CAB: Cache Aware Bi-tier Task-stealing in Multi-socket Multi-core Architecture

Quan Chen* , Zhiyi Huang†, Minyi Guo*, Jingyu Zhou*
*Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China
chen-quan@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn,zhou-jy@cs.sjtu.edu.cn
†Department of Computer Science, University of Otago, Dunedin, New Zealand
hzy@cs.otago.ac.nz

***Abstract*—Modern multi-core computers often adopt a multi-socket multi-core architecture with shared caches in each socket. However, traditional task-stealing schedulers tend to pollute the shared cache and incur more cache misses due to their random stealing. To relieve this problem, this paper proposes a Cache Aware Bi-tier (CAB) task-stealing scheduler, which improves the performance of memory-bound applications by reducing memory footprint and cache misses of tasks running inside the same CPU socket. CAB uses an automatic partitioning method to divide an execution Directed Acyclic Graph (DAG) into the inter-socket tier and the intra-socket tier. Tasks generated in the inter-socket tier are scheduled across sockets, while tasks generated in the intra-socket tier are scheduled within the same socket. Experimental results show that CAB can improve the performance of memory-bound applications up to 68.7% compared with the traditional task-stealing.**

***Keywords*-Multi-socket Multi-core architecture, Cache aware, Task-stealing, Work-stealing, Cilk**

I. Introduction

Multi-core processors have become mainstream as chip manufacturers like AMD and Intel keep producing new CPU chips with more cores. Modern multi-core computers often use a Multi-Socket Multi-Core (MSMC) architecture in order to obtain more computing power. In the MSMC architecture, multiple multi-core chips share the main memory (RAM), while the cores in the same CPU chip (also referred as CPU socket in this paper) share the L2 or L3 caches. This architecture is popular and will continue to be a dominating architecture for high performance computing in future.

Despite the rapid development of the multi-core technology, a lot of software are yet to be parallelized to utilize the power of multi-core computers. This need has promoted the development of parallel programming environments. Currently, popular parallel programming environments can be classified into two groups in terms of task scheduling. The first group is based on manual task scheduling, where programmers need to manually arrange tasks for each thread or processor for optimal performance. Pthread [1], MPI [2] and Maotai [3] are examples of this group. The drawback of this group is that manual task scheduling is often burdensome for developing applications.

The second group is based on automatic task scheduling. In these programming environments, programmers can specify and generate tasks at runtime. Parallelism in programs is mostly expressed as tasks that are scheduled automatically among executing threads. Examples of this group are Cilk [4], Cilk++ [5], TBB [6], OpenMP [7], Java's fork-join framework [8], X10 [9] and XWS [10]. This feature of automatic task scheduling enables convenient expression of dynamic tasks and automatic load balancing.

In programming environments with automatic task scheduling, the execution of a parallel program can be represented by a task graph, which is a Directed Acyclic Graph (DAG) $G = (V, E)$, where V is a set of nodes, and E is a set of directed edges [11]. A node n_i in a DAG represents a task (i.e., a set of instructions) that must be executed sequentially without preemption. The edges in a DAG, denoted by (n_j, n_k), correspond to the dependence relationship among the nodes.

Most DAG-based automatic task scheduling algorithms, such as task-stealing (also known as work-stealing[1]) [12] and task-sharing [7], schedule tasks onto processors randomly. This randomness in task scheduling causes *Task Relocation Incurred Cache Interference* (TRICI) syndrome in the MSMC architecture, which is depicted as follows.

Suppose there are three tasks γ_1, γ_2 and γ_3 to be executed in an MSMC architecture. γ_1 and γ_2 share data, but they share nothing with γ_3. If γ_1 and γ_2 are scheduled to the cores of the same CPU socket, the shared data are loaded into the shared caches (e.g., L3 cache) only once but can be accessed by both tasks. However, this data sharing is not respected by traditional task scheduling algorithms due to their randomness in selecting cores for the tasks. As a result, the task schedulers could move γ_1 or γ_2 to a core in a different socket, where γ_3 is being executed. Thus γ_1 and γ_2 cannot share cache and have to load data into their own caches separately.

The above random scheduling causes two problems. First, it increases cache misses. Suppose γ_2 is scheduled to the socket of γ_3. γ_2 cannot use the data already loaded into the caches by γ_1 and have to read data from the main memory. Second, the random scheduling enlarges the memory

* Quan Chen was a visiting PhD student at the University of Otago during the course of this research.

[1] we use "task-stealing" in this paper for the consistency of terms.

DOI 10.1109/ICPP.2011.32

footprint of the sockets. Since γ_2 and γ_3 share nothing but run in the same socket, the memory footprint of the socket will become larger. This increases the chance of cache misses and causes performance degradation, because γ_2 may pollute the cache entries for γ_3 due to conflicts or limited cache capacity. Such a degrading performance problem in the MSMC architecture is called the TRICI syndrome in this paper, which is caused by the random task scheduling policy.

In order to relieve the TRICI syndrome, we propose a *Cache Aware Bi-tier* (CAB) policy for the task-stealing scheduler. In traditional task-stealing [4], whenever a worker's task pool becomes empty, the worker will randomly choose a victim worker and steal a task from it. Unfortunately, such a task-stealing policy suffers from the TRICI syndrome due to the random stealing. CAB addresses the syndrome by scheduling tasks that share data onto the cores in the same socket in order for them to share data in caches. It divides the execution DAG of a program into two tiers: inter-socket tier and intra-socket tier. Tasks in the inter-socket tier are scheduled across the sockets, while tasks in the intra-socket tier are scheduled within the same socket. CAB can automatically and optimally partition the execution DAG into the two tiers according to the input data size of an application, the data cache size, and the number of sockets.

The contributions of this paper are three-fold.

- The CAB task-stealing significantly relieves the TRICI syndrome by scheduling tasks with shared data onto cores of the same socket.
- CAB presents an automatic partitioning method to divide a DAG into two tiers so that tasks in different tiers are generated and scheduled in different ways.
- The experiment shows that CAB can significantly achieve a performance gain of up to 68.7% for memory-bound applications.

The rest of this paper is organized as follows. Section II introduces the background and motivation of CAB. Section III presents the DAG partitioning method and the CAB task-stealing algorithm. Section IV gives the implementation details of CAB. Section V shows the experimental results and evaluates the performance. Section VI discusses related work. Section VII draws conclusions and sheds light on future work.

II. Background and Motivation

There are two main classes of automatic task scheduling algorithms: task-sharing and task-stealing. In task-sharing, workers push new tasks into a central task pool when they are generated. Tasks are popped out from the task pool when workers are free to execute them. The push and pop operations need to lock the central task pool, which often causes serious lock contention.

Task-stealing, on the other hand, uses a task pool for each worker. Most often each worker pushes and pops tasks to its own task pool without locking. Only when a worker's task pool is empty, should it try to steal tasks from other workers with locking. Since there are multiple task pools for stealing, the lock contention is much lower than task-sharing even at task steals. Therefore, task-stealing performs even better than task-sharing when the number of workers is increasing.

However, as mentioned before, task-stealing still suffers from the TRICI syndrome. Let us take the *five-point heat* program as an example, which simulates the heat distribution of a metal plate. In the program, the metal plate is divided into points of a two-dimensional grid. At each simulation step, the points in row r are computed based on the points in rows r, $r-1$ and $r+1$ of the previous step.

Given a $10*10$ matrix as the input data with the data type *double* (rows 0, 9 and columns 0, 9 are boundary data, and the real grid to be computed is an $8*8$ matrix). In the parallel heat program, the heat procedure recursively generates two sub-tasks until the data set for each task is small enough. Fig. 1 shows the execution DAG of the heat program. The input data is recursively divided into two parts until each of the leaf tasks in the DAG only processes two rows.

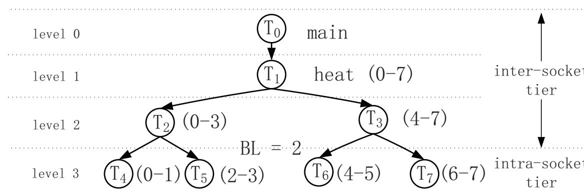


Figure 1. Execution DAG of five-point heat program

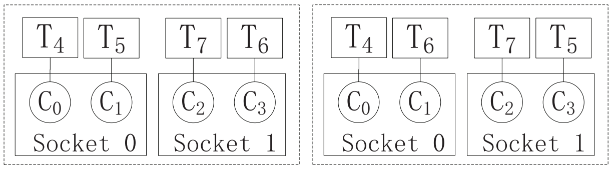
According to the dependence relationship, tasks in the DAG can be divided into levels. If a task γ in level i generates a task β, then β is in level $i+1$. The task that executes the "main" procedure is in level 0 and it is the initial task in the DAG.

Note that, in Fig. 1, only the leaf tasks (i.e., T_4, T_5, T_6, T_7) touch data, while all the other tasks in levels 0, 1, and 2 only divide the input data into two parts recursively.

Suppose this parallel heat program is executed on a dual-socket dual-core architecture with a hypothetical shared cache size of 480 bytes[2] in each socket.

In the above scenario, if the leaf tasks T_4, T_5, T_6 and T_7 are ideally scheduled in the way as shown in Fig. 2(a), data in the shared cache can be re-used and thus cache misses can be reduced. In Fig. 2(a), tasks running on the cores of the same socket (e.g., T_4 and T_5) share two rows of input data. The two tasks in each socket only need to read six rows into the shared cache from the main memory altogether, i.e., $6*10*8 = 480$ bytes. This data set size can fit into the shared cache of a socket. The overall memory footprint

[2]We use this hypothetical small cache size for ease of explanation, but it does not affect our analysis since input data will be proportionally larger for a real cache size.



(a) One possible scheduling (b) Another possible scheduling

Figure 2. Two possible scheduling of tasks of five-point heat running on a dual-socket dual-core architecture. The first scheduling can gain performance improvement from cache-sharing and reduction of memory footprint, because T_4 and T_5, T_6 and T_7 have shared data.

of the system is $2 * 480 = 960$ bytes if all four tasks are considered.

However, for traditional task-stealing, since it distributes tasks randomly, the leaf tasks T_4, T_5, T_6 and T_7 are very likely scheduled in the way as shown in Fig. 2(b), where tasks running on the cores of the same socket (e.g., T_4 and T_6) do not share any data. In this case, every task needs to access the main memory and reads four rows of the matrix into the cache. Because the two tasks in each socket need to read $2 * 4 * 10 * 8 = 680$ bytes, the data size exceeds the capacity of the shared cache of each socket, which leads to more capacity cache misses and increases the chances for conflict cache misses. Furthermore, the overall memory footprint of the system is $2 * 680 = 1280$ bytes, which leads to more compulsory cache misses.

In order to relieve the TRICI syndrome and schedule tasks in the same way as in Fig. 2(a), we propose the CAB task-stealing, which partitions the execution DAG into two tiers: inter-socket tier and intra-socket tier. Tasks in the inter-socket tier are scheduled across sockets, while tasks in the intra-socket tier are scheduled within the same socket. For example, in Fig. 1, tasks in levels 0-2 are in the inter-socket tier and tasks in level 3 are in the intra-socket tier. The tasks in level 2, the boundary of the two tiers, are called leaf inter-socket tasks. In CAB, the intra-socket tasks such as T_4 and T_5 are bound to the same socket. Since the intra-socket tasks generated by the same leaf inter-socket task often share data in real applications, their binding to the same socket in CAB can enforce the scheduling in Fig. 2(a) and results in fewer cache misses.

III. Cache Aware Bi-tier Task-stealing

This section presents CAB, a Cache Aware Bi-tier task-stealing scheduler. First, we give an overview of CAB. Then we introduce an automatic partitioning method for dividing the execution DAG into two tiers. Third, we present the CAB task generation algorithm, followed by the CAB task-stealing algorithm. Lastly, we discuss the theoretical time and space bounds of CAB.

A. *Overview of CAB*

CAB divides the workers into squads corresponding to the MSMC architecture. A *squad* is a group of workers running in the same socket. Each squad has a head worker. For an MSMC architecture that has M sockets with N cores each, CAB launches $M \times N$ workers (i.e. threads) to work on the DAG in parallel. The workers are divided into M squads with N workers in each squad. Each worker is affiliated with a hardware core, while each squad is affiliated with a socket. Fig. 3 depicts the relationship among cores, sockets, workers, and squads.

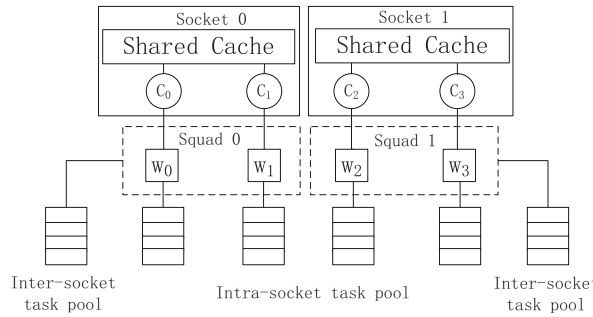


Figure 3. Relationship among cores, sockets, workers, and squads in a Dual-socket Dual-core architecture. Workers that run on the cores of the same socket are grouped into a squad. Each squad has an inter-socket task pool and each worker has an intra-socket task pool.

CAB adopts two types of task pools: inter-socket task pool and intra-socket task pool. A task pool is a double-ended queue (deque) that is used to store tasks. The inter-socket task pool is used to store tasks from the inter-socket tier of the DAG, and the intra-socket task pool stores tasks from the intra-socket tier. Each squad has one inter-socket task pool, and each worker has one intra-socket task pool, as shown in Fig. 3.

When CAB starts to execute a parallel program in an MSMC architecture, CAB uses an automatic DAG partitioning method (to be described shortly) to divide the execution DAG of the program into the inter-socket tier and the intra-socket tier. After the partitioning, CAB starts to execute the tasks by scheduling the inter-socket tasks and the intra-socket tasks based on the following stealing protocol.

A free worker in CAB first tries to obtain a task from its own intra-socket task pool. If the task pool is empty, it tries to steal a task from the intra-socket task pools of other workers in the same squad. If all the workers of the squad have empty task pools, the head worker of the squad tries to obtain a task from its own inter-socket task pool. If its inter-socket task pool is empty, the head worker of the squad tries to steal an inter-socket task from other squads.

The above protocol only allows the head worker to steal inter-socket tasks so that the lock contention of the inter-socket task pools is reduced. Also a squad is not allowed to execute more than one inter-socket task at the same time, because the data of different inter-socket tasks may pollute the shared caches if multiple inter-socket tasks are executed simultaneously in the same squad, which leads to more cache misses.

B. Automatic DAG partitioning method

As mentioned before, tasks in a DAG are divided into inter-socket tasks and intra-socket tasks according to their levels in the DAG. We compute a boundary level *BL* that partitions the DAG into the inter-socket tier (the upper tier) and the intra-socket tier (the lower tier). Tasks in the boundary level *BL* are called *leaf inter-socket tasks*. Since intra-socket tasks are scheduled within a squad, all the child tasks of a leaf inter-socket task are executed in the same socket.

However, finding the proper boundary level *BL* to partition the DAG optimally is challenging. If the intra-socket tier is too thick, the involved data for a squad can be too large to fit into the shared caches of the socket of the squad. On the other hand, if the intra-socket tier is too thin, the workload of a squad can be too small to get better balanced among the workers of the squad.

Therefore, we require that the DAG partitioning method satisfy three constraints. The first constraint is that there should be enough leaf inter-socket tasks to be distributed to the sockets. The second constraint is that the involved data size of a leaf inter-socket task is small enough to fit into the shared caches of a socket. The third constraint is that a leaf inter-socket task should be large enough to enable a squad to have sufficient intra-socket tasks. After careful study, we model these constraints using the following parameters: the input data size of the application, the number of sockets of the MSMC architecture, the shared cache size of each socket, and the branching degree of the DAG.

Note that, in the following model we assume that the program directly generates the task of the recursive divide-and-conquer procedure in the *main* procedure, which is the case for all our benchmarks. For example, in Fig. 1, the *main* procedure directly spawns the *heat* procedure that recursively spawns tasks executing itself until a cut-off point. However, if the recursive procedure is not directly generated by *main*, we need either manual adjustment of the *BL* value, or compiler support to adjust *BL* automatically. Further discussion on compiler support can be found in Section IV-D.

In the model, we suppose an M-socket N-core system has a shared cache size S_c for each socket and a recursive divide-and-conquer program has an input data of size S_d. We assume the program divides the input data into B parts each time sub-tasks are generated, i.e., the branching degree of the DAG of the recursive procedure is B. In this scenario, the boundary level *BL* should have B^{BL-1} leaf inter-socket tasks, since each task generates B sub-tasks for the next level and this is repeated $BL-1$ times until the boundary level, assuming levels are numbered from 0 and the level 0 starts with *main*.

Since there are M squads, in order to balance workload among squads, we should ensure that there are at least M leaf inter-socket tasks (the aforementioned first constraint). Therefore, *BL* needs to satisfy Eq. 1.

$$B^{BL-1} \geq M \tag{1}$$

Since the input data are often divided evenly among the leaf inter-socket tasks, the second constraint can be expressed with Eq. 2.

$$S_d/B^{BL-1} \leq S_c \tag{2}$$

From Eq. 1 and 2, we can deduce two conditions for selecting an appropriate value for *BL*, as shown in Eq. 3.

$$\begin{cases} BL \geq \log_B M + 1 \\ BL \geq \log_B (S_d/S_c) + 1 \end{cases} \tag{3}$$

From Eq. 3, we can select any *BL* that is large enough to satisfy the two inequations. But, unfortunately, if *BL* is too large, the number of the intra-socket tasks generated by a leaf inter-socket task will be too small, which leads to poor load balance within a squad. Therefore, we set *BL* to be the smallest value that satisfies both inequations in Eq. 3, as shown in Eq. 4.

$$BL = \max\{\lceil \log_B M + 1 \rceil, \lceil \log_B (S_d/S_c) + 1 \rceil\} \tag{4}$$

In our current implementation, we use a semi-automatic method to acquire parameters B, M, S_d, and S_c and then computes *BL* according to Eq. 4. Parameters M and S_c are automatically acquired from */proc/cpuinfo* by the runtime system, but S_d and B are provided through command line arguments. Section IV-D discusses how to automatically acquire the parameters by the compiler through program analysis.

In summary, CAB chooses *BL* to be the smallest value while ensuring that the data set of the leaf inter-socket tasks can fit into the shared cache and that the number of leaf inter-socket tasks is large enough so that there is at least one inter-socket task for each and every squad. Experimental results in Section V show that our automatic DAG partitioning method can find the optimal boundary level that enables the highest performance of the CAB scheduler.

C. CAB task generation

Tasks in the inter-socket tier and the intra-socket tier are generated with different policies in CAB. There are generally two policies for task generation: child-first and parent-first. In the child-first policy, a worker executes the child task immediately after it is generated, leaving the parent task for later execution or for stealing by other workers. For example, the MIT Cilk uses the child-first policy, which is called work-first in [4]. In the parent-first policy, a worker executes the parent task continually after spawning a child task, pushing the child task into the task pool. One such example is the help-first policy proposed in [13].

Both policies have advantages in different situations. The child-first policy works better than the parent-first policy when the execution DAG is deep. However, the parent-first policy works better when the initial DAG is shallow and the steals are frequent, since enough tasks can be quickly produced for free workers [13].

Because there are more steals needed in the inter-socket tier where the execution DAG is expanding initially, CAB adopts the parent-first policy in the inter-socket tier in order to distribute the leaf inter-socket tasks to squads as soon as possible. After a squad gets a leaf inter-socket task, it uses the child-first policy to generate the intra-socket tasks. Since the number of workers is small and the steals are not frequent in a squad, the child-first policy is more suitable for intra-socket tasks. Another advantage of the child-first policy is more space efficient.

D. CAB task-stealing

As mentioned before, a free worker follows the stealing protocol in Section III-A to obtain or steal tasks. According to the protocol, a squad is not allowed to execute more than one inter-socket task at the same time. In order to fulfill this requirement, CAB uses a boolean variable *busy_state* for each squad. *busy_state* indicates whether there is an inter-socket task running in the squad right now. When there is an inter-socket task running in a squad, *busy_state* of the squad is true. When a squad finishes its inter-socket task, its *busy_state* is set false. Only when *busy_state* is false, can the squad obtain or steal another inter-socket task. Algorithm I shows the detailed task-stealing algorithm that implements the stealing protocol.

Algorithm I
CAB TASK-STEALING ALGORITHM

Assumption: Suppose a worker w belongs to a squad ρ. The worker w is free and trying to get a new task.

Select victim from intra-socket tier:
Step 1: w tries to get a new task from its own task pool. If there is any task in the task pool, w obtains a task from the task pool and jumps to Step 7; otherwise, w goes to Step 2.
Step 2: w checks *busy_state* of ρ. If *busy_state* is true, w goes to Step 3; otherwise, if w is the head worker of ρ, w goes to Step 4, or else w goes back to Step 1.
Step 3: w tries to steal an intra-socket task from the workers in ρ. It chooses a victim worker w_{victim} within ρ randomly and then goes to Step 6.
Select victim from inter-socket tier:
Step 4: w tries to obtain an inter-socket task from ρ. If there is any task in the inter-socket task pool of ρ, w obtains a task from the task pool. Then w sets *busy_state* of ρ to be true and goes to Step 7. Otherwise, if the inter-socket task pool of ρ is empty, w goes to Step 5.
Step 5: w tries to steal an inter-socket task from other squads. w randomly chooses a victim squad ρ_{victim} and goes to Step 6.
Stealing from victim:
Step 6: (a) When the victim is a worker, if the task pool of w_{victim} is not empty, w pops a task from the task pool and then goes to Step 7, otherwise, w goes back to Step 2.
(b) When the victim is a squad, if the inter-socket task pool of ρ_{victim} is not empty, w pops a task from the task pool. Then w sets *busy_state* of ρ to be true and goes to Step 7. Otherwise, if the task pool of ρ_{victim} is empty, w goes to Step 5.
Step 7: w starts to execute the task.

E. Theoretical time and space bounds of CAB

We model the execution of a parallel program as an execution DAG $\mathcal{G}$. Each node in $\mathcal{G}$ represents a unit task, and each edge represents a dependence between tasks. Our following discussion is based on the time and space bounds of task-stealing proved in [12].

1) ***Time bound****:* For a DAG $\mathcal{G}$, the *work* $T_1(\mathcal{G})$ is the number of nodes in $\mathcal{G}$, and the critical-path length $T_\infty(\mathcal{G})$ is the number of nodes along the longest path from the start node to the end node.

Since CAB divides a DAG into two tiers and executes them differently, we need to divide a DAG into sub-DAGs using the leaf inter-socket tasks. Given a leaf inter-socket task γ, we use the notation $\mathcal{G}_\gamma$ to represent the subgraph rooted with γ, which includes the set of tasks that are generated from γ. Therefore, the total work of $\mathcal{G}$ is divided as in Eq. 5, where $\mathcal{G}_{inter}$ represents the subgraph of the inter-socket tier and K is the total number of the leaf inter-socket tasks at the boundary level BL.

$$T_1(\mathcal{G}) = T_1(\mathcal{G}_{inter}) + \sum_{i=1}^{K} T_1(\mathcal{G}_{\gamma_i}) \tag{5}$$

The execution time of $\mathcal{G}$ in an M-socket N-core architecture, $T_{M*N}(\mathcal{G})$, can be divided into two parts: the execution time of the inter-socket tier $T_{M*N}(\mathcal{G}_{inter})$ and the execution time of the intra-socket tier $T_{M*N}(\mathcal{G}_{intra})$. Even though the two parts can be overlapped, we use their sum to get the worst bound of $T_{M*N}(\mathcal{G})$ as shown in Eq. 6.

$$T_{M*N}(\mathcal{G}) = T_{M*N}(\mathcal{G}_{inter}) + T_{M*N}(\mathcal{G}_{intra}) \tag{6}$$

Since the inter-socket tier is executed by M head workers using task-stealing, according to the proof of [12], the execution time of $\mathcal{G}_{inter}$ is bounded by Eq. 7.

$$T_{M*N}(\mathcal{G}_{inter}) \leq \frac{T_1(\mathcal{G}_{inter})}{M} + T_\infty(\mathcal{G}_{inter}) \tag{7}$$

For the execution of the intra-socket tier, each $\mathcal{G}_{\gamma_i}$ is executed by N workers within a squad using task-stealing. Therefore, the execution time of $\mathcal{G}_{\gamma_i}$ is bounded by Eq. 8.

$$T_N(\mathcal{G}_{\gamma_i}) \leq \frac{T_1(\mathcal{G}_{\gamma_i})}{N} + T_\infty(\mathcal{G}_{\gamma_i}) \tag{8}$$

Since K leaf inter-socket tasks are scheduled among M squads using task-stealing, the execution time of the intra-socket tier is bounded by Eq. 9.

$$T_{M*N}(\mathcal{G}_{intra}) \leq \frac{\sum_{i=1}^{K} T_N(\mathcal{G}_{\gamma_i})}{M} + T_\infty(\mathcal{G}_{intra}) \tag{9}$$

Deducing from Eq. 8 and 9, we can get Eq. 10.

$$T_{M*N}(\mathcal{G}_{intra}) \leq \frac{\sum_{i=1}^{K} T_1(\mathcal{G}_{\gamma_i})}{M*N} + \frac{\sum_{i=1}^{K} T_\infty(\mathcal{G}_{\gamma_i})}{M} + T_\infty(\mathcal{G}_{intra}) \tag{10}$$

From Eq. 6, 7 and 10, $T_{M*N}(\mathcal{G})$ can be bounded as in Eq. 11.

$$T_{M*N}(\mathcal{G}) \leq \frac{T_1(\mathcal{G}_{inter})}{M} + T_\infty(\mathcal{G}_{inter}) + \frac{\sum_{i=1}^{K} T_1(\mathcal{G}_{\gamma_i})}{M*N} + \frac{\sum_{i=1}^{K} T_\infty(\mathcal{G}_{\gamma_i})}{M} + T_\infty(\mathcal{G}_{intra}) \tag{11}$$

After further tidying Eq. 11 up, we have Eq. 12.

$$T_{M*N}(\mathcal{G}) \leq \frac{T_1(\mathcal{G}_{inter})}{M} + \frac{\sum_{i=1}^{K} T_1(\mathcal{G}_{\gamma_i})}{M*N} + \frac{\sum_{i=1}^{K} T_\infty(\mathcal{G}_{\gamma_i})}{M} + T_\infty(\mathcal{G}) \tag{12}$$

According to Eq. 4, K is at most several times of M. Therefore, the third item in Eq. 12 can be merged with the fourth item. Finally, we have the time bound of $\mathcal{G}$ in an M-socket N-core architecture as shown in Eq. 13.

$$T_{M*N}(\mathcal{G}) = O(\frac{T_1(\mathcal{G}_{inter})}{M} + \frac{T_1(\mathcal{G}_{intra})}{M*N} + T_\infty(\mathcal{G})) \tag{13}$$

According to Eq. 13, the inter-socket tier is executed by only M head workers. However, in most recursively divide-and-conquer applications, only the leaf tasks in the DAG process input data, while the higher level tasks only divide the input data into smaller parts. Therefore, for a divide-and-conquer application, the main part of the execution time is spent by the leaf tasks, i.e., the intra-socket tasks. Our experiments show that the execution time of the inter-socket tier is often less than 5% of the overall execution time. Therefore, the time bound of Eq. 13 is very close to the traditional task-stealing schedulers such as Cilk for many divide-and-conquer applications.

*2) **Space bound analysis**:* According to the proof of [12], the space used by $\mathcal{G}$ in an M-socket N-core architecture is bounded by Eq. 14, where $S_1(\mathcal{G})$ denotes the space used by the serial execution of the program.

$$S_{M*N}(\mathcal{G}) \leq M*N*S_1(\mathcal{G}) \tag{14}$$

Eq. 14 assumes that there are at most $M*N$ workers expanding the DAG using the child-first policy. However, since CAB uses the parent-first policy to expand the inter-socket tier quickly, each of the leaf inter-socket tasks may use S_1 space in the worst case. Therefore, the space used by the CAB scheduler $S_{M*N}(\mathcal{G})$, can be bounded as in Eq. 15.

$$S_{M*N}(\mathcal{G}) \leq \max\{K*S_1(\mathcal{G}), M*N*S_1(\mathcal{G})\} \tag{15}$$

According to Eq. 4, the number of leaf inter-socket tasks, K, is not much larger than M, so the space bound has the same *O-notation* as the traditional task-stealing schedulers.

IV. Implementation of CAB

In this section, we present the implementation of CAB. First, we briefly introduce the MIT Cilk in which CAB is implemented. Then, we present the compiler support implemented for CAB, followed by the implementation of the CAB runtime system. Lastly, we discuss issues related to the implementation. Note that Cilk programs can run in our current implementation without any modifications.

A. Overview of MIT Cilk

MIT Cilk is one of the earliest parallel programming environments that implement task-stealing [4]. It extends C with three keywords: *cilk*, *spawn* and *sync* to declare parallelism in the program. *cilk* identifies a procedure as a *Cilk procedure*, *spawn* is used to generate a child task, and *sync* waits for all the child tasks that are generated by the current task to return. Only Cilk procedures can be invoked with *spawn* as a task.

MIT Cilk consists of a compiler and a scheduler. Cilk compiler, named as *cilk2c*, is a source-to-source translator that transforms a Cilk source into a C program. Once a task is generated, a task frame is created to store the information needed by the task and the scheduler. Cilk scheduler is a traditional task-stealing scheduler.

B. Compiler support of CAB

We have modified *cilk2c* to support two types of spawns for the inter-socket and intra-socket tasks respectively. At each spawn, we compare the DAG level of the current task with the boundary level BL. If the level is smaller than BL, we spawn the child task as an inter-socket task and follow the parent-first policy. Otherwise, we spawn the child task as in intra-socket task and follow the child-first policy.

We also modified *cilk2c* to support two types of *sync* for the inter-socket and intra-socket tasks. This is because we use the child-first policy to generate the intra-socket tasks but use the parent-first policy to generate the inter-socket tasks. We use the different *sync*s to manipulate different return behaviors.

We add into each task frame three variables: *level*, *parent* and *inter_counter*. *level* represents the level of the task in the execution DAG, *parent* is a pointer to the parent frame, and *inter_counter* is the number of outstanding child inter-socket tasks spawned by the task. For example, when a task γ generates a child inter-socket task γ_1, the *inter_counter* in the task frame of γ is increased by one. When γ_1 returns, through the *parent* pointer in its task frame, the *inter_counter* of γ is decreased by one. If γ's *inter_counter* equals zero, that means all the inter-socket tasks generated by γ have finished and the *sync* can be passed through.

C. CAB runtime system

For an M-socket N-core architecture, CAB launches $M*N$ workers and affiliates each worker to one individual core.

The ID of each worker is the same as the ID of the core on which the worker is running. CAB groups workers into squads according to their IDs. If the core i is in the socket j, the worker i is grouped into the squad j. In each squad, the worker with the smallest ID is the head worker.

CAB executes a parallel program following Algorithm II. Note that in the algorithm BL is set to 0 when there is only one socket in the architecture so that all tasks are generated as intra-socket tasks, which is the same as MIT Cilk.

Alogrithm II
CAB RUNTIME ALGORITHM

Assumption: Suppose an M-socket and N-core architecture and a worker w belongs to a squad ρ.

Global initiation:
Step 1: CAB launches $M * N$ workers and affiliates them to the corresponding cores.
Step 2: CAB calculates *BL*. If M equals 1, CAB sets *BL* to 0. Otherwise, CAB calculates *BL* according to Eq. 4.
Step 3: Worker 0 begins to execute the initial task, while all the other workers are trying to steal tasks.
Task scheduling: Assume worker w is executing task γ.
(a) γ **generates** γ_1**:** γ computes the level of γ_1. If γ_1 is in the inter-socket tier, it is generated as an inter-socket task. Then w pushes γ_1 into the inter-socket task pool of ρ and continues to execute γ. Otherwise, if γ_1 is in the intra-socket tier, it is generated as an intra-socket task, which is pushed into the task pool of w and to be executed by w immediately.
(b) γ **suspends:** w tries to obtain a task according to Algorithm I.
(c) γ **returns:** w returns the results of γ and sets *busy_state* of ρ to false if γ is an inter-socket task. Then w tries to get a task according to Algorithm I.
Termination: If all the tasks have finished, CAB terminates.

D. Discussion

Our current implementation uses a semi-automatic method to acquire the parameters for the calculation of *BL*. M and S_c are acquired automatically from the system, while the branching degree B and the input data size S_d of the recursive procedure are provided through command line.

It is desirable to fully automate the acquisition of all parameters used for the calculation of *BL*. Compiler support can be useful in finding the branching degree B by analyzing the source code based on the pattern of task generation, e.g., the keyword *spawn* in Cilk. However, the input data size S_d of a procedure can still be challenging for compilers, because the compiler needs to track the runtime calling chains and arguments of the procedure. Though it is easy to track the data size of arguments in many strongly typed languages, such a task is still challenging for the C language used by Cilk.

CAB scheduler also provides a new keyword *inter_spawn* to generate inter-socket tasks. This mechanism allows programmers to explicitly control the type of tasks and to fine-tune the program's behavior for the maximum efficiency. However, this method requires the programmer to manually modify the existing Cilk programs. According to our experiments, our semi-automatic method can achieve performance comparable to the well-tuned programs using this manual method.

Besides recursive task generating model, some programs use *flat task generating scheme*, where all the tasks are generated by a function at one time. For these programs, our CAB scheduler can also distribute tasks into inter-socket and intra-socket tiers for the maximum cache sharing in the same socket. Our preliminary experimental results show that programs can improve performance up to 25%. Due to space limit, this paper doesn't elaborate on this type of programs.

V. PERFORMANCE EVALUATION

In the performance evaluation, we use a Dell 16-core computer that has four AMD Quad-core Opteron 8380 processors (codenamed "Shanghai") running at 2.5 GHz. Each Quad-core socket has a 512K L2 cache for each core and a 6M L3 cache shared by all four cores. The computer has 16GB RAM and runs Linux 2.6.29. Accordingly, CAB sets up four squads with four workers each.

Table III
BENCHMARKS USED IN THE EXPERIMENTS

Name	Type(bound)	Description
Queens(20)	CPU	N-queens problem
Fft	CPU	Fast Fourier Transform
Ck	CPU	Rudimentary checkers
Cholesky	CPU	Cholesky decomposition
Heat	Memory	Five-point heat
Mergesort	Memory	Merge sort on $1024 * 1024$ numbers
SOR	Memory	2D Successive Over-Relaxation
GE	Memory	Gaussian elimination algorithm

Table III lists the benchmarks used in our experiments. The Cilk benchmarks run with CAB without any modification. All benchmarks are compiled with "cilkc -O2", which is based on gcc 4.4. For each test, every benchmark is run ten times and the average execution time is used as the result.

A. Performance of memory-bound applications

Fig. 4 shows the performance of four memory-bound applications with a $1024 * 1024$ matrix as input data. We can see that CAB can significantly improve the performance of memory-bound applications, with the performance gain ranging from 10% to 55%. Meanwhile, SOR has achieved up to 68.7% performance gain with CAB when the input data is a $512 * 512$ matrix (to be explained shortly in scalability part).

The performance gain of CAB is resulted from the relieved TRICI syndrome. Table IV shows that the L2 and L3 cache misses are prominently reduced in CAB compared with Cilk, this is because the data set used by a squad is often shared by the workers of the squad and can fit into the L3 cache in CAB. Cilk uses random scheduling that results in larger memory footprint, thus has more cache misses for workers inside the same socket. Due to the reduced cache misses, CAB performs significantly better than Cilk.

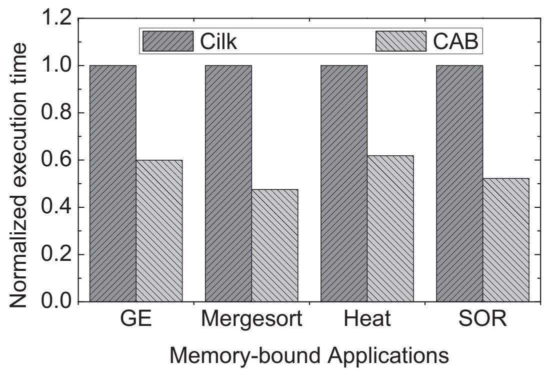


Figure 4. Normalized execution time of memory-bound applications with a 1k*1k matrix as input data.

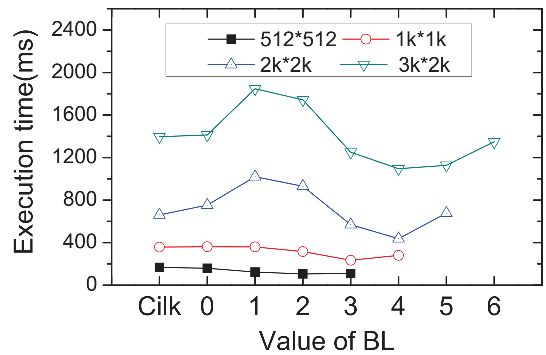


Figure 5. Impact of *BL* on performance of heat with different input data size. Our automatic DAG partitioning method can find the best value for *BL*.

Table IV
L2/L3 CACHE MISSES IN CAB AND CILK

	GE	Mergesort	Heat	SOR
L2 in Cilk	4203604	5717785	8457899	14134418
L2 in CAB	2617207	3448768	5577723	10863876
L3 in Cilk	1545310	1974802	2812464	5259771
L3 in CAB	180145	998605	755786	1256203

B. Effectiveness of automatic DAG partitioning method

In Section III-B, we have proposed a model to calculate the boundary level *BL* in order to partition the DAG. The model uses four parameters: B, M, S_d, and S_c, as shown in Eq. 4. This experiment uses *heat* to evaluate the effectiveness of the model, and we have verified that the model works for other applications as well.

We evaluate the performance of *heat* with all possible *BL* values. Since the *heat* program divides the input data into two parts each time sub-tasks are generated until the data size becomes 128 rows, there are fewer possible *BL* values when the input data sizes are small.

Fig. 5 shows the performance of *heat* with different input data sizes and all possible *BL* values. For example, for a $3k * 2k$ matrix of *double*, there are 7 levels (0-6) in the execution DAG and the overall input data size is $3072 * 2048 * 8 = 48MB$. According to Eq. 4, CAB calculates *BL* as $\max\{\lceil \log_2 4 + 1 \rceil, \lceil \log_2 (48MB/6MB) + 1 \rceil\} = 4$. From Fig. 5, we see that *heat* gains the best performance for data size $3k*2k$ when *BL* is 4. The *BL* values calculated for other data sizes are the ones with the best performance as well according to Fig. 5. This proves the effectiveness of Eq. 4 and our automatic DAG partitioning method.

Note that, for larger data sizes, when *BL* is smaller than 3, the performance of CAB is worse than Cilk. This is because, when *BL* is small, there are only a small number of leaf inter-socket tasks. In this situation, workload is not balanced well in CAB, because CAB may not utilize all the sockets due to the lack of inter-socket tasks. One such extreme case is when $BL = 1$, there is only one leaf inter-socket task, and thus only one squad can get the task.

On the other hand, if *BL* is too large (e.g., >4), each leaf inter-socket task only contains a small number of intra-socket tasks. In this situation, the workload within a squad cannot be balanced well. For example, for $BL = 6$ in the case of $3k * 2k$, leaf inter-socket tasks are in level 6 and do not generate any intra-socket tasks. In this case, there is only one worker contributing to the performance of every squad.

C. Scalability of CAB

Input data sizes can affect the performance of CAB. If input data is very large, the performance gain of CAB tends to be smaller. This experiment uses *heat* and *SOR* to illustrate the scalability of CAB and other benchmarks show similar results.

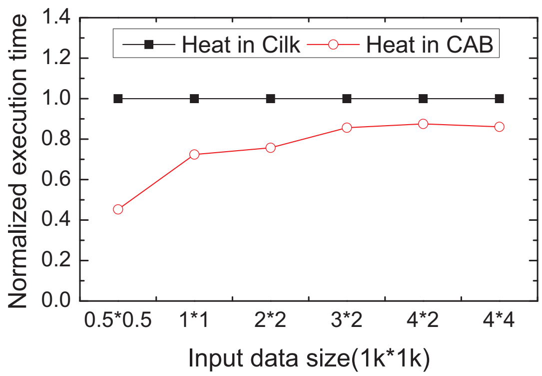
Fig. 6 shows the performance of *heat* and *SOR* with different input data sizes. We can observe that the performance gain of *heat* in CAB is 54.6% when the input data is small (512*512), but drops to 14% when the input data is large (4k*4k), the performance gain of SOR in CAB is 68.7% when the input data is small (512*512) but drops to 13.6% when the input data is large (4k*4k).

One reason for the diminishing gain is that, with the increasing input data sizes, the shared data set between intra-socket tasks becomes relatively smaller, which increases the proportion of non-shared data and the cache misses in the leaf inter-socket tasks. Fig. 7 shows the L2 and L3 cache misses of *heat* and *SOR* with different input data sizes. When the size of input data is small, CAB can reduce nearly 68.2% L3 cache misses and 43.1% L2 misses compared to Cilk. When the input data size is large, however, CAB can only reduce about 4% L3 cache misses and 2.1% L2 misses.

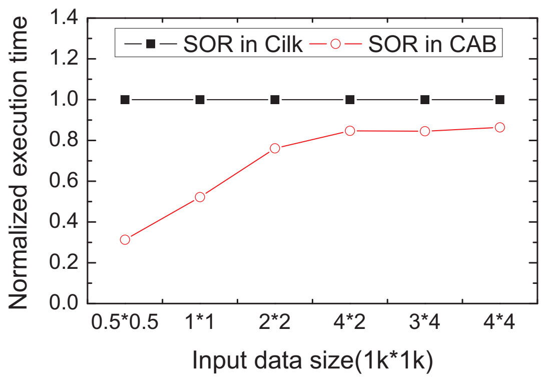
Another reason for the diminishing gain is that, when the input data is large, the granularity of the leaf tasks becomes large, which is not good for load balance within a squad.

D. Performance of CPU-bound applications

Since CAB is proposed to relieve the TRICI syndrome of memory-bound applications, CPU-bound applications cannot achieve better performance in CAB compared to the

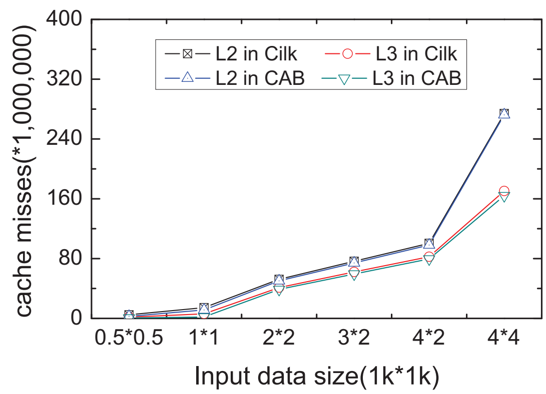


(a) Performance of Heat in CAB and Cilk

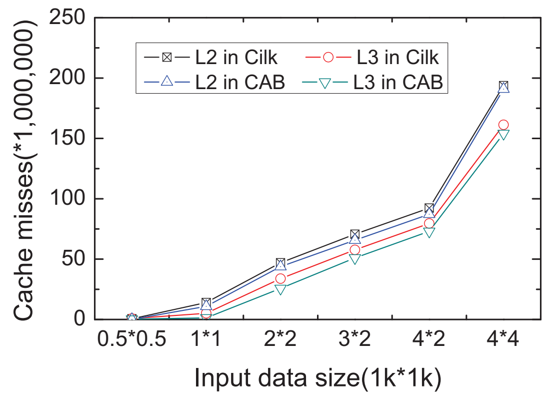


(b) Performance of SOR in CAB and Cilk

Figure 6. Performance result of Heat and SOR with different input data sizes.



(a) L2 and L3 cache misses of Heat in CAB and Cilk



(b) L2 and L3 cache misses of SOR in CAB and Cilk

Figure 7. L2 and L3 cache misses of Heat and SOR in Cilk and CAB.

traditional task-stealing. Therefore, CAB schedules the tasks of CPU-bound applications as the traditional task-stealing by setting *BL* to be 0.

Fig. 8 shows the performance of CPU-bound benchmarks listed in Table III. For most cases, the extra overhead added into the applications by CAB is around 1-2%. For *fft*, the extra overhead caused by manipulating the variable *level* in the task frames is less than 5%, though optimizations are possible to further reduce this overhead.

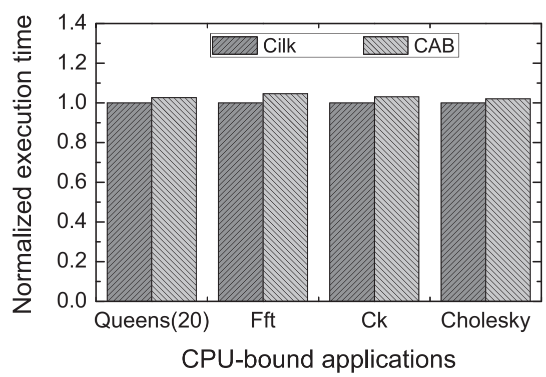


Figure 8. Normalized execution times of CPU-bound applications in CAB. By setting *BL* to be 0, CAB schedules tasks as the traditional task-stealing.

VI. Related Work

Task-stealing is increasingly popular for automatic task scheduling. There has been a lot of research work on its adaptation and improvement [14], [15], [16], [17], [18], [19].

There are generally two policies for task-stealing: child-first and parent-first. In [13], the performance of the two policies was compared. Both child-first and parent-first policies have their strengths and are used pervasively in task-stealing schedulers. For example, MIT Cilk [4], Cilk++ [5], and Intel TBB [6] use the child-first policy, while Java's fork-join framework [8], Wool [20] and Task Parallel Library (TPL) [21] use the parent-first policy. Also there are some task-stealing schedulers that adopt both policies, e.g., SLAW [22]. In SLAW, tasks are generated following either the child-first policy or the parent-first policy according to the stack pressure and task-stealing conditions. Although SLAW uses both policies as in our CAB scheduler, it does not associate the policies to the DAG level of tasks as we do in CAB. We adopt the parent-first policy to quickly generate the tasks in the inter-socket tier, but use the child-first policy to prevent the excessive task proliferation in the intra-socket tier.

Reducing the overhead of task-stealing has been a popular research issue. The overhead of task-stealing mainly includes task generating overhead and large numbers of unnecessary steals. AdaptiveTC [16] proposes an adaptive task generation strategy to keep all workers busy most of the time while reducing the number of tasks generated. XWS [10] proposes an adaptive batching schemes to batch several small tasks together into a larger task. In [17], a non-blocking steal-half algorithm was introduced for a worker to steal half of the tasks from the victim worker, which can reduce the number of steals. Wool [20] proposes a low overhead task-stealing algorithm to cope with the high overhead of task creating in applications with fine grained tasks. In [18], an idempotent task-stealing was introduced and several algorithms were proposed to exploit the relaxed semantics of task execution in order to achieve a better performance. The relaxed semantics guarantee that each task is eventually executed at least one time, instead of exactly one time. The techniques of these studies are orthogonal to our approach and could be integrated into our CAB scheduler to further reduce task-stealing overhead.

Other studies have extended task-stealing to asymmetric multi-processors and distributed memory systems. In [23], a work-stealing model in which each processor maintains an estimation of its speed was presented. The model allows a fast processor to grab tasks from a slow processor when all the task pools are empty. In [24], a scalable task-stealing scheduler that works on both shared memory and distributed memory architectures was proposed to balance workload among cores and processors dynamically. In [25], a runtime system was proposed for supporting task-stealing on 8,192 processing cores on a cluster computer with distributed memory. In [26], Guoping et al. designed a manycore architecture to support task-stealing at hardware level. In contrast to these special architectures, CAB is designed for the popular MSMC architecture.

Cache awareness is another interesting issue in task-stealing. In [27], a theoretical bound on the number of cache misses for random task-stealing was presented and a locality-guided task-stealing algorithm was implemented on a single-socket SMP. In [28], cache behaviors of task-stealing and a parallel depth-first scheduler were compared and analyzed on a multi-core simulator that has shared L2 caches among cores. It proposed to promote constructive cache sharing through controlling task granularity. However, the above studies did not take the MSMC architecture into consideration, and thus did not address the TRICI syndrome described in this paper.

There are also some researches aiming to gain good cache performance based on other techniques. In [29], a hybrid parallel depth first scheduling algorithm was proposed to avoid scheduling tasks that have large data sets (greater than the L2 cache) onto the same core simultaneously to reduce capacity and conflict cache misses. In [30], a bank-aware cache partitioning scheme was proposed. In the scheme, the last level shared cache was partitioned optimally for multiple running applications based on its cache miss prediction model. In [31], a profile-based cache-aware task dividing scheme was proposed to minimize cache misses for nested parallel loops in multi-core architecture. However, these techniques did not target problems in task-stealing.

VII. Conclusions and Future Work

Traditional random task-stealing suffers from the TRICI syndrome in the MSMC architecture. To address this problem, we have designed and implemented CAB scheduler, which automatically partitions the execution DAG into the inter-socket and the intra-socket tiers. Through careful calculations, tasks that have data sharing are placed on the same CPU socket, thus greatly reducing the number of cache misses. Experimental results demonstrate that CAB can achieve up to 68.7% performance gain for memory-bound applications and the extra overhead for CPU-bound applications is only 1-2%.

One of our future work is to design a more flexible DAG partitioning method that can decide inter-socket and intra-socket tasks with heuristic information and compiler support instead of a single boundary level. Pre-fetching with helper thread is another technique for improving performance [3]. An interesting future direction is to integrate this technique into CAB for more performance gains.

Acknowledgement

This work was partially supported by NFSC (Grant No. 60811130528, 61003012). Quan Chen would like to thank the Department of Computer Science, University of Otago for funding and hosting his internship.

References

[1] D. Butenhof, *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[2] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message passing interface*. MIT Press, 1999.

[3] J. Zhang, Z. Huang, W. Chen, Q. Huang, and W. Zheng, "Maotai: View-Oriented Parallel Programming on CMT processors," in *37th International Conference on Parallel Processing*, pp. 636–643, IEEE, 2008.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, (Montreal, Quebec, Canada), pp. 212–223, ACM, June 1998.

[5] C. Leiserson, "The Cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 522–527, ACM, 2009.

[6] J. Reinders, *Intel threading building blocks*. O'Reilly, 2007.

[7] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[8] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, pp. 36–43, ACM, 2000.

[9] J. Lee and J. Palsberg, "Featherweight X10: a core calculus for async-finish parallelism," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pp. 25–36, ACM, 2010.

[10] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *37th International Conference on Parallel Processing*, pp. 536–545, IEEE, 2008.

[11] A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276–291, 1992.

[12] R. D. Blumofe, *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 1995. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

[13] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, IEEE Computer Society, 2009.

[14] D. Hendler, Y. Lev, M. Moir, and N. Shavit, "A dynamic-sized nonblocking work stealing deque," *Sun Microsystems, Inc. Technical Reports; Vol. SERIES13103*, p. 69, 2005.

[15] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, p. 28, ACM, 2005.

[16] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P. Yew, "An adaptive task creation strategy for work-stealing scheduling," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 266–277, ACM, 2010.

[17] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, p. 289, ACM, 2002.

[18] M. M. Michael, M. T. Vechev, and V. A. Saraswat, "Idempotent work stealing," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 45–54, ACM, 2009.

[19] K. Agrawal, C. Leiserson, and J. Sukha, "Helper locks for fork-join parallel programming," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, pp. 245–256, ACM, 2010.

[20] K. Faxén, "Efficient Work Stealing for Fine Grained Parallelism," in *2010 39th International Conference on Parallel Processing*, pp. 313–322, IEEE, 2010.

[21] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 227–242, 2009.

[22] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: a scalable locality-aware adaptive work–stealing scheduler," in *the 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2010.

[23] M. Bender and M. Rabin, "Scheduling Cilk multithreaded parallel programs on processors of different speeds," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pp. 13–21, ACM, 2000.

[24] S. Olivier and J. Prins, "Scalable dynamic load balancing using UPC," in *37th International Conference on Parallel Processing*, pp. 123–131, IEEE, 2008.

[25] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11, ACM, 2009.

[26] G. Long, D. Fan, and J. Zhang, "Architectural support for cilk computations on many-core architectures," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pp. 285–286, ACM, 2009.

[27] U. Acar, G. Blelloch, and R. Blumofe, "The data locality of work stealing," *Theory of Computing Systems*, vol. 35, no. 3, pp. 321–347, 2002.

[28] S. Chen, P. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. Mowry, *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, p. 115, ACM, 2007.

[29] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 501–510, Society for Industrial and Applied Mathematics, 2008.

[30] D. Kaseridis, J. Stuecheli, and L. John, "Bank-aware Dynamic Cache Partitioning for Multicore Architectures," in *2009 International Conference on Parallel Processing*, pp. 18–25, IEEE, 2009.

[31] A. Kejariwal, A. Nicolau, A. Veidenbaum, U. Banerjee, and C. Polychronopoulos, "Efficient Scheduling of Nested Parallel Loops on Multi-Core Systems," in *2009 International Conference on Parallel Processing*, pp. 74–83, IEEE, 2009.