Adaptive Cache Aware Bitier Work-Stealing in Multisocket Multicore Architectures

Quan Chen, Minyi Guo, *Senior Member*, *IEEE*, and Zhiyi Huang

Abstract—Modern multicore computers often adopt a multisocket multicore architecture with shared caches in each socket. However, traditional work-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 an Adaptive Cache-Aware Bi-tier work-stealing (A-CAB) scheduler. A-CAB improves the performance of memory-bound applications by reducing memory footprint and cache misses of tasks running inside the same CPU socket. A-CAB adaptively uses a DAG partitioner to divide an execution Directed Acyclic Graph (DAG) into the intersocket tier and the intrasocket tier. Tasks in the intersocket tier are scheduled across sockets while tasks in the intrasocket tier are scheduled within the same socket. Experimental results tell us that A-CAB can improve the performance of memory-bound applications up to 74.4 percent compared with the traditional work-stealing.

Index Terms—Cache aware, work-stealing, multisocket multicore architectures, divide-and-conquer programs

1 Introduction

Multicore processors have become mainstream because they have better performance per watt and larger computational capacity than complex single-core processors. However, a single CPU die can hardly contain too many cores (e.g., more than 128 cores) due to the physical limitations in industrial manufacture. To fulfill the urgent demand on powerful computers, many multicore CPUs are integrated together into a Multisocket multicore (MSMC) architecture. In an MSMC architecture, each CPU die is plugged into a socket and the cores in the same socket have a shared cache; however, the cores from different sockets can only share the main memory.

To fully utilize the MSMC architectures, many parallel programming environments have been proposed. In some of them, such as Pthread [10], MPI [18], and Maotai [31], parallelism is expressed through multithreading. Programmers need to launch threads and assign tasks to these threads manually in multithreading. However, the manual assignment of tasks is often burdensome for developing applications. To relieve the burden of parallelization and task assignment, parallel programming environments, such as MIT Cilk [9], Cilk++ [26], TBB [29], X10 [24], and OpenMP [2], assign and schedule tasks automatically. *Work-sharing* [2] and *work-stealing* [8] are the two most famous task scheduling strategies.

In work-sharing, workers (i.e., threads) 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.

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

However, both work-sharing and work-stealing strategies schedule tasks without considering the data locality issue. The strategies can cause shared cache misses and degrade the performance of memory-bound applications on MSMC architectures (to be discussed in Section 2). A memory-bound application is an application whose performance is decided by the data access time. For example, two tasks with shared data may be allocated to different sockets due to the randomness in these strategies. In this case, both tasks cannot share the data loaded to the shared cache but have to read the shared data from the main memory, which could be hundreds times slower than the shared cache. If the two tasks are scheduled to the same socket, only one of them needs to read the shared data from the main memory while the other task can access the data from the shared cache directly.

Based on this observation, this paper proposes an Adaptive Cache-Aware Bi-tier (A-CAB) work-stealing scheduler that automatically schedules tasks with shared data into the same socket. It is targeting *divide-and-conquer* (D&C) memory-bound applications, which covers a wide range of scientific applications in fluid dynamics, quantum dynamics, binary alloys, electromagnetism, superconductivity, thermodynamics, environmental systems, and so on. The simulation of these systems has data parallelism that is often exploited with stencil-based approaches [4]. These

- *Q. Chen and M. Guo are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Room 3-415 SEIEE building, No. 800 Dongchuan Road, Shanghai 200240, China. E-mail: chen-quan@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn.*
- *Z. Huang is with the Department of Computer Science, University of Otago, PO Box 56, Dunedin 9054, New Zealand. E-mail: hzy@cs.otago.ac.nz.*

Manuscript received 22 Aug. 2012; revised 31 Oct. 2012; accepted 1 Nov. 2012; published online 28 Nov. 2012.
Recommended for acceptance by S.-Q. Zheng.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-08-0756.
Digital Object Identifier no. 10.1109/TPDS.2012.322.

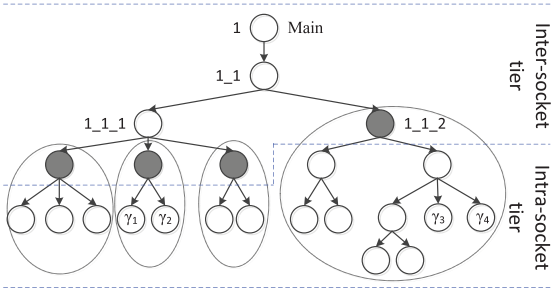
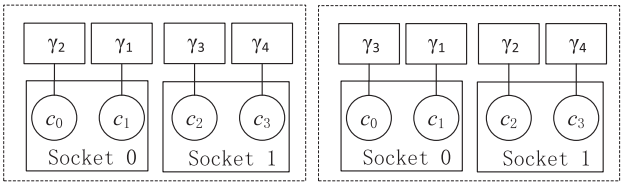


Fig. 1. A general execution DAG for D&C programs.



(a) Optimal scheduling (b) Another possible scheduling

Fig. 2. Two possible scheduling of γ_1, γ_2, γ_3, and γ_4 on a dual-socket dual-core architecture. The first scheduling can gain performance improvement due to cache sharing and reduction of memory footprint.

applications are often iterative since the computation is repeated in steps of time. Their execution Directed Acyclic Graph (DAGs) in systems like MIT Cilk are tree-shaped, which is ideally suitable for A-CAB.

A-CAB consists of a *DAG partitioner* and a *bitier work-stealing scheduler*. The DAG partitioner divides the execution DAG of a program into the intersocket tier and the intrasocket tier. The bitier work-stealing scheduler allows tasks in the intersocket tier to be stolen across sockets, while tasks in the intrasocket tier are scheduled within the same socket.

The contributions of this paper are as follows:

- We propose a *profiling-based method* and a *compiling-based method* that collect the data access feature of tasks for iterative programs and noniterative programs. Based on the collected information, a DAG partitioner divides tasks into the intersocket tier and the intrasocket tier.
- We propose a bitier work-stealing scheme that schedules tasks with shared data to the same socket.
- We demonstrate that A-CAB significantly reduces the shared cache misses and, thus, improves the performance of memory-bound applications. The experiment shows that A-CAB can achieve a performance gain of up to 74.4 percent compared with traditional work-stealing.

This paper is extended from our previous work [12]. The rest of this paper is organized as follows: Section 2 describes the problem and explains the motivation of A-CAB. Section 3 presents A-CAB, including the DAG partitioner and the bi-tier work-stealing scheduler. Section 4 shows the experimental results and the limitations of A-CAB. Section 5 discusses the related work. Section 6 draws conclusions and sheds light on future work.

2 Problem and Motivation

For many parallel programming environments such as MIT Cilk, the execution of a parallel program can often be expressed by a Directed Acyclic Graph $G = (V, E)$, where V is a set of nodes, and E is a set of directed edges [17]. Each node in a DAG represents a task (i.e., a set of instructions) that must be executed sequentially without preemption, and the edges in a DAG correspond to the dependence relationship among the nodes. Fig. 1 shows execution DAG of a general parallel program. In the figure, the solid lines represent the task generating relationship and the strings by the side of nodes are the identifiers of the corresponding tasks.

2.1 The Problem

We use Fig. 1 as an example to explain the problem of shared cache pollution in an MSMC architecture. In many parallel D&C programs, neighbor tasks need to access some shared data. For example, *Five-point heat distribution* is an example of such parallel programs. Therefore, γ_1 and γ_2, γ_3 and γ_4 in Fig. 1 have shared data, respectively.

We assume the program in Fig. 1 runs on a dual-socket dual-core computer. If γ_1, γ_2, γ_3, and γ_4 are scheduled as shown in Fig. 2a, the shared data between γ_1 and γ_2 and the shared data between γ_3 and γ_4 is only read into the shared cache once from the main memory. Since most tasks can access the shared data in the shared cache, cache misses are reduced.

However, for traditional work-stealing, because it randomly chooses a victim to steal tasks, γ_1, γ_2, γ_3, and γ_4 are likely to be scheduled to the cores as shown in Fig. 2b. In this case, each task needs to read all its data from the main memory. This larger memory footprint leads to more compulsory cache misses. Even worse, if the memory footprint exceeds the capacity of the shared cache, the situation leads to more capacity cache misses and increases the chances of conflict cache misses. The resulted larger number of cache misses will lead to the worse performance of memory-bound applications.

Though there were some task schedulers proposed to reduce cache misses, they are not general enough for MSMC architectures [5], [6].

2.2 Proposed Solution

If a work-stealing scheduler can ensure tasks with shared data are scheduled to the same socket as shown in Fig. 2a, the shared cache misses will be minimized and the performance of memory-bound applications can be improved. To achieve the purpose, we propose the *Adaptive Cache Aware Bi-tier work-stealing* scheduler for memory-bound D&C parallel programs.[1]

A-CAB divides an execution DAG into the intersocket tier and the intrasocket tier. For example in Fig. 1, A-CAB may divide the execution DAG into two tiers separated by the shaded tasks. The shaded tasks are called *leaf intersocket tasks*. Tasks above the leaf intersocket tasks, including the leaf intersocket tasks, are called *intersocket tasks*, which belong to the intersocket tier. Tasks in a subtree rooted with

1. All the programs mentioned below are memory-bound D&C parallel programs.

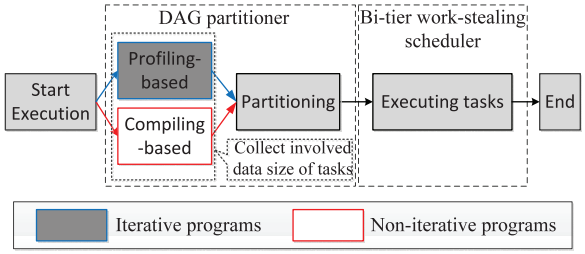


Fig. 3. The processing flow of a parallel program in A-CAB.

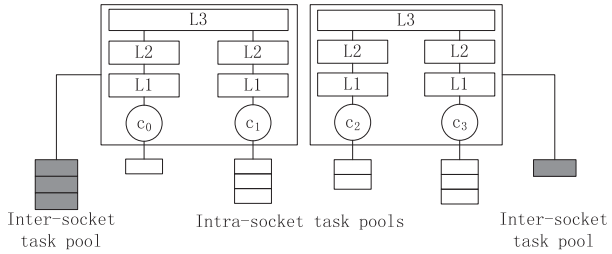


Fig. 4. A-CAB runtime environment in a dual-socket dual-core architecture.

a leaf intersocket task are called *intrasocket tasks*, which belong to the intrasocket tier. A subtree rooted with a leaf intersocket task is called an *intrasocket subtree*. For example, in Fig. 1, tasks in an ellipse consist in an intrasocket subtree. The goal of A-CAB is to schedule tasks in the same intrasocket subtree within the same socket. In this way, A-CAB can ensure γ_1 and γ_2 (or γ_3 and γ_4) to be executed in the same socket.

However, if an intrasocket subtree is too large, the involved data can be too large to fit into the shared cache of the socket. On the other hand, if an intrasocket subtree is too small, the workload of the subtree can be too small to get better balanced among the cores of the same socket.

To find the proper leaf intersocket tasks, A-CAB should find out the involved data size of all the tasks first. The DAG partitioner of A-CAB finds out the involved data size of tasks using a *profiling-based method* for iterative programs and a *compiling-based method* for noniterative programs.

Based on the involved data sizes of tasks that are either collected in profiling-based method or compiling-based method, the DAG partitioner in A-CAB adaptively divide the execution DAG into two tiers (to be discussed in Section 3.2).

After the partitioning of the DAG, a bitier work-stealing algorithm is adopted in A-CAB to schedule tasks in the two tiers differently. The intersocket tasks are scheduled across sockets, while the tasks in the same intrasocket subtree are scheduled within the same socket. A-CAB ensures that each socket can only execute one intrasocket subtree at the same time to avoid cache pollution. In this way, the shared data can be reused without reloading among tasks within an intrasocket subtree. Fig. 3 illustrates the detailed processing flow of a parallel program in A-CAB.

3 Adaptive Cache Aware Bitier Work-Stealing

This section presents A-CAB, an Adaptive Cache Aware Bi-tier work-stealing scheduler. First, we give the A-CAB runtime environment. Then, we describe the DAG partitioner, including the profiling-based method and the compiling-based method for calculating the involved data size of tasks and the way of dividing the execution DAG into two tiers. Lastly, we present the bitier work-stealing algorithm.

In addition, in Section 2 of the supplemental document, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.322, we present the task-generating policy in A-CAB. In Section 3 of the supplemental document, available online, we present the implementation of A-CAB. In addition, we discuss the theoretical time and space bounds of A-CAB in Section 4 of the supplemental document, available online.

A-CAB is proposed based on the following three observations on the execution of D&C parallel programs as shown in Fig. 1. First, parallel tasks create child tasks recursively until the data set for each leaf task is small enough. During the procedure, only the leaf tasks physically touch the data. Second, neighbor tasks usually share some data. Lastly, if the parallel program is iterative, it often works on the same data set for a large number of iterations.

3.1 A-CAB Runtime Environment

To support the processing flow in Fig. 3, we have built a runtime environment for A-CAB as follows: For an M-socket N-core architecture, A-CAB launches $M \times N$ workers (i.e., threads) at runtime and affiliates each worker with one individual hardware core as shown in Fig. 4. For convenience of presentation, we use the term *core* to mean a worker in the rest of the paper.

In each socket, only one core is selected as the head core of the socket to look after the intersocket task scheduling. In A-CAB, we choose "core 0" in each socket as the socket's header core.

To schedule intersocket tasks and intrasocket tasks in different ways in bitier work-stealing, A-CAB creates an *intersocket task pool* for each socket to store intersocket tasks, and an *intrasocket task pool* for each core to store intrasocket tasks, as shown in Fig. 4. Note a task pool is a double-ended queue here.

For an iterative program, during its first iteration, all the tasks are generated and pushed into intrasocket task pools when they are generated. In this case, tasks are scheduled adopting traditional work-stealing policy. That is, in the first iteration, tasks in intrasocket task pools can be scheduled across sockets because the profiling information has not been collected and, thus, the execution DAG has not been partitioned. In the following iterations, tasks are generated and pushed into different pools accordingly.

For a noniterative program, the DAG partitioner divides the execution DAG directly on the basis of user-provided information and information provided by compiler. Therefore, for the program, all the tasks are generated and pushed into different pools directly.

In bitier work-stealing, if core c in socket ρ generates a task γ that is an intersocket task, γ is pushed into ρ's intertask pool. Otherwise, if γ is an intrasocket task, it is pushed into c's intrasocket task pool.

3.2 DAG Partitioner

To partition an execution DAG into two tiers appropriately, the most challenging problem is to find the proper leaf intersocket tasks. Once the proper leaf intersocket tasks are identified, the DAG can be easily divided into two tiers: all the tasks above the leaf intersocket tasks (including the leaf intersocket tasks) belong to the intersocket tier, and those tasks in the subtrees rooted with leaf intersocket tasks belong to the intrasocket tier.

An appropriate partitioning of an execution DAG should satisfy two constraints. The first constraint is that, for any intra-socket subtree *ST*, the involved data of all the tasks in *ST* is small enough to fit into the shared cache of a socket. The second constraint is that an intrasocket subtree *ST* should be large enough to allow a socket to have sufficient intrasocket tasks.

To fulfill the two constraints, for any task γ in the execution DAG, A-CAB should collects its involved data size. For convenience of description, we use *Size Of Involved Data* (SOID) to represent the involved data size of a task γ. That is, SOID includes the data accessed by all tasks in the subtree rooted with γ. Once the SOIDs for all tasks in the execution DAG are known, the DAG partitioner can divide the execution DAG into two tiers appropriately.

As stated before, A-CAB uses the profiling-based method to collect SOIDs of tasks for iterative programs while using the compiling-based method to collect SOIDs of tasks for noniterative programs.

3.2.1 Profiling-Based Method

According to our observation of iterative D&C parallel programs, the collected profiling information in the first iteration can be used to predict the execution behavior of the following iterations. Therefore, an optimal partitioning of DAG based on the profiling information of the first iteration will also be optimal for the following iterations.

Therefore, for an iterative program, A-CAB profiles the program during the first iteration of the execution. During the online profiling, we use the hardware Performance Monitoring Counters (PMC) [3] to collect cache misses, based on which the SOIDs for all tasks are calculated. The performance counter event we have used is the last level private data cache (e.g., L2 in AMD Quad-core Opteron 8380) misses. That is, we have used the performance counter event "07Eh" with mask of "02h" to collect the last level private data cache misses in AMD Quad-core Opteron 8380. For detailed information of the performance counter events, refer to *BIOS and Kernel Developer's Guide* of the corresponding processor. Though it is straightforward to collect the event statistics of the last level private data cache misses in modern multicore machines like X86_64, it is very tricky to calculate the SOIDs of the tasks based on the last level private data cache misses.

First, limited by the hardware PMCs, a core can only collect the cache misses of its own, but a task may have multiple child tasks executing on different cores. Therefore, it is impossible to collect the overall cache misses for a task directly.

Second, it is nontrivial to relate the private cache misses to the SOID of a task. For a task γ that runs on a core c in socket ρ, if γ fails to get its data from the last level private cache of c, it requests the data from the shared cache of ρ. Since c does not execute other tasks when it is executing γ, the last level private cache misses of c are totally caused by γ. The last level private cache misses of c can be used to approximate to the size of data accessed by γ for the following reasons. Many memory-bound applications adopt data parallelism. As mentioned in our second observation in Section 2.2, only the leaf tasks physically access data. The data of leaf tasks do not have much overlapping with each other. Even when two neighbor leaf tasks have a small portion of shared data, the chances for them to be executed in the same core are small in a traditional work-stealing scheduler, which is adopted during the profiling stage. Therefore, the above approximation is accurate enough for us to calculate the SOIDs of all tasks.

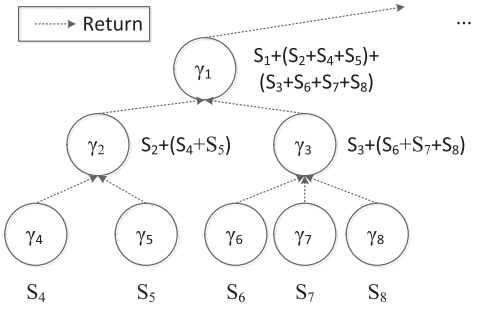


Fig. 5. Collect size of involved data for tasks in the profiling-based method.

Based on the collected last level private cache misses of γ, its SOID is calculated as follows: If γ is a leaf task, the number of cache misses of γ times the cache line size (e.g., 64 bytes in AMD Quad-core Opteron 8380) is γ's SOID. Otherwise, if γ is not a leaf task, its SOID is the sum of its cache misses times the cache line size plus the SOIDs of all its child tasks. Given a task β with n subtasks $\beta_1, \beta_2, \ldots, \beta_n$. Suppose M is β's number of cache misses times the size of cache line, and the SOIDs of its child tasks are $S_1, S_2, \ldots, S_n$, respectively, then β's SOID, denoted by S_β, is calculated as in

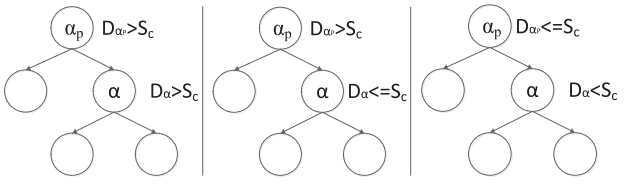
$$S_\beta = M + \sum_{i=1}^{n} S_i. \tag{1}$$

Based on (1), Fig. 5 presents an example of calculating SOIDs for all the tasks. In the figure, S_i is the SOID for leaf task γ_i, but represents the size of data physically accessed by the task itself for nonleaf tasks. In fact, for many memory-bound applications, S_i for nonleaf tasks is very small, if it is not zero, since nonleaf tasks do not physically access data.

As shown in Fig. 5, the SOID of a task is returned to its parent task when it is completed. For example, in Fig. 5, γ_2's SOID is added to γ_1's SOID when γ_1 is completed. Therefore, when all the tasks in the first iteration are completed, the SOIDs of all the tasks can be calculated.

3.2.2 Compiling-Based Method

The profiling-based method is not applicable for noniterative programs because the programs only access the target



(a) α is an inter-socket task (b) α is a leaf inter-socket task (c) α is an intra-socket task

Fig. 6. Conditions that α is an intersocket task, a leaf intersocket task or an intrasocket task.

data set once. For noniterative programs, the DAG partitioner calculates the SOIDs of tasks in the semiautomatic compiling-based method.

In the compiling-based method, for any task γ, we calculate its SOID S_γ using the effective input data size of the program and the branching degree of all its ancestors in the DAG. Note that, in the following calculation, we assume a task divides its data set into several parts evenly according to its branching degree. This assumption is true in most of current D&C programs.

Suppose the effective input data size of the application is S_{input}, the ancestors of task γ in the DAG are $\gamma_0, \gamma_1, \ldots, \gamma_i$ whose branching degrees are $B_0, B_1, \ldots, B_i$ accordingly. Then, the SOID of γ can be calculated with (2). Most D&C programs satisfy this equation:

$$S_\gamma = \frac{S_{input}}{\prod_{j=0}^{i} B_j}. \quad (2)$$

We automatically acquires the branching degree of each task by analyzing the task generating pattern in the source code through the compiler. The shared cache size can be obtained from *proc/cpuinfo* by our runtime system. However, the effective input data size of the application S_{input} has to be provided through a command line argument.

3.2.3 DAG Partitioning

Once the SOIDs of all the tasks are calculated, the DAG partitioner divides the execution DAG into intersocket tier and intrasocket tier automatically.

To satisfy the aforementioned constraints, the DAG partitioner identifies leaf intersocket tasks as follows: For a task α and its parent task α_p, let D_α and D_{α_p} represent SOIDs of α and α_p, respectively. α is a leaf intersocket task if and only if D_α is smaller than the size of the shared cache and D_{α_p} is larger than the size of the shared cache. More precisely, given a task α and its parent task α_p, our DAG partitioning method determines α's tier as follows:

- If both D_{α_p} and D_α are larger than the shared cache of a socket, α is an intersocket task, as shown in Fig. 6a.
- If D_{α_p} is larger than the shared cache and D_α equals to or is smaller than the shared cache, α is a leaf inter-socket task, as shown in Fig. 6b.
- If D_{α_p} equals to or is smaller than the shared cache and D_α is smaller than the shared cache, α is an intrasocket task, as shown in Fig. 6c.

After the partitioning, the DAG partitioner has already divided the execution DAG into two tiers appropriately. Then, based on the partitioning, bitier work-stealing can be adopted to schedule tasks for optimizing shared cache in the following iterations.

For iterative programs, to identify the same task in the following iterations, during the execution of a parallel program, each task is given an identifier (a string) according to the spawning relationship between tasks. If a task γ's identifier is S, then its ith subtask's identifier is S_i. For example, Fig. 1 shows the way of constructing identifiers for tasks. The strings beside the tasks are the identifiers in Fig. 1. The identifiers of all the completed tasks are saved in a hash table with their SOIDs. When a new task is spawned, A-CAB tries to find its identifier in the hash table. If the identifier is found, it means the first iteration has completed because a new task in the same location of the execution DAG has been spawned. In this case, A-CAB uses the bitier work-stealing scheduler to schedule tasks based on their tiers, which are decided according to their SOIDs as shown above.

It is worth noting that, in our implementation, for iterative programs, all the needed information for optimal bitier work-stealing is obtained automatically by the runtime system of A-CAB. In this way, A-CAB automatically improves the performance of iterative programs without any human intervention. For noniterative programs, A-CAB only needs users to provide the input data size of the programs while all the other information is obtained by the compiler automatically. Compared with our previous work CAB [13] that also needs human intervention to divide execution DAG, the profiling-based method and the compiling-based method are adaptive to irregular and unbalanced DAGs where the leaf intersocket tasks do not have the same depth, because it determines the leaf intersocket tasks according to the SOIDs of tasks instead of the task's depth in the execution DAG.

3.3 Bitier Work-Stealing Scheduler

When A-CAB starts to execute a parallel program, if the program is an iterative program, A-CAB has not partitioned its execution DAG into two tiers during the first iteration. Therefore, the cores adopt the traditional work-stealing algorithm to obtain or steal a new task in the first iteration. In the following iterations, A-CAB adopts a bitier work-stealing algorithm to schedule tasks so that tasks in a subtree rooted with a leaf intersocket task are scheduled to the same socket. If the program is a noniterative program, A-CAB directly adopts the bitier work-stealing algorithm to schedule tasks. A command line argument is used to indicate whether a program is iterative or not.

Since traditional work-stealing has been discussed in detail in [8], we only present the bitier work-stealing in A-CAB here.

When a core c in socket ρ is free, it first tries to obtain a task from its own intrasocket task pool. If its task pool is empty, c tries to steal a task from the intrasocket task pools of other cores in ρ. If the task pools of all the cores in ρ are empty, the head core of ρ tries to obtain a task from its intersocket task pool. If its intersocket task pool is empty, the head core tries to steal an intersocket task from other sockets.

In A-CAB, only the head core of each socket can steal intersocket tasks so that the lock contention of the intersocket task pools is reduced. In addition, cores in the

TABLE 1
Benchmarks Used in the Experiments

Name	Bound	Description
Mandelbrot	CPU	Calculate Mandelbrot Set
Queens(15)	CPU	N-queens problem
FFT	CPU	Fast Fourier Transform
GA	CPU	Island Model of Genetic Algorithm
Knapsack	CPU	0-1 knapsack problem
Heat	Memory	Five-point heat
Heat-ub	Memory	Five-point heat (unbalance)
SOR	Memory	Successive Over-Relaxation
SOR-ub	Memory	Successive Over-Relaxation (unbalance)
GE	Memory	Gaussian elimination
GE-ub	Memory	Gaussian elimination (unbalance)

same socket are not allowed to execute tasks in different intrasocket subtrees at the same time. This policy can avoid the situation where different intrasocket subtrees pollute the shared caches with different data sets. The downside of the policy is that some cores in a socket may be idle waiting for other cores to finish their tasks. An alternative policy is to allow a socket to execute tasks from more than one intrasocket subtrees at the same time. This alternative policy can ensure most cores are busy, but different intrasocket subtrees may pollute the shared caches, which leads to more cache misses. For the memory-bound applications that A-CAB is targeting, the cache misses are more critical to the performance according to our experimental results. Therefore, we have adopted the first policy in A-CAB.

To fulfill the first policy, we use a Boolean variable *busy_state* in each socket to indicate whether there is an intersocket task running in the socket. If a socket obtains or steals an intersocket task successfully, its *busy_state* is set true. Once the socket finishes its intersocket task, its *busy_state* is set false. Only if *busy_state* is false, should the socket obtain or steal another intersocket task. Suppose w is a core in a socket ρ, and w is free and trying to get a new task.

4 Evaluation

We use a Dell 16-core computer that has four AMD Quad-core Opteron 8380 processors (codenamed "Shanghai") running at 2.5 GHz to evaluate the performance of A-CAB. Each Quad-core socket has a 512K private L2 cache for each core and a 6M L3 cache shared by all four cores. The computer has 16-GB RAM and runs Linux 2.6.29.

Since A-CAB is proposed to reduce cache misses, we use memory-bound benchmarks to evaluate the performance of A-CAB. However, CPU-bound benchmarks are also used to measure the extra overhead of A-CAB compared with traditional work-stealing.

To evaluate the performance of A-CAB in different scenarios, we use only benchmarks that have both balanced and unbalanced execution DAGs in the experiments; however, A-CAB can improve the performance of many stencil-based simulation programs [4], as we mentioned before. Table 1 lists the used benchmarks. Since we can configure the iteration number of the memory-bound benchmarks, we can evaluate the compiling-based method for noniterative programs by adjusting the benchmarks to run only 1 iteration.

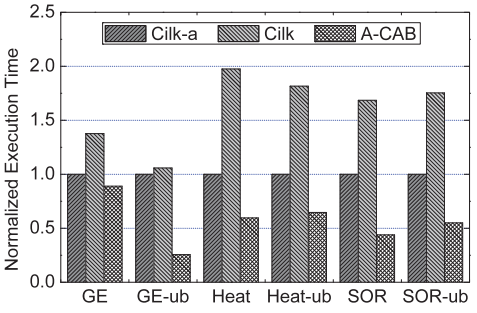


Fig. 7. The performance of iterative memory-bound benchmarks in Cilk-a, Cilk, and A-CAB.

We manually construct benchmarks whose execution DAGs are unbalanced trees (i.e., *Heat-ub*, *GE-ub*, and *SOR-ub*) because it is hard to find benchmarks whose execution DAGs are unbalanced trees natively. For example, we implement *Heat-ub* in Algorithm 1 in the supplemental document, available online.

As mentioned before, A-CAB affiliates each worker with a hardware core. However, MIT Cilk does not affiliate workers with the cores. Therefore, we have modified the MIT Cilk (denoted as *Cilk*) to affiliate each worker with a hardware core (denoted as *Cilk-a*) to ensure fair comparison, because the affiliation of workers with cores can improve the performance of memory-bound applications (to be shown in Fig. 7). We implement Cilk-a and A-CAB based on MIT Cilk. The MIT Cilk programs run with Cilk-a and A-CAB without any modification.

All benchmarks are compiled with "cilk2c -O2", which is based on gcc 4.4.3. Furthermore, for each test, every benchmark runs 10 times. Since the execution time is very stable, the average execution time is used in the final results.

4.1 Performance of Memory-Bound Applications

Fig. 7 shows the performance of memory-bound benchmarks in Cilk, Cilk-a, and A-CAB with a $1{,}024 \times 512$ matrix as the input data. For *GE* and *GE-ub*, the used input data are a $1{,}024 \times 1{,}024$ matrix. All the benchmarks consist of 200 iterations in this experiment. Since the benchmarks are iterative, the DAG partitioner of A-CAB adopts the profiling-based method to calculate the SOIDs of tasks at the first iteration of the benchmarks.

As we can see from Fig. 7, A-CAB with the profiling-based method can significantly improve the performance of iterative memory-bound applications compared to Cilk-a while the performance improvement ranges from 35.3 to 74.4 percent.

To explain why A-CAB can improve the performance of memory-bound applications compared with Cilk-a, we collect the cache misses of all the benchmarks and list them in Table 2. Observed from the table, we can find that the shared cache (L3) misses are prominently reduced while the private cache (L1 and L2) misses are also slightly reduced in A-CAB compared with Cilk-a. Since A-CAB schedules tasks with shared data into the same socket, the shared cache misses have been significantly reduced.

Although scheduling tasks with shared data to the same socket only reduces the shared L3 cache misses, the affiliation of an intrasocket subtree with a socket in A-CAB can help

TABLE 2
Cache Misses in Cilk-a and A-CAB (*1E6)

Application	Scheduler	L1	L2	L3
GE	Cilk-a	60.8	58.8	14.5
	A-CAB	53.9	50.3	2.94
GE-ub	Cilk-a	37.2	37.1	10.7
	A-CAB	23.9	20	2.15
Heat	Cilk-a	82.7	79.6	24.8
	A-CAB	71.1	67.5	5.9
Heat-ub	Cilk-a	82.2	78.7	29.7
	A-CAB	71.3	67.6	3.72
SOR	Cilk-a	88.5	85	29.6
	A-CAB	70.7	66.2	4.75
SOR-ub	Cilk-a	89.8	85.5	30.7
	A-CAB	73.6	67.4	8.27

reduce the L2 cache misses slightly. In A-CAB, for a task γ_i in an intrasocket subtree, if it is executed by core c in socket ρ, its neighbor tasks (i.e., γ_{i-1} and γ_{i+1}) are also executed by c as well unless they are stolen by other cores in ρ. Compared with traditional work-stealing where any free cores can steal γ_i's neighbor tasks, there are fewer cores that can steal γ_i's neighbor tasks in A-CAB. Therefore, the probability that neighbor tasks are executed by the same core is larger in A-CAB. For this reason, the private cache (e.g., L1 and L2) misses have also been slightly reduced in A-CAB.

Fig. 8 shows the SOIDs of tasks in *Heat* with a $2{,}048 \times 512$ matrix as input data that are calculated with (1). The real involved data size of tasks in Fig. 8 are shown in the circles. Since *Heat* uses two matrices of "double" during the execution, the overall input data size is $2{,}048 \times 512 \times 8 \times 2 = 16$ MB. Then, the real data set is evenly divided every time when the tasks are spawned. From the figure, we can find that the calculated SOIDs are close to the real involved data sizes, which shows the profiling-based method is reasonably accurate. In future, to calculate SOIDs more accurately, we will explore more hardware performance counters.

To evaluate the compiling-based method, Fig. 9 shows the performance of noniterative benchmarks in Cilk, Cilk-a, and A-CAB. We create the noniterative benchmarks by setting their iteration number as 1.

With the compiling-based method, A-CAB significantly improves the performance of noniterative memory-bound applications compared to Cilk-a while the performance improvement ranges from 22.3 to 55.1 percent. Same as iterative benchmarks, the good performance of noniterative applications in A-CAB origins from the reduced shared cache misses as well.

From Figs. 7 and 9, we can find that Cilk-a provides better performance compared with Cilk. For memory-bound

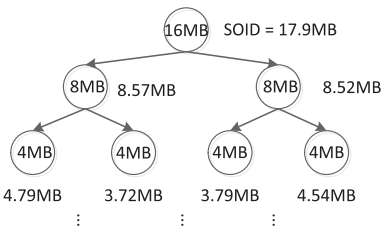


Fig. 8. SOIDs of tasks in *Heat* with a $2{,}048 \times 512$ matrix as input data.

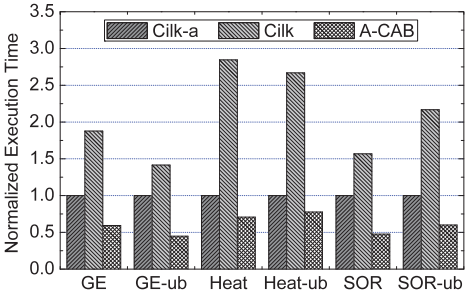


Fig. 9. The performance of noniterative memory-bound benchmarks in Cilk-a, Cilk, and A-CAB.

applications, the better performance in Cilk-a results from the affiliation of the workers with the cores. In the rest of our experiments, we only compare the performance of A-CAB with Cilk-a.

In summary, with the profiling-based method for iterative programs and the compiling-based method for noniterative programs, A-CAB is effective for memory-bound D&C programs.

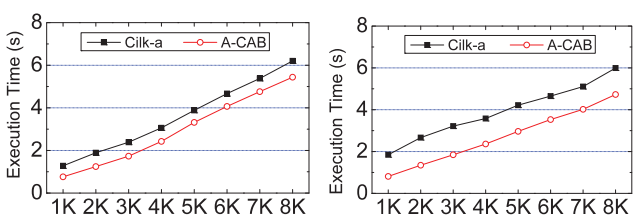
4.2 Scalability of A-CAB

To evaluate scalability of A-CAB in different scenarios, we use benchmarks that have both balanced and unbalanced execution DAGs. We list the experiment results for benchmarks with unbalanced execution DAGs in Section 5.2 of the supplemental document, available online. In this experiment, we execute benchmarks with different input data sizes in A-CAB and Cilk-a to compare their scalability.

During the execution of all the benchmarks, every task divides its data set into several parts by rows to generate child tasks unless the task meets the cutoff point (i.e., the data set size of a leaf task). Since the data set size of the leaf tasks affects the measurement of scalability, we should ensure that the data set size of the leaf tasks is constant in our experiment. To satisfy this requirement, we use a constant cutoff point, eight rows, for the leaf tasks, and a constant number of columns, 512, for the input data. We only adjust the number of rows of the input matrix in the experiment. In this way, we can measure the scalability of A-CAB without the impact of the granularity of the leaf tasks. In all the following figures, the x-axis represents the number of rows of the input matrix.

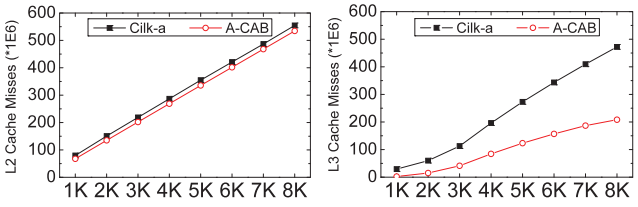
We use *Heat* and *SOR* as benchmarks to evaluate the scalability of A-CAB for applications with balanced execution DAGs. Other benchmarks, such as *GE*, have similar results.

Fig. 10 shows the performance of *Heat* and *SOR* with different input data sizes in Cilk-a and A-CAB. From



(a) Performance of Heat (b) Performance of SOR

Fig. 10. Performance of Heat and SOR with different input data sizes.



(a) L2 cache misses of Heat (b) L3 cache misses of Heat

Fig. 11. L2 and L3 cache misses of Heat with different input data sizes.

Fig. 10, we can see that *Heat* and *SOR* achieve better performance in A-CAB for all sizes of the input data up to 8,192 rows compared with Cilk-a. When the input data size is small (i.e., $1{,}024 \times 512$), A-CAB reduces 40.4 percent execution time of *Heat* and reduces 56.1 percent execution time of *SOR*. When the input data size is large (i.e., $8{,}192 \times 512$), A-CAB reduces 12.3 percent execution time of *Heat* and reduces 21.1 percent execution time of *SOR*.

Fig. 11 shows the L2 and L3 cache misses of *Heat* with different input data sizes in Cilk-a and A-CAB. Observed from the figure, we can find that both the shared cache misses and the private cache misses are reduced in A-CAB compared with Cilk-a. The better performance of *Heat* in A-CAB results from the less cache misses in A-CAB compared with Cilk-a. When the input data size is small ($1{,}024 \times 512$), A-CAB can reduce 76.1 percent L3 cache misses and 15.2 percent L2 cache misses compared with Cilk-a. When the input data size is large ($8{,}192 \times 512$), A-CAB can reduce 55.9 percent L3 cache misses and 3.6 percent L2 cache misses compared with Cilk-a. Therefore, when A-CAB schedules regular applications with balanced execution DAGS, it is scalable. Other benchmarks show similar results of cache misses.

In addition, Fig. 11 further verifies that A-CAB can also slightly reduce private cache misses by scheduling tasks with shared data into the same socket, which is due to the same reason explained previously.

We further discuss the impact of task granularity on the performance of memory-bound applications in A-CAB, and prove that the extra overhead of A-CAB for CPU-bound applications is negligible in Sections 5.3 and 5.4 of the supplemental document, available online.

5 Related Work

Work-stealing is increasingly popular for automatic load balancing inside parallel applications. There has been a lot of research work on its adaptation and improvement [21], [11], [30], [22], [27], [13].

There are generally two policies for work-stealing: child-first and parent-first. In [19], the performance of the two policies was compared. Both child-first and parent-first policies have their strengths and are used pervasively in work-stealing schedulers. For example, MIT Cilk [9], Cilk++ [26], and Intel TBB [29] use the child-first policy, while Java's fork-join framework [23] and Task Parallel Library (TPL) [25] use the parent-first policy. Also there are some work-stealing schedulers that adopt both policies, for example, SLAW [20]. In SLAW, tasks are generated following either the child-first policy or the parent-first policy according to the stack pressure and work-stealing conditions. Although SLAW uses both policies as in our A-CAB scheduler, it does not associate the policies to the DAG tiers as we do in A-CAB.

Cache awareness is an interesting issue in work-stealing. In [1], a theoretical bound on the number of cache misses for traditional work-stealing was presented and a locality-guided work-stealing algorithm was implemented on a single-socket SMP. In [15], the authors analyzed the cache misses of algorithms using traditional work-stealing, focusing on the effects of false sharing. In [14], cache behaviors of work-stealing and a parallel depth-first scheduler were compared and analyzed on a multicore simulator that has shared L2 caches among cores. It was proposed to promote constructive cache sharing through controlling task granularity. However, the above studies did not take the MSMC architecture into consideration.

In [28], Probability Work-Stealing (PWS) and Hierarchical Work-Stealing (HWS) were proposed to reduce communications among different computers for hierarchical distributed platform. In PWS, processors had higher probability to steal tasks from processors in the same computer. HWS used a rigid boundary level to divide tasks into global tasks and local tasks which are similar to intersocket tasks and intrasocket tasks in A-CAB. It is also worth noting that PWS and HWS were proposed for reduction of communications in distributed environments.

Similar to HWS, our previous work CAB [13] used a rigid boundary level to divide tasks into global tasks and local tasks. Though the boundary level is calculated at runtime, users have to provide a number of command line arguments for the scheduler to calculate the boundary level. If the arguments are not correct, the performance of applications may degrade seriously. In addition, CAB is not as adaptive as A-CAB because it cannot work with irregular and unbalanced execution DAGs that A-CAB works with.

The method of dividing an execution DAG into sub-DAGs for reducing cache misses was also used in other studies. In [5], CONTROLLED-PDF was proposed to reduce cache misses in single-socket multicore architecture. The scheduler divided nodes of a DAG into *L2-supernodes* that contain data fit for the shared L2 cache and further divided L2-supernodes into *L1-supernodes* with data fit for the private L1 cache. By executing L1-supernodes in the same L2-supernode in parallel but executing L2-supernodes sequentially, the cache misses can be reduced. The scheduler needed users to provide space complexity function of the executed program and was only applicable to single-socket multicore architecture. Also, the paper did not provide the detailed method for creating L1-supernodes and L2-supernodes and did not evaluate the proposed scheduler through experiment as we do.

There are also some studies aiming at good cache performance based on other techniques. Cache-oblivious algorithms can achieve good cache performance by tuning the original algorithms carefully [7]. In [16], ULCC was proposed to explicitly manage and optimize last level cache usage by allocating proper cache space for different data sets of different threads based on a page-coloring technique. Although ULCC provides a good way to manage the last

level cache, the management is still burdensome for programming. In contrast, A-CAB can improve the last level cache (L3) performance of memory-bound applications automatically.

6 Conclusions and Future Work

Although traditional work-stealing works efficiently in a multicore processor, it tends to pollute the shared caches in MSMC architectures. To solve the problem, we have designed and implemented A-CAB that consists of a DAG partitioner and a bitier work-stealing scheduler. For an iterative program, by profiling it at its first iteration, The DAG partitioner uses the profiling-based method to divide the execution DAG into intersocket tier and intrasocket tier without human intervention. For a noniterative program, with the profiling-based method, the DAG partitioner divides the execution DAG into intersocket tier and intrasocket tier based on compiler-provided and user-provided information. With bitier work-stealing, A-CAB reduces the shared cache misses significantly by scheduling tasks of an intrasocket subtree within the same socket. Experimental results demonstrate that A-CAB can achieve up to 74.4 percent performance gain for memory-bound applications compared with traditional work-stealing. The extra overhead of A-CAB for CPU-bound applications is negligible.

One future research direction is to improve A-CAB for more complex architectures such as NUMA and cc-NUMA architectures.

Acknowledgments

This work was partially supported by 863 program 2011AA01A202, Shanghai Excellent Academic Leaders Plan (No. 11XD1402900), NSFC (Grant No. 60725208, 61003012), and National Science Fund for Distinguished Young Scholars with Grant Nos. 61028005. Quan Chen would like to thank the University of Otago for hosting his internship during the course of this research.

References

[1] 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.

[2] 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 Trans. Parallel and Distributed Systems,* vol. 20, no. 3, pp. 404-418, Mar. 2009.

[3] R. Azimi, M. Stumm, and R. Wisniewski, "Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters," *Proc. 19th Ann. Int'l Conf. Supercomputing,* pp. 101-110, 2005.

[4] M. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations," *J. Computational Physics,* vol. 53, no. 3, pp. 484-512, 1984.

[5] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms," *Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 501-510, 2008.

[6] G. Blelloch, J. Fineman, P. Gibbons, and H.V. Simhadri, "Scheduling Irregular Parallel Computations on Hierarchical Caches," *Proc. 20th ACM Symp. Parallel Algorithms and Architectures,* June 2011.

[7] G. Blelloch, P. Gibbons, and H. Simhadri, "Low Depth Cache-Oblivious Algorithms," *Proc. 22nd ACM Symp. Parallelism in Algorithms and Architectures,* pp. 189-199, 2010.

[8] R.D. Blumofe, "Executing Multithreaded Programs Efficiently," PhD thesis, Dept. of Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, Technical Report MIT/LCS/TR-677, MIT Laboratory for Computer Science, Sept. 1995.

[9] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *J. Parallel and Distributed computing,* vol. 37, no. 1, pp. 55-69, Aug. 1996.

[10] D. Butenhof, *Programming with POSIX Threads.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[11] D. Chase and Y. Lev, "Dynamic Circular Work-Stealing Deque," *Proc. 17th Ann. ACM Symp. Parallelism Algorithms and Architectures,* pp. 21-28, 2005.

[12] Q. Chen, M. Guo, and Z. Huang, "Cats: Cache Aware Task-Stealing Based on Online Profiling in Multi-Socket Multi-Core Architectures," *Proc. 26th Int'l Conf. Supercomputing,* pp 163-172, 2012.

[13] Q. Chen, Z. Huang, M. Guo, and J. Zhou, "CAB: CachE-Aware Bi-Tier Task-Stealing In Multi-Socket Multi-Core Architecture," *Proc. 40th Int'l Conf. Parallel Processing,* pp. 722-732, 2011.

[14] S. Chen et al., "Scheduling Threads for Constructive Cache Sharing on CMPs," *Proc. 19th Ann. ACM Symp. Parallel Algorithms and Architectures,* pp. 105-115, 2007.

[15] R. Cole and V. Ramachandran, "Analysis of Randomized Work Stealing with False Sharing," *ArXiv e-prints,* Mar. 2011.

[16] X. Ding, K. Wang, and X. Zhang, "ULCC: A User-Level Facility for Optimizing Shared Cache Performance on Multicores," *Proc. ACM SIGPLAN Symp. Principles and Practice Parallel Programming,* pp. 103-112, 2011.

[17] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *J. Parallel and Distributed Computing,* vol. 16, no. 4, pp. 276-291, 1992.

[18] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1999.

[19] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism," *Proc. IEEE 23th Int'l Parallel and Distributed Processing Symp.,* pp. 1-12, 2009.

[20] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: A Scalable Locality-Aware Adaptive Work-Stealing Scheduler," *Proc. IEEE 24th Int'l Parallel and Distributed Processing Symp.,* pp. 1-12, 2010.

[21] D. Hendler, Y. Lev, M. Moir, and N. Shavit, "A Dynamic-Sized Nonblocking Work Stealing Deque," Technical Report TR-2005-144, Sun Microsystems, Inc., p. 69, 2005.

[22] D. Hendler and N. Shavit, "Non-Blocking Steal-Half Work Queues," *Proc. 21th Ann. Symp. Principles Distributed Computing,* pp. 280-289, 2002.

[23] D. Lea, "A Java Fork/Join Framework," *Proc. ACM Conf. Java Grande,* pp. 36-43, 2000.

[24] J. Lee and J. Palsberg, "Featherweight X10: A Core Calculus for Async-Finish Parallelism," *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Computing,* pp. 25-36, 2010.

[25] 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.

[26] C. Leiserson, "The Cilk++ Concurrency Platform," *Proc. 46th Ann. Design Automation Conf.,* pp. 522-527, 2009.

[27] M.M. Michael, M.T. Vechev, and V.A. Saraswat, "Idempotent Work Stealing," *Proc. 14th ACM SIGPLAN Symp. Principles and Practice Parallel Programming,* pp. 45-54, 2009.

[28] J.-N. Quintin and F. Wagner, "Hierarchical Work-Stealing," *Proc. 16th Int'l Euro-Par Conf. Parallel processing: Part I,* pp. 217-229, 2010.

[29] J. Reinders, *Intel Threading Building Blocks.* O'Reilly, 2007.

[30] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P. Yew, "An Adaptive Task Creation Strategy for Work-Stealing Scheduling," *Proc. IEEE/ACM Eighth Ann. Int'l Symp. Code Generation and Optimization,* pp. 266-277, 2010.

[31] J. Zhang, Z. Huang, W. Chen, Q. Huang, and W. Zheng, "Maotai: View-Oriented Parallel Programming on CMT Processors," *Proc. 37th Int'l Conf. Parallel Processing,* pp. 636-643, 2008.

Quan Chen received the BS degree in computer science from Tongji University, China, in 2007, the MS degree in computer science from the Shanghai Jiao Tong University, China, in 2009, and is currently working toward the PhD degree at Embedded and Pervasive Computing Center (EPCC), Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include parallel and distributed processing, task scheduling, cloud computing, and chip multiprocessor.

Minyi Guo received the BS and ME degrees in computer science from Nanjing University, China, in 1982 and 1986, respectively, and the PhD degree in information science from the University of Tsukuba, Japan, in 1998. From 1998 to 2000, he had been a research associate of NEC Soft, Ltd. Japan. He was a visiting professor at the Department of Computer Science, Georgia Institute of Technology. He was a full professor at the University of Aizu, Japan, and is the head of the Department of Computer Science and Engineering at Shanghai Jiao Tong University, China. His primary interests include automatic parallelization and data-parallel languages, bioinformatics, compiler optimization, high-performance computing, and pervasive computing. He is a senior member of the IEEE and has published more than 150 papers in well-known conferences and journals.

Zhiyi Huang received the BSc degree in 1986 and the PhD degree in 1992 in computer science from the National University of Defense Technology (NUDT) in China. He is an associate professor at the Department of Computer Science, University of Otago, New Zealand. He was a visiting professor at EPFL (Swiss Federal Institute of Technology Lausanne) and Tsinghua University in 2005, and a visiting scientist at MIT CSAIL in 2009. His research fields include parallel/distributed computing, multicore architectures, operating systems, green computing, cluster/grid/cloud computing, high-performance computing, and computer networks.