manager_0$ can never become a worker as discussed in Section 3.2.

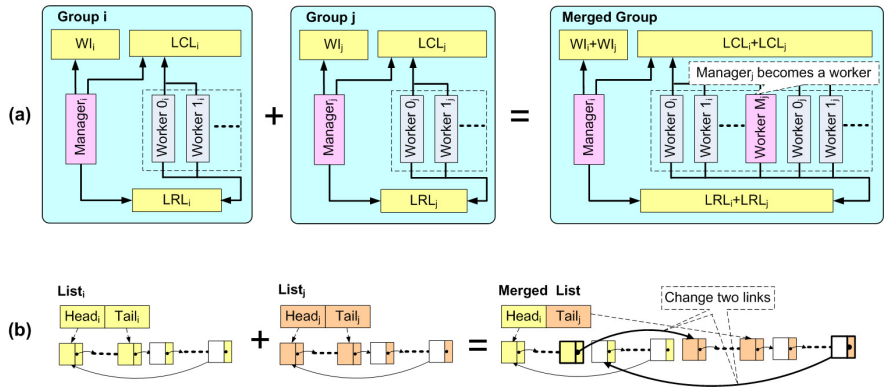


Fig. 3. (a) Merge Group$_i$ and Group$_j$. (b) Merge circular lists $List_i$ and $List_j$. The head (tail) points to the first (last) tasks stored in the list. The blank elements have no task stored yet.

Algorithm 1 Group merge

Input: Group$_i$ and Group$_j$.
Output: Group$_k$ = Group$_i$ + Group$_j$

$\{\texttt{Merge } LRL_i \texttt{ and } LRL_j\}$
1: Let $LRL_j.Head.Predecessor$ points to $LRL_i.Tail.Successor$
2: Let $LRL_i.Tail.Successor$ points to $LRL_j.Head$
3: $LRL_k.Head = LRL_i.Head$, $LRL_k.Tail = LRL_j.Tail$

$\{\texttt{Merge } LCL_i \texttt{ and } LCL_j\}$
4: Let $LCL_j.Head.Predecessor$ points to $LCL_i.Tail.Successor$
5: Let $LCL_i.Tail.Successor$ points to $LCL_j.Head$
6: $LCL_k.Head = LCL_i.Head$, $LCL_k.Tail = LCL_j.Tail$

$\{\texttt{Merge } WI_i \texttt{ and } WI_j\}$
7: $WI_k = WI_i + WI_j$

The mediate level is called the *group level*, where the manager in each group collaborates with each other and allocates tasks for the workers in the group. The purpose of collaborating between managers is to improve the load balance across the groups. Specifically, the managers ensure that the workload in the local ready lists is roughly equal for all groups. A manager is hosted by the first thread in a group.

The bottom level is called the *within-group level*, where the workers in each group perform self-scheduling. That is, once a worker finishes a task execution and updates LCL in its group, it fetches a new task, if any, from LRL immediately. Self-scheduling keeps all workers busy, unless the LRL is empty. Each worker is hosted by a separate thread.

The hierarchical scheduler behaves between centralized and distributed schedulers, so that it can adapt to the input task graph. Note that each group consists of a manager thread and several worker threads. When all the groups are merged into a single group, the proposed method becomes a centralized scheduler; when multiple groups exist, the proposed method behaves as a distributed scheduler.

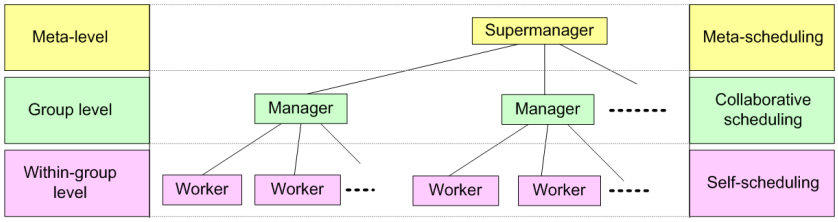


Fig. 4. The hierarchical relationship between the supermanager, managers and workers, and the corresponding scheduling schemes.

3.4 Scheduling Algorithm and Analysis

We propose a sample implementation of the hierarchical scheduler presented in Section 3.3. Based on the organization shown in Section 3.1, we use the following notations to describe the implementation: Assume there are P threads, each bound to a core. The threads are divided into groups consisting of a manager and several workers. GL and GRL denote the global task list and global ready list, respectively. LRL_i and LCL_i denote the local ready list and local completed task list of Group$_i$, $0 \leq i < P$. d_T and w_T represent the dependency degree and the weight of task T, respectively. WI_i is the workload indicator of Thread$_i$. Parameters δ_M, δ_+ and δ_- are given thresholds. The boxes show the statements that access variables shared by all groups.

Algorithm 2 illustrates the framework of the hierarchical scheduler. In Lines 1-3, thread groups are initialized, each with a manager and a worker, along with a set of ready-to-execute tasks stored in LRL_j, where the overall task weight is recorded in WI_j. A boolean flag f_{exit} in Line 3 notifies if the threads can exit the scheduling iteration (Lines 5-15). $rank$ controls the size of groups: Increasing $rank$ leads to merging of two adjacent groups; while decreasing $rank$ leads to partitioning of current groups. $rank = 1$ corresponds to the minimum group size i.e. two threads per group. Thus, we have $1 \leq rank \leq \log P$. The group size Q is therefore given by:

$$Q = \frac{P}{2^{\log P - rank}} = 2^{rank} \tag{1}$$

Line 4 in Algorithm 2 starts all the threads in parallel. The threads perform various scheduling schemes according to their thread IDs. The first thread in each group is a manager (Line 8). In addition, the first thread in Group$_0$ i.e. Thread 0 performs as the supermanager (Line 10). The rest of the threads are workers (Line 13). Given thread ID i, the corresponding group is $\lfloor i/Q \rfloor$.

Algorithm 3 shows the meta-scheduling method for the supermanager. The algorithm consists of two parts: updating $rank$ (Lines 1-2) and re-grouping (Lines 3-11). We use a heuristic to update $rank$: Note that WI_j is the computational workload for Group$_j$. A large WI_j requires more workers for task execution. $|LCL_j|$ is the number of completed tasks and d is the average number of successive tasks. For each completed task, the manager reduces the dependency degree of the successive tasks and moves ready-to-execute tasks to LRL_j. Thus, $(|LCL_j| \cdot d)$ represents the workload for the scheduler. A larger $(|LCL_j| \cdot d)$ requires more managers for task scheduling. In Line 1, the ratio r tells us if we need more managers or more workers. If more workers are

Algorithm 2 A Sample Implementation of Hierarchical Scheduler

Input: P threads; Task dependency graph stored in GL; Thresholds δ_M, δ_+ and δ_-.
Output: Assign each task to a worker thread

{Initialization}
1: Group$_j$={*Manager*: Thread$_{2j}$, *Worker*: Thread$_{2j+1}$}, $j = 0, 1, \cdots, P/2-1$
2: Evenly distribute tasks $\{T_i | T_i \in GL$ and $d_i = 0\}$ across LRL_j, $WI_j = \sum_{T \in LRL_j} w_T$, $\forall j = 0, 1, \cdots, P/2-1$
3: f_{exit} =**false**, $rank = 1$
{Scheduling}
4: **for** Thread $i = 0, 1, \cdots, P-1$ **pardo**
5: **while** f_{exit} =**false do**
6: $Q = 2^{rank}$
7: **if** $i\%Q = 0$ **then**
{Manager thread}
8: Group level scheduling at Group$_{\lfloor i/Q \rfloor}$ (*Algorithm* 4)
9: **if** $i = 0$ **then**
{Supermanager thread}
10: Meta-level scheduling (*Algorithm* 3)
11: **end if**
12: **else**
{Worker thread}
13: Within-group level scheduling at Group $_{\lfloor i/Q \rfloor}$ (*Algorithm* 5)
14: **end if**
15: **end while**
16: **end for**
17: **if** $GL = \emptyset$ **then** f_{exit} =**true**

needed, we increase $rank$ in Line 2. In this case, groups are merged to provide more workers per manager. Otherwise, $rank$ decreases. Line 2 also ensures that $rank$ is valid by checking the boundary values. d, δ_+ and δ_- are given as inputs. The re-grouping depends on the value of $rank$. If $rank$ increases, two groups merge (Line 5); if $rank$ decreases, the merged group is partitioned (Line 9). The two operators Merge($\cdot$) and Partition($\cdot$) are discussed in Section 3.2. Line 12 flips f_{exit} if no task remains in GL. This notifies all of the threads to terminate (Line 5 in Algorithm 3).

Algorithm 4 shows an iteration of the group level scheduling for managers. Each iteration consists of three parts: updating WI_i (Lines 1-2 and 15), maintaining precedence relationship (Lines 3-8) and allocating tasks (Lines 9-14). Lines 3-8 check the successors of all tasks in LCL_i in batch mode to reduce synchronization overhead. Let $m = 2^{rank} - 1$ denote the number of workers per group. In the batch task allocation part (Lines 9-14), we first fetch m tasks from GRL. Line 12 is an adaptive step of this algorithm. If the overall workload of the m tasks is too light ($\sum_{T \in S'} w_T < \Delta W$) or the current tasks in LRL_i is not enough to keep the workers busy ($WI_i < \delta_M$), more tasks are fetched for the next iteration. This dynamically adjusts the workload distribution and prevents possible starvation for any groups. In Line 10, the manager inspects a set of tasks and selects m tasks with relatively more successors. This is a widely used

heuristic for scheduling [12]. Several statements in Algorithm 4 are put into boxes, where the managers access shared components across the groups. Synchronization cost of these statements varies as the number of groups changes.

Algorithm 3 Meta-Level Scheduling for Supermanager

{Update rank}
1: $r = \sum_{j=0}^{P/Q}(WI_j/(|LCL_j| \cdot d))$, $rank_{old} = rank$
2: $rank = \begin{cases} \min(rank+1, \log P), & r > \delta_+ \\ \max(rank-1, 1), & r < \delta_- \end{cases}$
{regrouping}
3: **if** $rank_{old} < rank$ **then**
{Combine Groups}
4: **for** $j = 0$ to $P/(2 \cdot Q) - 1$ **do**
5: Group$_j$ = Merge(Group$_{2j}$, Group$_{2j+1}$)
6: **end for**
7: **else if** $rank_{old} > rank$ **then**
{Partition Group}
8: **for** $j = P/Q - 1$ downto 0 **do**
9: (Group$_{2j}$, Group$_{2j+1}$) = Partition(Group$_j$)
10: **end for**
11: **end if**

Algorithm 4 Group Level Scheduling for the Manager of Group$_i$

{Update workload indicator}
1: $\Delta W = \sum_{\tilde{T} \in LCL_i} w_{\tilde{T}}$
2: $WI_i = WI_i - \Delta W$
{Update precedence relations}
3: **for all** $T \in \{$successors of $\tilde{T}$, $\forall \tilde{T} \in LCL_i\}$ **do**
4: $\boxed{d_T = d_T - 1}$
5: **if** $d_T = 0$ **then**
6: $\boxed{GRL = GRL \cup \{T\}; GL = GL \backslash \{T\}}$
7: **end if**
8: **end for**
{Batch task allocation}
9: **if** LRL_i is not full **then**
10: $\boxed{S' \Leftarrow \text{fetch } m \text{ tasks from } GRL, \text{ if any}}$
11: **if** $\sum_{T \in S'} w_T < \Delta W$ or $WI_i < \delta_M$ **then**
12: $\boxed{\text{Fetch more tasks from } GRL \text{ to } S',}$ $\boxed{\text{so that } \sum_{T \in S'} w_T \approx \Delta W + \delta_M}$
13: **end if**
14: $LRL_i = LRL_i \cup \{S'\}$
15: $WI_i = WI_i + \sum_{T \in S'} w_T$
16: **end if**

The workers schedule tasks assigned by their manager (Algorithm 5). This algorithm is a straightforward self-scheduling, where each idle worker fetches a task from LRL_i and then puts the tasks to LCL_i after execution. Although LRL_i and LCL_i are shared by the manager and worker threads in the same group, no worker accesses any variables shared between groups.

4 Experiments

4.1 Computing Facilities

The Sun UltraSPARC T2 (Niagara 2) platform was a Sunfire T2000 server with a Sun UltraSPARC T2 multithreading processor [16]. UltraSPARC T2 has 8 hardware multithreading cores, each running at 1.4 GHz. In addition, each core supports up to 8 hardware threads with 2 shared pipelines. Thus, there are 64 hardware threads. Each

Algorithm 5 Within-Group Level Scheduling for a Worker of Group$_i$

Input:
Output:
1: Fetch T from LRL_i
2: **if** $T \neq \emptyset$ **then**
3: Execute task T
4: $LCL_i = LCL_i \cup \{T\}$
5: **end if**

core has its own L1 cache shared by the threads within a core. The L2 cache size is 4 MB, shared by all hardware threads. The platform had 32 GB DDR2 memory shared by all the cores. The operating system was Sun Solaris 11 and we used Sun Studio CC with Level 4 optimization (-xO4) to compile the code.

4.2 Baseline

To compare the performance of the proposed method, we performed DAG structured computations using Charm++ [6] Cilk [5] and OpenMP [14]. In addition, we implemented *three* typical schedulers called `Cent ded`, `Dist shared` and `Steal`, respectively. We evaluated the baseline methods along with the proposed scheduler using the same input task dependency graphs.

(a) Scheduling DAG structured computations using Charm++ (`Charm++`): Charm++ runtime system employs a phase-based dynamic load balancing scheme facilitated by virtualization, where the computation is monitored for load imbalance and computation objects (tasks) are migrated between phases by message passing to restore balance. Given a task dependency graph, each task is packaged as an object called *chore*. Initially, all tasks with dependency degree equal to 0 are submitted to the runtime system. When a task completes, it reduces the dependency degree of the successors. Any successors with reduced dependency degree equal to 0 are submitted to the runtime system for scheduling.

(b) Scheduling DAG structured computations using Cilk (`Cilk`): This baseline scheduler performed work stealing based scheduling using the Cilk runtime system. Unlike the proposed 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 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 dependency degree equal to 0. We used spinlocks for the dependency degrees to prevent concurrent write.

(c) Scheduling DAG structured computation using OpenMP (`OpenMP`): This baseline initially inserted all tasks with dependency degree equal to 0 into a ready queue. Then, using the OpenMP pragma directives, we created threads to execute these tasks in parallel. During executing the tasks in the ready queue, we inserted new ready-to-execute tasks into another ready queue for parallel execution in the next iteration. Note

that the number of tasks in the ready queue can be much greater than the number of cores. We let the OpenMP runtime system to dynamically schedule tasks to underutilized cores.

(d) Centralized scheduling with dedicated core (`Cent ded`): This scheduling method bound each thread to a separate core. One thread was the manager and the rest were workers. The input DAG was *local* to the manager. Each worker had a ready task list *shared* with the scheduler thread. There was also a completed task list *shared* by all the threads. The manager was also in charge of all the activities related to scheduling and the workers executed assigned tasks only. Pthread mutex locks were used for the ready task lists and completed task list.

(e) 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 list. 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 list. Then, the scheduler in each thread updated the dependency degree of the successors of tasks in the completed task list, and fetched the tasks with dependency degree equal to 0 for allocation. Pthreads mutex locks were used for the global task list and the ready task list.

(f) Task stealing based scheduling with distributed ready task list (`Steal`): 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.

4.3 Datasets and Data Layout

We experimented with both synthetic and real datasets to evaluate the performance of the proposed scheduler. For the synthetic datasets, we varied the task dependency graphs so that we can evaluate our scheduling method using task dependency graphs with various graph topologies, sizes, task workload, task types and accuracies in estimating task weights. For the real datasets, we used task dependency graphs for blocked matrix multiplication (BMM), LU and Cholesky decomposition. In addition, we also

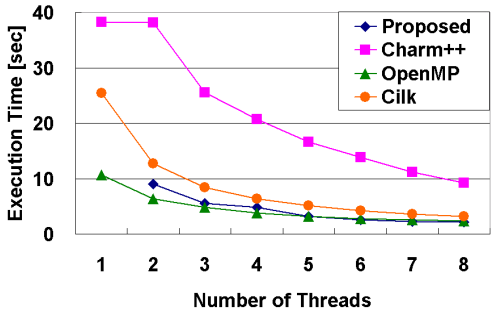
used the task dependency graph for exact inference, a classic problem in artificial intelligence, where each task consists of data intensive computations between a set of probabilistic distribution tables (also known as *potential tables*) involving both regular and irregular data accesses [19].

We used the following data layout in the experiments: The task dependency graph was stored as an array in the memory, where each element represents a task with a task ID, weight, number of successors, a pointer to the successor array and a pointer to the task meta data. Thus, each element took 32 Bytes, regardless of what the task consisted of. The task meta data was the data used for task execution. For LU decomposition, the task meta data is a matrix block; for exact inference, it is a set of potential tables. The lists used by the scheduler, such as GRL, LRLs and LCLs, were circular lists, each having a head and a tail pointer. In case any list was full during scheduling, new elements were inserted on-the-fly.

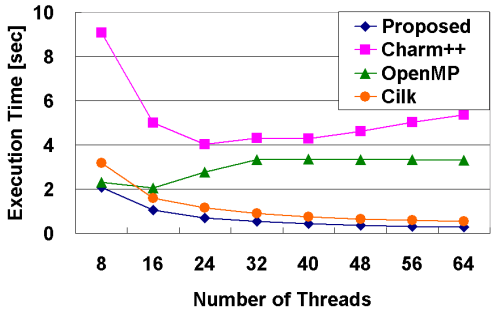
4.4 Results

We compared the performance of the proposed scheduling method with two state-of-the-art parallel programming systems i.e. Charm++[6], Cilk [5] and OpenMP [14]. We used a task dependency graph for which the structure was a random DAG with 10,000 tasks and there was an average of 8 successors for each task. Each task was a dense operation, e.g., multiplication of two 30×30 matrices. For each scheduling method, we varied the number of available threads, so that we could observe the achieved scalability. The results are shown in Figure 5. Similar results were observed for other tasks. Given the number of available threads, we repeated the experiments five times. The results were consistent; the standard deviation of the results were almost within 5% of the execution time. In Figure 5(a), all the methods exhibited scalability, though `Charm++` showed relatively large overhead. A reason for the significant overhead of `Charm++` compared with other methods is that Charm++ runtime system employs message passing based mechanism to migrate tasks for load balancing (see Section 4.2). This increased the amount of data transferring on the system bus. Note that the proposed method required at least two threads to form a group. In Figure 5(b) where more threads were used, our proposed method still showed good scalability; while the performance of the `OpenMP` and `Charm++` degraded significantly. As the number of threads increased, the `Charm+` required frequent message passing based task migration to balance the workload. This stessed the system bus and caused the performance degradation. The performance of `OpenMP` degraded as the number of threads increase, because it can only schedule the tasks in the ready queue (see Section 4.2), which limits the parallelism. `Cilk` showed scalability close to the proposed method, but the execution time was higher.

We compared the proposed scheduling method with three typical schedulers, a centralized scheduler, a distributed scheduler and a task-stealing based scheduler addressed in Section 4.2. We used the same dataset as in the previous experiment, but the matrix sizes were 50×50 (*large*) and 10×10 (*small*) for Figures 6(a) and (b), respectively. We normalized the throughput of each experiment for comparison. We divided the throughput of each experiment by the throughput of the proposed method using 8 threads. The results exhibited *inconsistencies* for the two baseline methods: `Cent ded` achieved



(a) Scalability with respect to 1-8 threads

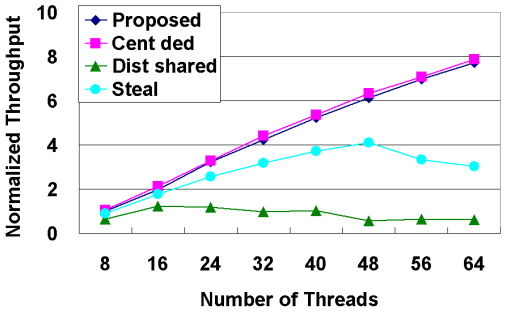


(b) Scalability with respect to 8-64 threads

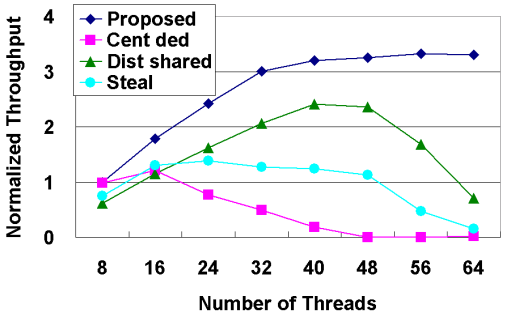
Fig. 5. Comparison of average execution time with existing parallel programming systems.

much better performance than `Dist shared` with respect to large tasks, but significantly poorer performance with respect to small tasks. Such inconsistencies implied that the impact of the input task dependency graphs on scheduling performance can be significant. An explanation to this observation is that the large tasks required more resources for task execution, but `Dist shared` dedicated many threads to scheduling, which limits the resources for task execution. In addition, many schedulers frequently accessing shared data led to significant overheads due to coordination. Thus, the throughput decreased for `Dist shared` as the number of threads increased. When scheduling small tasks, the workers completed the assigned tasks quickly, but the single scheduler of `Cent ded` could not process the completed tasks and allocate new tasks to all the workers in time. Therefore, `Dist shared` achieved higher throughput than `Cent ded` in this case. When scheduling large tasks, the proposed method dynamically merged all the groups and therefore became the same as `Cent ded` (Figure 6(a)). When scheduling small tasks, the proposed scheduler became a distributed scheduler by keeping each core (8 threads) as a group. Compared with `Dist shared`, 8 threads per group led to the best throughput (Figure 6(b)). `Steal` exhibited increasing throughput with respect to the number of threads for large tasks. However, the performance tapered off when more than 48 threads were used. One reason for this observation is that, as the

number of thread increases, the chance of stealing tasks also increases. Since a thread must access shared variables when stealing tasks, the coordination overhead increased accordingly. For small tasks, `Steal` showed limited performance compared with the proposed method. As the number of threads increases, the throughput was adversely affected. The proposed method dynamically changed the group size and merged all the groups for the large tasks. Thus, the proposed method becomes `Cent ded` except for the overhead of grouping. The proposed scheduler kept each core (8 threads) as a group when scheduling the small tasks. Thus, the proposed method achieved almost the same performance as `Cent ded` in Figure 6(a) and the best performance in Figure 6(b).



(a) Performance with respect to large tasks (50×50 matrix multiplication for each task)

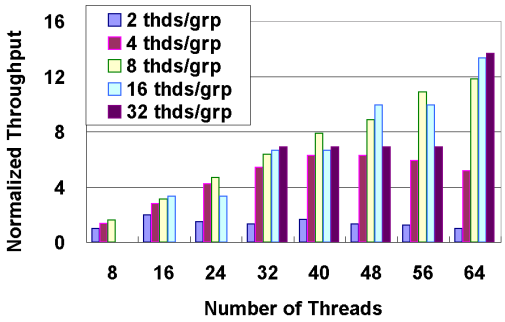


(b) Performance with respect to small tasks (10×10 matrix multiplication for each task)

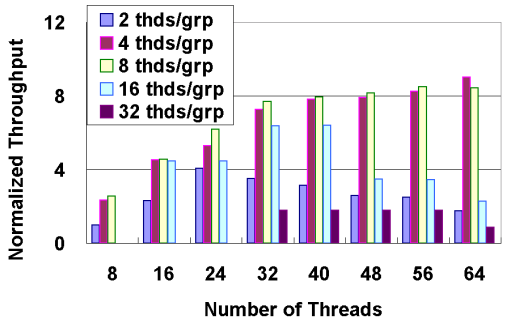
Fig. 6. Comparison with baseline scheduling methods using task graphs of various task sizes.

We experimentally show the importance of adapting the group size to the task dependency graphs in Figure 7. In this experiment, we modified the proposed scheduler by fixing the group size. For each fixed group size, we used the same dataset in the previous experiment and measured the performance as the number of threads increases. According to Figure 7, larger group size led to better performance for large tasks; while for the small tasks, the best performance was achieved when the group size was 4 or 8.

Since the optimized group size varied according to the input task dependency graphs, it is necessary to adapt the group size to the input task dependency graph.



(a) Performance with respect to large tasks (50×50 matrix multiplication for each task)

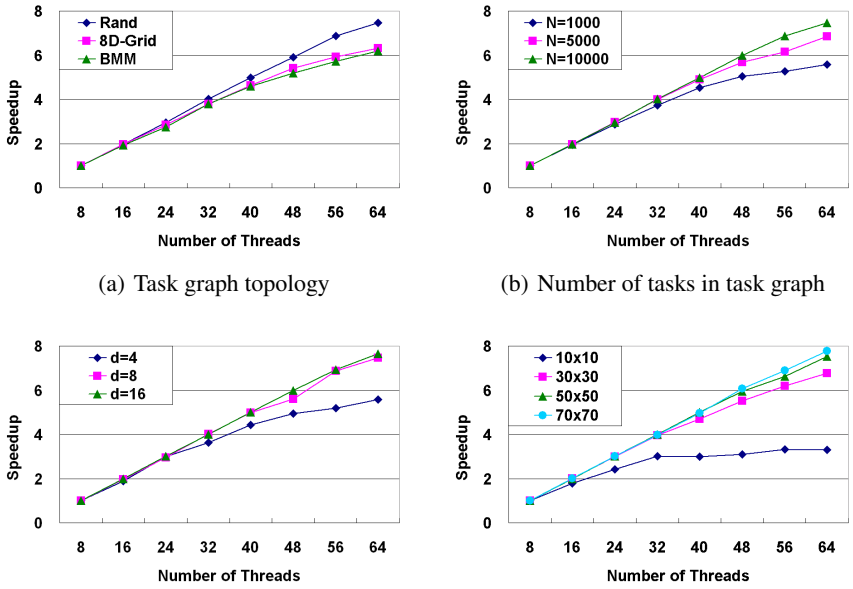


(b) Performance with respect to small tasks (10×10 matrix multiplication for each task)

Fig. 7. Performance achieved by the proposed method *without* dynamically adjusting the scheduler group size (number of threads per group, *thds/grp*) with respect to task graphs of various task sizes.

In Figure 8, we illustrated the impact of various properties of task dependency graphs on the performance of the proposed scheduler. We studied the impact of the topology of the graph structure, the number of tasks in the graph, the number of successors and the size of the tasks. We modified these parameters of the dataset used in the previous experiments. The topologies used in Figure 8(a) included a random graph (Rand), a 8-dimensional grid graph (8D-grid) and the task graph of blocked matrix multiplication (BMM). Note that we only used the topology of the task dependency graph for BMM in this experiment. Each task in the graph was replaced by a matrix multiplication. We evaluate the full BMM as a real-life problem in Figure 13. According to the results, for most of the scenarios, the proposed scheduler achieved almost linear speedup. Note that the speedup for 10×10 task size was relatively lower than others. This was because synchronization in scheduling was relatively large for the task de-

pendency graph with small task sizes. Note that we used the speedup as the metric in Figure 8. By speedup, we mean the serial execution time over the parallel execution time, when all the parameters of the task dependency graph are given.



(a) Task graph topology

(b) Number of tasks in task graph

(c) Number of successors of each task

(d) Task size

Fig. 8. Impact of various properties of task dependency graphs on speedup achieved by the proposed method.

In Figure 9, we investigated the impact of task types on scheduling performance. The computation intensive tasks (Computation) were matrix multiplications, for which the complexity was $O(N^3)$, assuming the matrix size was $N \times N$. In our experiments, we had $N = 50$. The memory access intensive tasks (Mem Access) summed an array of N^2 elements using $O(N^2)$ time. For the last task type (Mixed), we let all the tasks with an even ID perform matrix multiplication and the rest sum an array. We achieved speedup with respect to all task types. The speedup for memory access intensive tasks was relatively lower due to the latency of memory access.

Figure 10 reflects the efficiency of the proposed scheduler. We measured the execution time of each thread to check if the workload was evenly distributed, and normalized the execution time of each thread for the sake of comparison. The underlying graph was a random graph. We also limited the number of available cores in this experiment to observe the load balance in various scenarios. Each core had 8 threads. As the number of cores increased, there was a minor imbalance across the threads. However, the percentage of the imbalanced work was very small compared with the entire execution time.

For real applications, it is generally difficult to estimate the task weights accurately. To study the impact of the error in estimated task weight, we intentionally added noise to the estimated task weight in our experiments. We included noise that added 5%, 10% and 15% of the real task execution time. The noise was drawn from uniform distribution using the POSIX math library. According to the results in Figure 11, the impact was not significant.

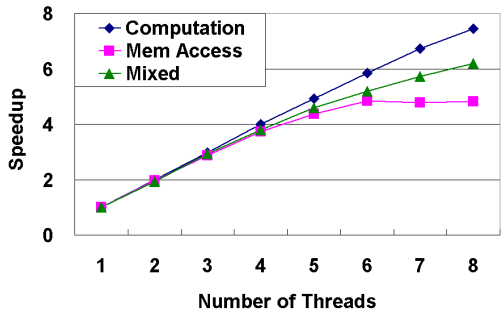


Fig. 9. Performance of the proposed method with respect to computation intensive tasks, memory access intensive tasks and the mix.

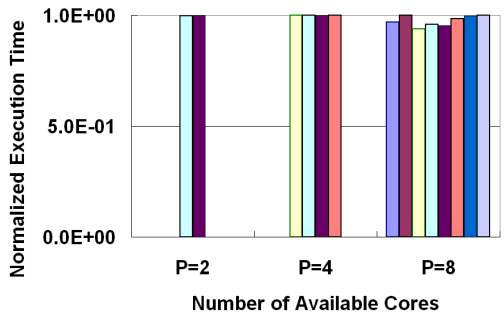


Fig. 10. Load balance achieved by the proposed method with respect to various number of available cores.

In Figure 12, we investigated the overhead of the proposed scheduler. Using the same dataset used in the previous experiment, we first performed hierarchical scheduling and recorded to which thread a task was allocated. According to such allocation information, we performed static scheduling to eliminate the overhead due to the proposed dynamic scheduler. We illustrate the execution time in Figure 12. Unlike the previous experiments, we show the results with respect to execution time to compare

both the scalability and the scheduling overhead for a given number of threads. As we can see, the overhead due to dynamic scheduling was very small.

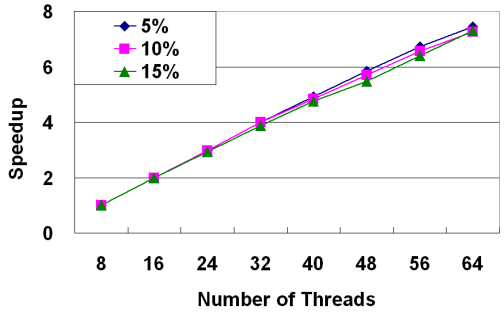


Fig. 11. Impact of the error in estimated task weight on speedup achieved by the proposed method.

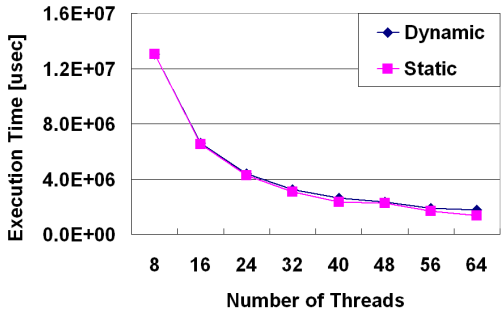


Fig. 12. Overhead of the proposed scheduling method.

The above experiments were conducted using synthetic datasets, so that we could control the parameters and then study the impact of various factors to the scheduling performance. We achieved consistent results for real application datasets too. In Figure 13, we constructed the task dependency graph according to blocked matrix multiplication (BMM), LU decomposition and Cholesky decomposition [17]. For the BMM, we used a matrix of size 600×600 with block size 50×50. The total number of tasks was 3312. For both the LU and Cholesky decomposition, the matrix size was 1000×1000 and block size was 50×50. The total number of tasks was 2870. In Figure 14, we applied the proposed scheduler for parallel exact inference [19]. The task dependency graph for this problem had 1023 nodes and each node had a potential table of 4096 entries. We manually partitioned the potential tables at different sizes and therefore had

three datasets. The sizes of the partitioned potential table were 4096, 1024 and 256 for large, mediate and small tasks, respectively. The proposed scheduler worked well for all the real applications. Note that we used the metric speedup instead of *absolute* execution time or throughput. This is because the absolute performance requires optimization of both the tasks and the scheduler. We only focused on scheduler design in this paper, therefore we used the metric of speedup.

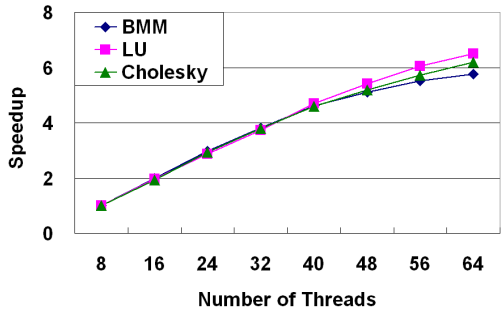


Fig. 13. Performance of the proposed scheduler for real applications.

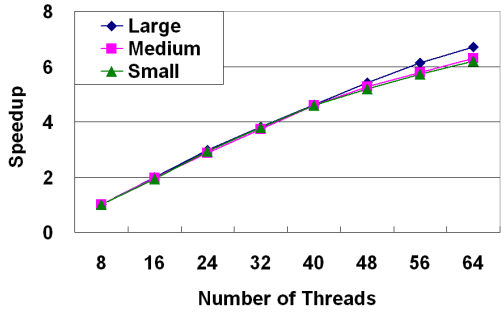


Fig. 14. Performance of the proposed scheduler for exact inference.

5 Conclusion

We proposed a hierarchical scheduling scheme for manycore processors. In our method, we divided the threads into groups, each having a manager to perform scheduling at the group level and several workers to perform self-scheduling for the tasks assigned by the manager. A supermanager was used to dynamically adjust the group size, so that the

scheduler could adapt to the input task dependency graph. We analyzed the proposed method and demonstrated its advantages for manycore architectures. The experimental results on the Sun UltraSPARC T2 processors were encouraging, compared with typical baseline schedulers and existing parallel programming systems. In the future, we plan to study data layout for high throughput processors to efficiently use the data cache of the UltraSPARC processors, since the L2 cache is no more than 4 MB, shared by up to 64 hardware threads. We would also like to explore the heuristics for assigning tasks of various types to a core. For example, interleaving the computationally intensive tasks with memory access intensive tasks may improve the overall performance.

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. D. Bader. High-performance algorithm engineering for large-scale graph problems and computational biology. In *4th International Workshop on Efficient and Experimental Algorithms*, pages 16–21, 2005.
4. A. Benoit, M. Hakem, and Y. Robert. Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems. *Parallel Computing*, 35(2):83–108, 2009.
5. 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.
6. Charm++ programming system. http://charm.cs.uiuc.edu/research/charm/.
7. E. G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, New York, NY, 1976.
8. M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
9. Intel Threading Building Blocks. http://www.threadingbuldingblocks.org/.
10. V. Karamcheti and A. Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–17, 1998.
11. J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. Technical report, 2009.
12. Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.
13. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Systems Journal*, 45(1):85–102, 2006.
14. OpenMP Application Programming Interface. http://www.openmp.org/.
15. 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.
16. D. Sheahan. Developing and tuning applications on UltraSPARC T1 chip multithreading systems. Technical report, 2007.

17. 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.
18. G. Tan, V. C. Sreedhar, and G. R. Gao. Analysis and performance results of computing betwenness centrality on ibm cyclops64. *Journa of Supercomputing*, 2009.
19. Y. Xia, X. Feng, and V. K. Prasanna. Parallel evidence propagation on multicore processors. In *The 10th International Conference on Parallel Computing Technologies*, pages 377–391, 2009.
20. 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.
21. W. Zhu, P. Thulasiraman, R. K. Thulasiram, and G. R. Gao. Exploring financial applications on many-core-on-a-chip architecture: A first experiment. In *Frontiers of High Performance Computing and Networking*, pages 221–230, 2006.