Locality-Aware Work Stealing Based on Online Profiling and Auto-Tuning for Multisocket Multicore Architectures

QUAN CHEN, School of Software Engineering, Shanghai Jiao Tong University
MINYI GUO, Department of Computer Science and Engineering, Shanghai Jiao Tong University

Modern mainstream powerful computers adopt multisocket multicore CPU architecture and NUMA-based memory architecture. While traditional work-stealing schedulers are designed for single-socket architectures, they incur severe shared cache misses and remote memory accesses in these computers. To solve the problem, we propose a locality-aware work-stealing (LAWS) scheduler, which better utilizes both the shared cache and the memory system. In LAWS, a load-balanced task allocator is used to evenly split and store the dataset of a program to all the memory nodes and allocate a task to the socket where the local memory node stores its data for reducing remote memory accesses. Then, an adaptive DAG packer adopts an auto-tuning approach to optimally pack an execution DAG into cache-friendly subtrees. After cache-friendly subtrees are created, every socket executes cache-friendly subtrees sequentially for optimizing shared cache usage. Meanwhile, a triple-level work-stealing scheduler is applied to schedule the subtrees and the tasks in each subtree. Through theoretical analysis, we show that LAWS has comparable time and space bounds compared with traditional work-stealing schedulers. Experimental results show that LAWS can improve the performance of memory-bound programs up to 54.2% on AMD-based experimental platforms and up to 48.6% on Intel-based experimental platforms compared with traditional work-stealing schedulers.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Memory subsystem, History-based auto-tuning, Task scheduling

ACM Reference Format:
Quan Chen and Minyi Guo. 2015. Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures. ACM Trans. Architec. Code Optim. 12, 2, Article 22 (July 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2766450

22

Extension of Conference Paper. This article is extended for previous conference paper "LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures" [Chen et al. 2014], which was published in the International Conference on Supercomputing (ICS 2014). The 30% new material comes from two aspects:

—We have analyzed the theoretical time and space bounds of LAWS. Based on our analysis, the theoretical time and space bounds are comparable to the original random work-stealing scheduler.
—This article has also significantly enhanced the experimental evaluation. We have evaluated the performance of LAWS on both Intel-based platforms and AMD-based platforms.

MINYI GUO is the correspondence author of this article.
This work is partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC) (61261160502, 61272099), the Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), the Scientific Innovation Act of STCSM (13511504200), and the EU FP7 CLIMBER project (PIRSES-GA-2012-318939).
Authors' addresses: Q. Chen and M. Guo, 3-119/3-417, SEIEE building, No. 800 Dongchuan Road, Shanghai, China; email: chen-quan@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn.

DOI: http://dx.doi.org/10.1145/2766450

1. INTRODUCTION

Although hardware manufacturers keep increasing cores in CPU chips, the number of cores cannot be increased unlimitedly due to physical limitations. To meet the urgent need for powerful computers, multiple CPU chips are integrated into a multisocket multicore (MSMC) architecture, in which each CPU chip has multiple cores with a shared last-level cache and is plugged into a socket.

To efficiently utilize the cores, programming environments with dynamic load-balancing policies are proposed. *Work sharing* [Ayguadé et al. 2009] and *work stealing* [Blumofe 1995] are the two best-known dynamic load-balancing policies. For instance, TBB [Reinders 2007], XKaapi [Gautier et al. 2013b], Cilk++ [Leiserson 2009], and X10 [Lee and Palsberg 2010] use work stealing, and OpenMP [Ayguadé et al. 2009] uses work sharing. With dynamic load-balancing polices, the execution of a parallel program is divided into a large amount of fine-grained tasks and is expressed by a task graph (aka directed acyclic graph or DAG [Gerasoulis and Yang 1992]). Each node in a DAG represents a task (i.e., a set of instructions) that must be executed sequentially without preemption.

While all the workers (threads, cores) share a central task pool in work sharing, work stealing provides an individual task pool for each worker. In work stealing, most often each worker pushes tasks to and pops tasks from its task pool without locking. When a worker's task pool is empty, it tries to steal tasks from other workers, and that is the only time it needs locking. Since there are multiple task pools for stealing, the lock contention is low even at task steals. Therefore, work stealing performs better than work sharing due to its lower lock contention.

However, modern shared-memory MSMC computers and large-scale supercomputing systems often employ NUMA-based (*nonuniform memory access*) memory systems, in which the whole main memory is divided into multiple memory nodes and each node is attached to the socket of a chip. The memory node attached to a socket is called its local memory node, and those that are attached to other sockets are called remote memory nodes. The cores of a socket access its local memory node much faster than the remote memory nodes. Traditional work stealing is inefficient in this architecture.

Figure 1 gives an example of MSMC computers that employ NUMA-based memory systems. As shown in the figure, different sockets are connected through interconnect models. For example, in an Intel-based machine, QPI [Intel 2009] is used to connect different sockets, while in an AMD-based machine, HyperTransport [Consortium 2010] is used to connect different sockets. In traditional work stealing, since a free worker *randomly* selects victim workers to steal new tasks when its own task pool is empty, the tasks are distributed to all the workers nearly randomly. This randomness can cause more accesses to remote memory in NUMA, as well as more shared cache misses inside a CPU chip, which often degrades the performance of memory-bound applications in MSMC architectures (to be discussed in detail in Section 2).

To reduce remote memory accesses and shared cache misses, this article proposes a locality-aware work-stealing (LAWS) scheduler that automatically schedules tasks to the sockets where the local memory nodes store their data and executes the tasks inside each socket in a shared-cache-friendly manner. LAWS targets iterative divide-and-conquer applications that have tree-shaped execution DAGs. While existing work-stealing schedulers incur bad data locality, to the best of our knowledge, LAWS is the first locality-aware work-stealing scheduler that improves the performance of memory-bound programs leveraging both NUMA optimization and shared cache optimization.

The main contributions of this article are as follows:

—We propose a load-balanced task allocator that automatically allocates a task to the particular socket where the local memory node stores its data and that can balance the workload among sockets.

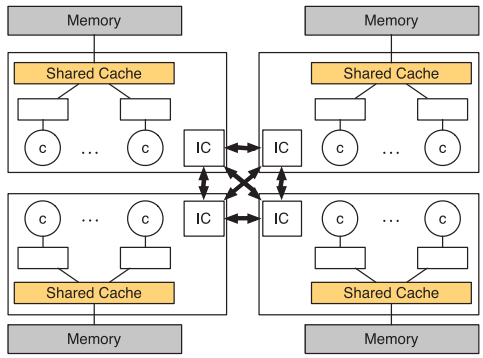


Fig. 1. An example of MSMC architectures that employ NUMA-based memory systems.

—We propose an adaptive DAG packer that can further pack an execution DAG into *cache-friendly subtrees* (CF subtrees) for optimizing shared cache usage based on online-collected information and auto-tuning.
—We propose a triple-level work-stealing scheduler to schedule tasks accordingly so that a task can access its data from either the shared cache or the local memory node other than the remote memory nodes.
—We demonstrate that LAWS significantly improves the performance of memory-intensive applications. The experiment shows that LAWS can achieve a performance gain of up to 54.2% on an AMD-based experimental platform and up to 48.6% on an Intel-based experimental platform for memory-intensive applications.

The rest of this article is organized as follows. Section 2 explains the motivation of LAWS. Section 3 presents locality-aware work stealing, including the balanced data allocator, adaptive DAG packer, and triple-level work-stealing scheduler. Section 4 gives the implementation of LAWS. Section 5 validates LAWS theoretically and analyzes the time and space bounds of programs in LAWS. Section 6 evaluates LAWS. Section 7 discusses the related work. Section 8 draws conclusions.

2. MOTIVATION

Similar to many popular work-stealing schedulers (e.g., Cilk [Blumofe et al. 1996] and CATS [Chen et al. 2013]), this article targets iterative *divide-and-conquer* (D&C) programs that have tree-shaped execution DAGs. Most stencil programs [Shaheen and Strzodka 2012] and algorithms based on jacobi iteration (e.g., *Heat Distribution* and *Successive Over Relaxation*) are examples of iterative D&C programs.

Figure 2(a) gives a general execution DAG for iterative D&C programs. In a D&C program, its dataset is recursively divided into several parts until each of the leaf tasks only processes a small part of the whole dataset.

Suppose the execution DAG in Figure 2(a) runs on an MSMC architecture with a NUMA memory system as shown in Figure 2(b). In the MSMC architecture, a memory node N_i is attached to the socket ρ_i. In Linux memory management for NUMA, if a chunk of data is first accessed by a task that is running on a core of the socket ρ, a physical page from the local memory node of ρ is automatically allocated to the data. This data allocation strategy employed in Linux kernel and Solaris is called the *first touch strategy*. In this work, we take advantage of this strategy of memory allocation.

For a parallel program, its dataset is often first accessed by tasks in the first iteration or an independent initialization phase. By scheduling these tasks to different sockets, the whole dataset of the program that has the execution DAG in Figure 2(a) is split and stored in different memory nodes as in Figure 2(b) due to the first touch strategy.

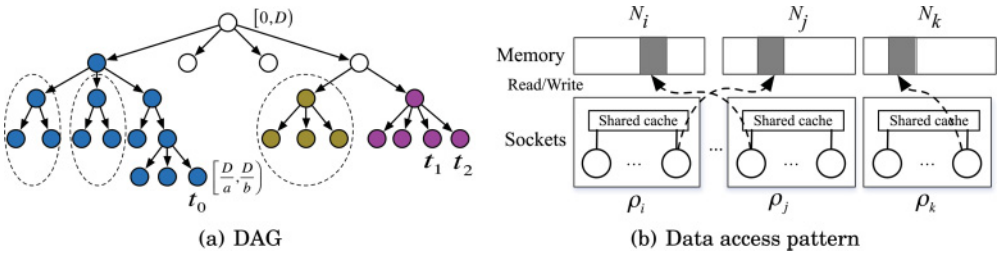


Fig. 2. An example DAG and the data access pattern in traditional work stealing on MSMC architecture.

However, traditional work stealing suffers from two main problems when scheduling the execution DAG in Figure 2(a) in MSMC architectures. First, most tasks have to access their data from remote memory nodes in all the iterations. Second, the shared caches are not utilized efficiently.

As for the first problem, suppose the whole dataset of the program in Figure 2(a) is [0, D), and the task t_0 is the first task that accesses the part of the data $[\frac{D}{a}, \frac{D}{b})$ $(a > b \geq 1)$. If task t_0 is scheduled to socket ρ_i, the part of the data $[\frac{D}{a}, \frac{D}{b})$ is automatically allocated to the memory node, N_i, of socket ρ_i, due to the first touch strategy. Suppose task t_r in a later iteration processes the data $[\frac{D}{a}, \frac{D}{b})$. Due to the randomness of work stealing, it is very likely that t_r is not scheduled to socket ρ_i. In this situation, t_r cannot access its data from its fast local memory node; instead, it has to access a remote memory node for its data.

As for the second problem, neighbor tasks (e.g., t_1 and t_2 in Figure 2(a)) are likely to be scheduled to different sockets due to the randomness of stealing in traditional work-stealing schedulers. This causes more shared cache misses as neighbor tasks in DAG often share some data. For example, in Figure 2(a), both t_1 and t_2 need to read all their data from the main memory if they are scheduled to different sockets. However, if we could schedule t_1 and t_2 to the same socket, their shared data is only read into the shared cache once by one task, while the other task can read the data directly from the shared cache.

To solve the two problems, we propose the LAWS scheduler that consists of a load-balanced task allocator, an adaptive DAG packer, and a triple-level work-stealing scheduler. The load-balanced task allocator can evenly distribute the dataset of a program to all the memory nodes and allocate a task to the socket where the local memory node stores its data. The adaptive DAG packer can pack the execution DAG of a program into CF subtrees so that the shared cache of each socket can be used effectively. The triple-level work-stealing scheduler schedules tasks accordingly to balance the workload and reduce shared cache misses.

LAWS ensures that the workload is balanced and most tasks can access data from either the shared cache or the local memory node. The performance of memory-bound programs can be improved due to balanced workload and shorter data access latency.

3. LOCALITY-AWARE WORK STEALING

In this section, we first give a general overview of the design of LAWS. Then, we present the load-balanced task allocator, the adaptive DAG packer, and the triple-level work-stealing scheduler in LAWS, respectively. Lastly, we analyze the time and space bounds of programs in LAWS.

3.1. Design of LAWS

Figure 3 illustrates the processing flow of an iterative program in LAWS.

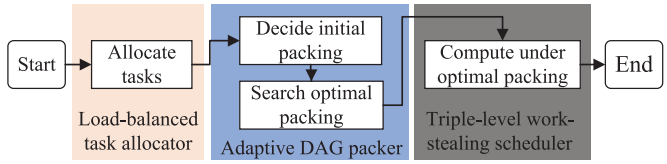


Fig. 3. The processing flow of an iterative D&C program in LAWS.

In every iteration, the task allocator carefully allocates tasks to different sockets to evenly distribute the dataset of the program to all the memory nodes and allocate each task to the socket where the local memory node stores its data. In this situation, the workload of different sockets is balanced in general since the time for processing the same amount of data is similar among tasks in D&C programs. There may be some slight load unbalance, which will be resolved by the triple-level work-stealing scheduler.

For each socket, LAWS further packs the tasks allocated to it into a number of CF subtrees based on runtime information collected in the first iteration, so that shared cache can be better utilized. For example, in Figure 2(a), the subtree in each ellipse is a CF subtree. In the first several iterations, the packer automatically adjusts the packing of tasks to search for the optimal one that results in the minimum makespan. Because the execution DAGs of different iterations are the same and the tasks in the same position of the execution DAGs work on the same part of the dataset in D&C programs, the optimal packing for the completed iterations is also optimal for future iterations. Once the optimal packing is found, LAWS packs the tasks in all the following iterations in a way suggested by the optimal packing.

LAWS adopts a triple-level work-stealing scheduler to schedule tasks in each iteration. The tasks in the same CF subtrees are scheduled within the same socket. If a socket completes all its CF subtrees, it steals a CF subtree from a randomly chosen socket in order to resolve the possible slight load unbalance from the task allocator.

Because tasks in the same CF subtree often share some data, the shared data is only read into the shared cache once but can be accessed by all the tasks of the same CF subtree. In this way, the shared cache can be better utilized.

It is worth noting that LAWS does not need users to provide any information. All the information needed is obtained automatically at runtime by LAWS.

3.2. Load-Balanced Task Allocator

The load-balanced task allocator is proposed based on an assumption that a task divides its dataset into several parts evenly according to its branching degree. This assumption is true in most of the current D&C programs.

The load-balanced task allocator should satisfy two main constraints when allocating tasks to sockets. First, to balance workload, the size of data processed by tasks allocated to each socket should be the same in every iteration. Second, to reduce shared cache misses, the adjacent data should be stored in the same memory node since adjacent data is processed by neighbor tasks that should be scheduled to the same socket. Traditional work-stealing schedulers do not satisfy the two constraints due to the randomness of stealing.

Suppose a program runs on an M-socket architecture. If its dataset is D, to balance workload, the tasks allocated to each socket need to process $\frac{1}{M}$ of the whole dataset. Note that, in the load-balanced task allocator, we do not need to know the real value of D. We use D to represent the whole dataset for ease of description. Without loss of generality, LAWS makes sure that the tasks allocated to the ith ($1 \leq i \leq M$) socket

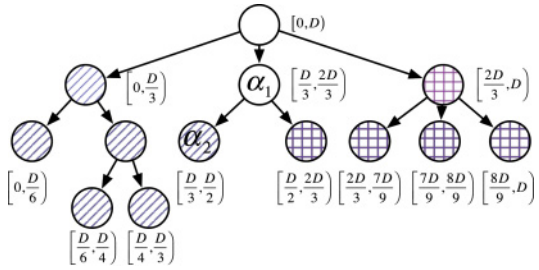


Fig. 4. Allocate the tasks to the two sockets of a dual-socket architecture.

should process the part of the whole dataset ranging from $(i-1)\times\frac{D}{M}$ to $i\times\frac{D}{M}$ (denoted by $[(i-1)\times\frac{D}{M}, i\times\frac{D}{M})$).

To achieve this objective, we need to find out which task processes which part of the whole dataset. For a task α_2 in Figure 4, to find out its part, LAWS analyzes the structure of the dynamically generated execution DAG when α_2 is spawned. Suppose task α_2 is task α_1's ith subtask and the branching degree of α_1 is b. If α_1 processes the part of data $[D_s, D_e)$, Equation (1) gives the part of data that α_2 will process:

$$\left[(i-1)\times\frac{D_e-D_s}{b}+D_s, i\times\frac{D_e-D_s}{b}+D_s\right). \tag{1}$$

Figure 4 gives an example of allocating the tasks to the two sockets of a dual-socket architecture. The range of data beside each task is calculated according to Equation (1).

In the dual-socket architecture, the tasks that process the datasets $[0, \frac{D}{2})$ and $[\frac{D}{2}, D)$ should be allocated to the first socket and the second socket, respectively. For instance, in Figure 4, because α_2 is responsible for processing data range $[\frac{D}{3}, \frac{D}{2})$ that is within $[0, \frac{D}{2})$, it should be allocated to the first socket. For the same reason, the slash-shaded tasks are allocated to the first socket and the mesh-shaded tasks are allocated to the second socket. If a task is allocated to a socket, all its child tasks are allocated to the same socket. For example, all the tasks rooted with α_2 will be allocated to the first socket.

Because the task allocator allocates a task according to the range of its dataset, in the following iterations, the tasks processing the same part of the whole dataset will be allocated to the same socket. In this way, the tasks in all the iterations can find their data in the local memory node. Therefore, the first problem discussed in Section 2 in traditional work stealing will be solved.

3.3. Adaptive DAG Packer

After the tasks are allocated to appropriate sockets, each socket will still have to execute a large number of tasks. The data involved in these tasks is often too large to fit into the shared cache of a socket. To utilize the shared cache efficiently, LAWS further packs the tasks allocated to each socket into CF subtrees that will be executed sequentially.

It is worth noting that the work-stealing scheduler and each task often generate some intermediate data during the execution of a program. Therefore, the precise size of data involved in each task is not known during the execution of a parallel program, even if the size of the whole input data of the program is known. It is not trivial to further pack tasks into CF subtrees lacking of precise data usage information. To solve this problem, as described later, we use an auto-running approach based on online-collected profiling information to search for the optimal packing.

3.3.1. Decide Initial Packing. LAWS makes sure that the data accessed by all the socket-local tasks in each CF subtree can be fully stored in the shared cache of a socket. Note that the tasks in the same CF subtree (called *socket-local tasks*) are scheduled in the same socket and the root task of a CF subtree is called a *CF root task*. In this way, the data shared by tasks in the same CF subtree is read into the shared cache once but can be shared and accessed by all the tasks.

To achieve the previous objective, we need to know the size of the shared cache used by each task, which cannot be collected directly. To circumvent this problem, in the first iteration, for any task α, LAWS collects the number of last-level private cache (e.g., L2) misses caused by it. The size of the shared cache used by α can be estimated as the number of the above cache misses times the cache line size (e.g., 64 bytes).

The approximation is reasonable for two reasons. First, the core c that executes α does not execute other tasks concurrently. All the last-level private cache misses of c during the execution are caused by α. Second, once a last-level private cache miss happens, c accesses the shared cache or memory and will use a cache line in the shared cache.

For task α, we further calculate its *SOSC*, which represents the *size of shared cache used by all the tasks in the subtree rooted with* α. The SOSC of α is calculated in a bottom-up manner. Suppose α has m direct child tasks $\alpha_1, \ldots, \alpha_m$ and their SOSCs are $S_1, \ldots, S_m$, respectively. The SOSC of α (denoted by S_α) can be calculated in Equation (2), where M_α equals the number of last-level cache misses caused by α itself times the cache line size:

$$S_\alpha = M_\alpha + \sum_{i=1}^{m} S_i. \tag{2}$$

Once all the tasks in the first iteration are completed, SOSCs of all the tasks are calculated. Based on SOSCs of all the tasks, the DAG packer can group the tasks into CF subtrees by identifying all the CF root tasks as follows.

Let S_c represent the shared cache size of a socket. Suppose α's parent task is β, and their SOSCs are S_α and S_β, respectively. Then, if $S_\alpha \leq S_c$ and $S_\beta > S_c$, α is a CF root task, which means all the data involved in the descendent tasks of α just fit into the shared cache. If $S_\beta < S_c$, α is a socket-local task.

Once all the CF root tasks are identified, the initial packing of tasks into CF subtrees is determined.

3.3.2. Search-Optimal Packing. If S_α in Equation (2) precisely equals the real size of shared cache used by the subtree rooted with α, the data involved in any CF subtree would not exceed the capacity of a socket's shared cache.

However, S_α is only a close approximation for the following reasons. Suppose tasks α_1 and α_2 in the subtree rooted with α share some data. Although they are allocated to the same socket by the load-balanced task allocator, they can be executed by different cores. In this case, both α_1 and α_2 need to read the shared data to the last-level private cache, and thus the size of the shared data is accumulated twice in Equation (2). On the other hand, if some data stored in the shared cache has already been prefetched into the private cache, it does not incur last-level private cache misses and the size of the prefetched data is missed in Equation (2). The multiple accumulation of shared data and the prefetching make S_α of Equation (2) slightly larger or smaller than the actual size of shared cache used by the subtree rooted with α.

Therefore, the initial packing of tasks into CF subtrees is only a near-optimal packing. LAWS further uses an auto-tuning approach to search the optimal packing. In the approach, LAWS packs tasks into CF subtrees differently in different iterations, records the execution time of each iteration, and chooses the packing that results in the shortest makespan as the optimal packing.

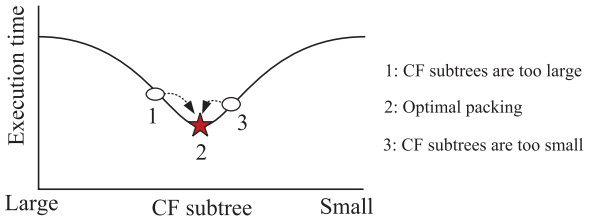


Fig. 5. Execution time of an iteration when the execution DAG is packed differently.

Figure 5 shows the execution time of an iteration when tasks are packed differently. If CF subtrees are too large (contain too many socket-local tasks, point 1 in Figure 5), the data accessed by tasks in each CF subtree cannot be fully stored in the shared cache of a socket. On the other hand, if CF subtrees are too small (contain too few socket-local tasks, point 3 in Figure 5), the data accessed by tasks in each CF subtree is too small to fully utilize the shared cache.

Starting from the packing of the execution DAG into CF subtrees in Section 3.3.1, LAWS first evaluates smaller CF subtrees. If smaller CF subtrees result in shorter execution time, CF subtrees in the initial packing are too large. In this case, LAWS evaluates smaller and smaller CF subtrees until the packing that results in the shortest execution time (point 2 in Figure 5) is found. If smaller CF subtrees result in longer execution time, CF subtrees in the initial packing are too small. In this case, LAWS evaluates larger and larger CF subtrees instead until the optimal packing is found.

ALGORITHM 1: Algorithm for searching the optimal way to pack an execution DAG into CF subtrees

```
   Input: α_1, ..., α_m (CF root tasks in the initial packing)
   Input: T (Execution time under the initial packing)
   Output: Optimal CF root tasks
 1 int T_n = 0, T_c = T;                          // New & current makespan
 2 int EvalLarger = 1;                            // Eval. larger subtrees?
 3 while CF root tasks have child tasks do
 4 |   Set child tasks of the current CF root tasks as the new CF root tasks;
 5 |   Execute an iteration under the new packing;
 6 |   Record the execution time T_n;
 7 |   if T_n < T_c then // Point 1 in Figure 5
 8 |   |   T_c = T_n; Save new CF root tasks; EvalLarger = 0;
 9 |   else break
10 if EvalLarger == 1 then // Point 3 in Figure 5
11 |   Restore CF root tasks to {α_1, ..., α_m};
12 |   T_c = T;
13 |   while CF root tasks have parent tasks do
14 |   |   Set parent tasks of the current CF root tasks as the new CF root tasks;
15 |   |   Execute an iteration under the new packing;
16 |   |   Record the execution time T_n;
17 |   |   if T_n > T_c then break;
18 |   |   else T_c = T_n; Save new CF root tasks
```

Algorithm 1 gives the auto-tuning algorithm for searching the optimal way to pack the tasks allocated to a socket into CF subtrees. To generate larger or smaller CF subtrees, we select the parent tasks or child tasks of the current CF root tasks as the new CF root tasks.

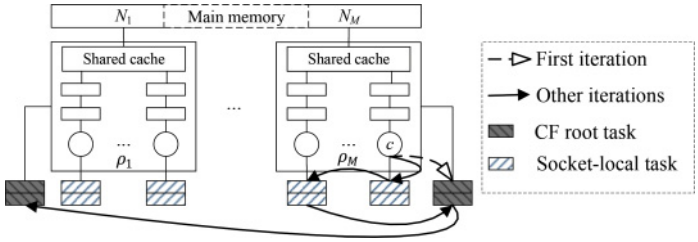


Fig. 6. Architecture of LAWS on an M-socket multicore architecture.

Since the initial packing is already near optimal, LAWS can find the optimal packing in a few iterations. Theoretically, there is a small possibility that some CF subtrees are too large while some other CF subtrees are too small. However, since there are a great many CF subtrees in an execution DAG, it is too complex to tune the size of every CF subtree independently in a small number of iterations at runtime. To simplify the problem, we increase or decrease the size of all the CF subtrees at the same time in Algorithm 1 of this article. Actually, according to the experiment in Section 6.3, our current auto-tuning strategy in Algorithm 1 works efficiently.

The approach of packing DAGs into CF subtrees in LAWS partially originates from CATS [Chen et al. 2012], which also packs the execution DAGs of parallel programs into subtrees for optimizing shared cache usage. However, once a DAG is packed in CATS, the packing cannot be adjusted even if the packing is not optimal. The experiment in Section 6.3 shows that the performance of applications can be further improved with the auto-tuning algorithm described in Algorithm 1. Worse, CATS did not consider the NUMA memory system at all and suffered from a large amount of remote memory accesses. We will further compare the performance of CATS and LAWS in detail in Section 6.

3.4. Triple-Level Work-Stealing Scheduler

Figure 6 gives the architecture of LAWS on an M-socket multicore architecture and illustrates the triple-level work-stealing policies in LAWS. In Figure 6, the main memory is divided into M memory nodes and node N_i is the local memory node of socket ρ_i. In each socket, core "0" is selected as the head core of the socket.

For each socket, LAWS creates a *CF task pool* to store CF root tasks allocated to the socket and the tasks above the CF root tasks in the execution DAG. For each core, LAWS creates a *socket-local task pool* to store socket-local tasks.

Suppose a core c in socket ρ is free; in different phases, it obtains new tasks in different ways as follows.

In the first iteration of an iterative program (and the independent initialization phase if the program has the phase), there is no socket-local task and all the tasks are pushed into CF task pools since the tasks have not been packed into CF subtrees. In the period, c can only obtain a new task from the CF task pool of ρ. Core c is not allowed to steal a task from other sockets because the dataset of a task will be stored in the wrong memory node if it is stolen in the first iteration due to the first touch strategy.

Starting from the second iteration, the tasks in each iteration have been packed into CF subtrees. Adopting triple-level work stealing, free core c can steal a new task from three levels: socket-local task pool of other cores in its socket ρ, CF task pool of ρ, and CF task pools of other sockets.

More precisely, when c is free, it first tries to obtain a task from its own socket-local task pool. If its own task pool is empty, c tries to steal a task from the socket-local task

pools of other cores in ρ. If the task pools of all the cores in ρ are empty and c is the head core of ρ, c tries to obtain a new CF root task from ρ's CF task pool.

LAWS allows a socket to help other sockets execute their CF subtrees. For instance, after all the tasks in the CF task pool of ρ are completed, the head core of ρ tries to steal a task from the CF task pools of other sockets. Although ρ needs a longer time to process the CF subtrees that are allocated to other sockets, the workload is balanced and the performance of memory-bound programs can be improved.

In LAWS, cores in the same socket are not allowed to execute tasks in multiple CF subtrees concurrently. This policy can avoid the situation that tasks in different CF subtrees pollute the shared caches with different datasets. A socket is only allowed to steal entire CF subtrees from other sockets for optimizing shared cache usage.

4. IMPLEMENTATION

We implement LAWS by modifying MIT Cilk, which is one of the earliest parallel programming environments that implemented work stealing [Frigo et al. 1998]. 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. 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.

Existing work-stealing schedulers adopt either a parent-first policy or a child-first policy when generating new tasks. In the parent-first policy (called *help-first policy* in Guo et al. [2009]), a core continually executes the parent task after spawning a new task. In the child-first policy (called *work-first policy* in Blumofe et al. [1996]), a core continually executes the spawned new task once the child is spawned. The parent-first policy works better when the steals are frequent, while the child-first policy works better when the steals are infrequent [Guo et al. 2009].

During the first iteration, LAWS adopts the parent-first policy to generate new tasks, because it is difficult to collect the numbers of last-level private cache misses caused by each task with the child-first policy. If a core is executing a task α, with the child-first policy, it is very likely that the core will also execute some of α's child tasks before α is completed. In this case, the number of last-level cache misses caused by α itself, which is used to calculate SOSCs of tasks, may not be collected correctly as it could include the number of last-level private cache misses of α's child tasks.

Starting from the second iteration, LAWS generates tasks above CF root tasks with the parent-first policy since the steals are frequent in the beginning of each iteration. LAWS generates socket-local tasks with the child-first policy since the steals are infrequent in each CF subtree.

We have modified the *cilk2c* compiler to support both the parent-first and child-first task-generating policy, while the original Cilk only supports the child-first policy. If a task α is spawned in the first iteration, the task is spawned with the parent-first policy and is pushed to the appropriate CF task pool based on the method in Section 3.2. If α is spawned in the later iterations and it is a socket-local task, LAWS spawns α with the child-first policy and pushes α into the socket-local task pool of the current core. Otherwise, if α is a CF root task or a task above CF root tasks and it is allocated to socket ρ, it is spawned with the parent-first policy and pushed into ρ's CF task pool.

We use the "libpfm" library in Linux kernel to program hardware performance units for collecting last-level private cache misses of each task. We have also modified the work-stealing scheduler of MIT Cilk to implement the triple-level work-stealing algorithm in Section 3.4.

5. THEORETICAL ANALYSIS

In this subsection, we analyzed D&C programs that are the targeted programs of the popular work-stealing environments, for example, TBB [Reinders 2007], Cilk++ [Leiserson 2009], and X10 [Lee and Palsberg 2010]. Generally speaking, a memory-bound D&C program has three main features. First, only leaf tasks physically access the data, while other tasks divide the dataset recursively into smaller pieces. Second, each leaf task only processes a small part of the whole dataset of the program. Third, the execution time of a leaf task is decided by its data access time.

5.1. Effectiveness Validation

Based on the three features, we prove that LAWS can improve the performance of memory-bound D&C programs theoretically.

Consider a memory-bound program that runs on an M-socket architecture. Suppose a leaf task α in its execution DAG is responsible for processing data of S bytes and α still accesses B bytes of boundary data besides its own part of data. Let V_l and V_r represent the speeds (bytes/cycle) of a core to access data from local memory nodes and remote memory nodes, respectively. Needless to say, $V_l > V_r$.

If we adopt a traditional work-stealing scheduler to schedule the program, the probability that α can access all the data from the local memory node is $\frac{1}{M}$. Therefore, the cycles expected for α to access all the needed data in traditional work stealing (denoted by T_R) can be calculated in Equation (3):

$$T_R = \frac{S+B}{V_l} \times \frac{1}{M} + \frac{S+B}{V_r} \times \frac{M-1}{M}. \tag{3}$$

If we adopt LAWS to schedule the program and benefit from the task allocator, α can access its own part of data from the local memory node. As a consequence, the cycles needed by α to access all the needed data in LAWS (denoted by T_L) can be calculated in Equation (4), because α also has a high chance to access its boundary data from the local memory node:

$$T_L \leq \frac{S}{V_l} + \frac{B}{V_r}. \tag{4}$$

Deduced from Equations (3) and (4), we can get Equation (5):

$$T_R - T_L \geq \left(\frac{1}{MV_r} - \frac{1}{MV_l}\right) \times [(M-1)S - B]. \tag{5}$$

In Equation (5), because $V_r < V_l$, we know that $\frac{1}{MV_r} - \frac{1}{MV_l} > 0$. Therefore, $T_R - T_L > 0$ if $(M-1)S - B > 0$ is always true in almost all the D&C programs empirically since a task's own dataset (S) is always far larger than its boundary data (B). In summary, we prove that α needs a shorter time to access all the needed data in LAWS.

Because leaf tasks need a shorter time to access their data in LAWS than in traditional work-stealing schedulers, LAWS can always improve the performance of memory-bound D&C programs even when the optimization on reducing shared cache misses in LAWS is not taken into account.

5.2. Theoretical Time and Space Bounds

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. Suppose an m-socket n-core computer is executing the execution DAG $\mathcal{G}$. For $\mathcal{G}$ on the m-socket n-core computer, Table I lists the parameters that are used to analyze $\mathcal{G}$'s time and space

Table I. Parameters Used in the Bound Analysis

Parameters	Description
$\mathcal{G}_\gamma$	The subtree rooted at task γ in $\mathcal{G}$
m	Number of sockets
n	Number of cores per socket
c	Number of overall cores ($m \times n$)
$T_1(\mathcal{G})$	The total number of nodes in $\mathcal{G}$
$T_\infty(\mathcal{G})$	The critical-path length of $\mathcal{G}$
$T_c(\mathcal{G})$	Makespan of $\mathcal{G}$ on a c-core computer
$T_c(\mathcal{G}_{free})$	Makespan of the free tier
$T_c(\mathcal{G}_{local})$	Makespan of the socket-local tier
$S_c(\mathcal{G})$	Space used by $\mathcal{G}$ on a c-core computer

bounds in LAWS. Our following discussion is based on the time and space bounds of work stealing proved in Blumofe [1995].

5.2.1. Time Bound Analysis. For $\mathcal{G}$, 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.

In LAWS, all the CF root tasks divide an execution DAG into two tiers. The tasks above the CF root tasks consist of a *free tier,* in which the tasks can be scheduled to any core; the socket-local tasks consist of a *socket-local tier*, in which the tasks can only be scheduled among cores in the same socket. Given a CF root task γ, we use the notation $\mathcal{G}_\gamma$ to represent the subtree 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 Equation (6), where $\mathcal{G}_{free}$ represents the subgraph of the free tier and k is the total number of CF root tasks:

$$T_1(\mathcal{G}) = T_1(\mathcal{G}_{free}) + \sum_{i=1}^{k} T_1(\mathcal{G}_{\gamma_i}). \tag{6}$$

The execution time of $\mathcal{G}$ in an m-socket n-core architecture, $T_c(\mathcal{G})$, can be divided into two parts: the execution time of the free tier $T_c(\mathcal{G}_{free})$ and the execution time of the socket-local tier $T_c(\mathcal{G}_{local})$. Even though the two parts can be overlapped, we use their sum to get the worst bound of $T_c(\mathcal{G})$ as shown in Equation (7):

$$T_c(\mathcal{G}) = T_c(\mathcal{G}_{free}) + T_c(\mathcal{G}_{local}). \tag{7}$$

Since the free tier is executed by m head cores using work stealing, according to the proof of Blumofe [1995], the execution time of $\mathcal{G}_{free}$ is bounded by Equation (8):

$$T_c(\mathcal{G}_{free}) \leq \frac{T_1(\mathcal{G}_{free})}{m} + T_\infty(\mathcal{G}_{free}). \tag{8}$$

For the execution of the socket-local tier, each $\mathcal{G}_{\gamma_i}$ is executed by n cores within a socket using work stealing. Therefore, the execution time of $\mathcal{G}_{\gamma_i}$ is bounded by Equation (9):

$$T_n(\mathcal{G}_{\gamma_i}) \leq \frac{T_1(\mathcal{G}_{\gamma_i})}{n} + T_\infty(\mathcal{G}_{\gamma_i}). \tag{9}$$

Since k CF root tasks are scheduled among m sockets using work stealing, the execution time of the socket-local tier is bounded by Equation (10):

$$T_c(\mathcal{G}_{local}) \leq \frac{\sum_{i=1}^{k} T_n(\mathcal{G}_{\gamma_i})}{m} + T_\infty(\mathcal{G}_{local}). \tag{10}$$

Deducing from Equations (9) and (10), we can get Equation (11):

$$T_c(\mathcal{G}_{local}) \leq \frac{\sum_{i=1}^{k} T_1(\mathcal{G}_{\gamma_i})}{m \times n} + \frac{\sum_{i=1}^{k} T_\infty(\mathcal{G}_{\gamma_i})}{m} + T_\infty(\mathcal{G}_{local}). \tag{11}$$

From Equations (7), (8), and (11), $T_c(\mathcal{G})$ can be bounded as in Equation (12):

$$T_c(\mathcal{G}) \leq \frac{T_1(\mathcal{G}_{free})}{m} + T_\infty(\mathcal{G}_{free}) + \frac{\sum_{i=1}^{k} T_1(\mathcal{G}_{\gamma_i})}{m \times n} + \frac{\sum_{i=1}^{k} T_\infty(\mathcal{G}_{\gamma_i})}{m} + T_\infty(\mathcal{G}_{local}). \tag{12}$$

After further tidying Equation (12) up, we have Equation (13):

$$T_c(\mathcal{G}) \leq \frac{T_1(\mathcal{G}_{free})}{m} + \frac{T_1(\mathcal{G}_{local})}{m \times n} + \frac{\sum_{i=1}^{k} T_\infty(\mathcal{G}_{\gamma_i})}{m} + T_\infty(\mathcal{G}). \tag{13}$$

Our experiments show that the execution time of the free tier is often less than 5% of the overall execution time. Therefore, the time bound of Equation (13) is very close to the traditional work-stealing schedulers such as MIT Cilk for D&C applications.

5.2.2. Space Bound Analysis. According to the proof of Blumofe [1995], the space used by $\mathcal{G}$ in an m-socket n-core architecture is bounded by Equation (14), where $S_1(\mathcal{G})$ denotes the space used by the serial execution of the program:

$$S_c(\mathcal{G}) \leq m \times n \times S_1(\mathcal{G}). \tag{14}$$

Equation (14) assumes that there are at most $m \times n$ workers expanding the DAG using the child-first policy. However, since LAWS uses the parent-first policy to expand the free tier quickly, each of the CF root tasks may use S_1 space in the worst case. Therefore, the space used by the LAWS scheduler $S_{m \times n}(\mathcal{G})$ can be bounded as in Equation (15):

$$S_c(\mathcal{G}) \leq \max\{k \times S_1(\mathcal{G}), m \times n \times S_1(\mathcal{G})\}. \tag{15}$$

6. EVALUATION

In this section, we evaluate the performance of LAWS. In the beginning, we introduce the two experimental hardware platforms used in the experiment. Then, on each experimental platform, we present the experimental results respectively. More precisely, we present the performance of memory-bound benchmarks in LAWS, the effectiveness of the adaptive DAG packer, and the scalability and the overhead of LAWS.

6.1. Experimental Platforms

We use an Intel server and an AMD server to evaluate the performance of LAWS. Table II lists the detailed hardware configurations. In the Intel server, Intel hyper-threading (HT) technology that delivers two processing threads per physical core is disabled.

We compare the performance of LAWS with the performance of Cilk [Blumofe et al. 1996] and CATS [Chen et al. 2012]. Cilk uses the pure child-first policy to spawn and schedule tasks. Similar to LAWS, CATS also packs the execution DAG of a parallel program into subtrees to reduce shared cache misses in MSMC architectures. Once an execution DAG is packed in CATS, the packing cannot be adjusted at runtime even if the packing is not optimal. In addition, CATS did not consider the underlying NUMA memory system.

For fairness in comparison, we also implement CATS by modifying Cilk and we have improved CATS so that it also allocates the data evenly to all the memory nodes in the first iteration as LAWS does. The Cilk programs run with CATS and LAWS without any modification. In our experiment, the number of workers (i.e., threads) launched in Cilk, CATS, and LAWS is equal to the number of physical cores in the hardware platform.

Table II. Configurations of the Experimental Platforms

AMD Server	CPU	AMD Opteron 8380
	Num of Sockets	4
	Cores per socket	4
	L2 Cache (per core)	512KB
	L3 Cache (per socket)	6MB
	DRAM	16GB
	Operating System	Linux 3.2.0-14
Intel Server	CPU	Intel Xeon X7560
	Num of Sockets	4
	Cores per socket	8 cores (16 HW threads)
	L2 Cache (per core)	2MB
	L3 Cache (per socket)	24MB
	DRAM	64GB
	Operating System	Linux 3.13.0-13

Table III. Benchmarks Used in the Experiments

Name	Bound Type	Description
Heat/Heat-ir	Memory	2D heat distribution
SOR/SOR-ir	Memory	Successive overrelaxation
GE/GE-ir	Memory	Gaussian elimination algorithm
9P/9P-ir	Memory	2D 9-point stencil computing
6P/6P-ir	Memory	3D 6-point stencil computing
25P/25P-ir	Memory	3D 25-point stencil computing
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

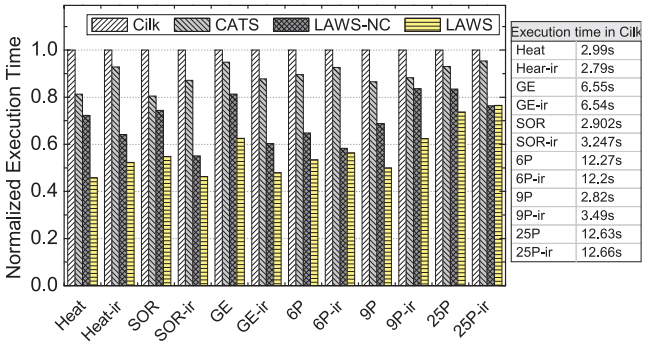
Furthermore, to avoid any performance variation due to OS-level thread scheduling, we pin each worker with an individual hardware core.

To evaluate the performance of LAWS in different scenarios, we use benchmarks listed in Table III that have both regular execution DAGs and irregular execution DAGs in the experiment. Since there are no standardized large-scale benchmarks available for work-stealing schedulers so far, most of the benchmarks are examples in the MIT Cilk package. We port the other benchmarks in the same way the examples of MIT Cilk are developed. The benchmarks we used are the same as in previous papers [Blumofe et al. 1996; Chen et al. 2012; Guo et al. 2010]. According to our experiment in Section 6.4, for current benchmarks, the larger the makespan, the better LAWS performs, which indicates the potential benefit of LAWS for large-scale benchmarks.

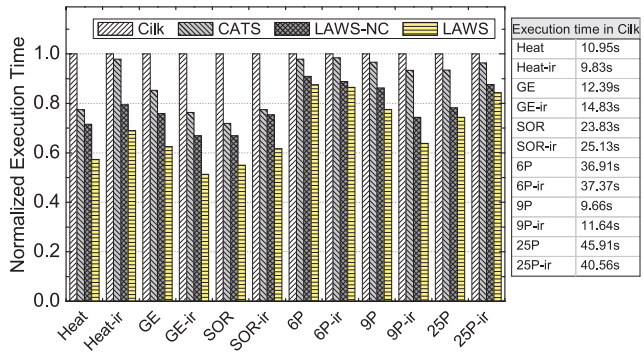
Heat-ir, *GE-ir*, *SOR-ir*, *9P-ir*, *6P-ir,* and *25P-ir* implement the same algorithm as their counterparts, respectively, except their execution DAGs are irregular. We create the programs with irregular execution DAGs in the same way as suggested in Chen et al. [2012]. If all the nodes (except the leaf tasks) in the DAG have the same branching degrees, the execution DAG is regular. All benchmarks are compiled with "-O2." For each test, every benchmark is run 10 times and the average execution time is reported as the result.

6.2. Performance of LAWS

Figure 7 shows the performance of memory-intensive benchmarks in Cilk, CATS, and LAWS on the AMD server and Intel server.



(a) AMD server



(b) Intel server

Fig. 7. The performance of memory-bound benchmarks in Cilk, CATS, and LAWS.

On the AMD server, for *Heat*, *Heat-ir*, *SOR*, *SOR-ir*, *9P,* and *9P-ir,* the input data used is an 8096×1024 matrix. For *GE* and *GE-ir*, the input data used is a 2048×2048 matrix due to algorithm constraint. For *6P*, *6P-ir*, *25P,* and *25P-ir*, the input data is an $8096 \times 64 \times 64$ 3D matrix. On the Intel server, for *Heat*, *Heat-ir*, *SOR*, *SOR-ir*, *9P,* and *9P-ir,* the input data used is an 8096×4096 matrix. For *GE* and *GE-ir*, the input data used is an 8192×8192 matrix due to algorithm constraint. For *6P*, *6P-ir*, *25P,* and *25P-ir*, the input data is an $8096 \times 128 \times 128$ 3D matrix.

As we can see from Figure 7, on the AMD server, LAWS can significantly improve the performance of benchmarks compared with Cilk, and the performance improvement ranges from 23.5% to 54.2%. CATS can also improve the performance of benchmarks up to 19.6% compared with Cilk. On the Intel server, LAWS can also significantly improve the performance of benchmarks compared with Cilk, and the performance improvement ranges from 12.5% to 48.6%. CATS can also improve the performance of benchmarks up to 28.1% compared with Cilk.

In MSMC architectures, the performance of a memory-bound application is decided by the *straggler socket* that seldom accesses data from its local memory node because the tasks in the straggler socket need the longest time to access their data. During the execution of a memory-bound application, any socket in the MSMC architecture can be the straggler socket.

To explain why LAWS outperforms both Cilk and CATS for memory-bound applications on both the AMD server and Intel server, we also collect the shared cache misses (Event "LLC_MISSES") and the local memory accesses (Event "MEM_UNCORE_RETIRED:LOCAL_DRAM") of the straggler socket using the

Table IV. Shared Cache Misses and Local Memory Accesses of the Straggler Socket

				Heat	SOR	GE	6P	9P	25P
Regular Benches	AMD Server	L3 Cache Misses	Cilk	5.72E8	1.15E9	2.20E8	2.52E9	5.73E8	2.48E9
			CATS	5.31E8	1.07E9	1.47E8	2.42E9	5.39E8	2.38E9
			LAWS	4.62E8	1.01E9	2.91E7	2.38E9	5.05E8	2.34E9
		Local Memory Accesses	Cilk	1.61E7	3.28E7	6.1E6	8.15E7	1.72E7	8.32E7
			CATS	2.13E7	4.14E7	4.5E6	1.01E8	2.19E7	9.06E7
			LAWS	2.58E7	5.71E7	6.5E5	1.519E8	2.72E7	1.25E8
	Intel Server	L3 Cache Misses	Cilk	1.19E9	2.39E9	7.82E8	3.48E9	9.41E8	2.31E9
			CATS	1.1E9	2.17E9	7.68E8	3.11E9	9.27E9	2.24E9
			LAWS	9.96E8	2.01E9	4.96E8	3.07E9	9.23E8	2.22E9
		Local Memory Accesses	Cilk	9062	79239	112344	110218	7576	269769
			CATS	17105	72522	133341	106469	4885	201611
			LAWS	27563	99510	145126	131165	27643	373682
				Heat-ir	SOR-ir	GE-ir	6P-ir	9P-ir	25P-ir
Regular Benches	AMD Server	L3 Cache Misses	Cilk	5.74E8	1.01E9	2.30E8	2.54E9	5.77E8	2.48E9
			CATS	5.42E8	8.86E8	1.13E8	2.36E9	4.69E8	2.37E9
			LAWS	5.05E8	8.76E8	2.87E7	2.34E9	4.46E8	2.35E9
		Local Memory Accesses	Cilk	1.72E7	2.9E7	5.64E6	7.44E7	1.53E7	8.15E7
			CATS	1.86E7	3.04E7	3.58E6	9.73E7	1.93E7	8.58E7
			LAWS	2.75E7	3.93E7	4.7E5	1.347E8	2.48E7	1.18E8
	Intel Server	L3 Cache Misses	Cilk	1.17E9	2.43E9	8.35E8	2.85E9	9.42E8	2.35E9
			CATS	1.05E9	2.19E9	8.16E8	2.78E9	9.41E9	2.27E9
			LAWS	9.9E8	1.97E9	4.97E8	2.38E9	9.33E8	2.23E9
		Local Memory Accesses	Cilk	8444	71723	121324	86822	4142	266848
			CATS	10309	79019	141726	92804	5250	248204
			LAWS	24569	102833	147161	125722	19406	381267

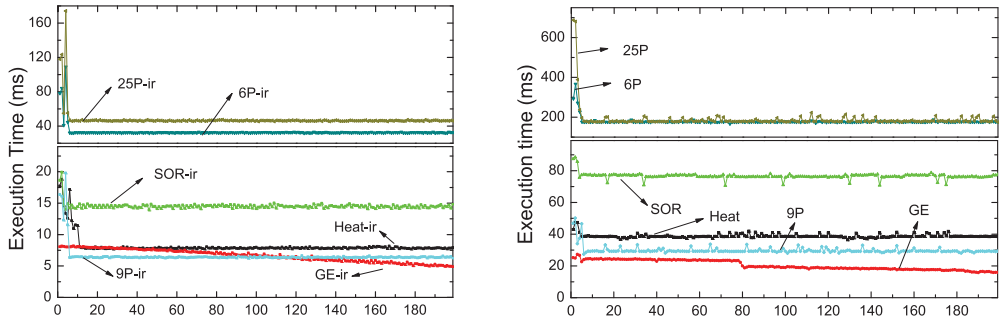
"libpfm" library in Linux. For each benchmark, Table IV lists its shared cache misses and the local memory accesses of the straggler socket in Cilk, CATS, and LAWS.

As can be observed from Table IV, we can find that the shared cache (L3) misses are reduced and the local memory accesses of the straggler socket are prominently increased in LAWS compared with Cilk and CATS. Since LAWS schedules tasks to the sockets where the local memory nodes store their data, the tasks can access their data from the local memory node and thus the local memory accesses have been significantly increased. Furthermore, since LAWS packs tasks allocated to each socket into CF subtrees to preserve shared data in the shared cache, the shared cache misses are also reduced.

Only for *GE* and *GE-ir* on the AMD server are the local memory accesses of the straggler socket not increased in LAWS. This is because their input data is small enough to be put into the shared cache directly. In this situation, most tasks can access the data from the shared cache directly and do not need to access the main memory anymore. Because the L3 cache misses are prominently reduced, LAWS can still significantly improve the performance of *GE* and *GE-ir* compared to Cilk and CATS.

The performance improvement of the benchmarks in CATS is due to the reduced shared cache misses. However, since CATS cannot divide an execution DAG optimally like LAWS, it still has more shared cache misses than LAWS, as shown in Table IV.

Careful readers may find that CATS performs much worse here than in the original paper [Chen et al. 2012]. While CATS can only improve the performance of benchmarks up to 19.6% here, it can improve their performance up to 74.4% in Chen et al. [2012]. The reduction of performance improvement of CATS comes from the much larger input



(a) AMD server (b) Intel server

Fig. 8. Execution time of each iteration in all the benchmarks in LAWS on the AMD server and Intel server.

dataset used in this article. This result matches with the findings in Chen et al. [2012]. That is, with the increasing of the size of the input dataset, the percentage of shared data among tasks decreases and the effectiveness of CATS degrades in consequence.

6.3. Effectiveness of the Adaptive DAG Packer

To evaluate the effectiveness of the adaptive DAG packer in LAWS, we compare the performance of LAWS with LAWS-NC, a scheduler that only schedules each task to the socket where the memory node stores its part of data but does not further pack the tasks into CF subtrees.

From Figure 7, we find that LAWS-NC performs better than Cilk and CATS. This is because most tasks in LAWS-NC can access their data from local memory nodes. However, since tasks are not packed into CF subtrees for optimizing shared cache in LAWS-NC, LAWS-NC incurs more shared cache misses and performs worse than LAWS.

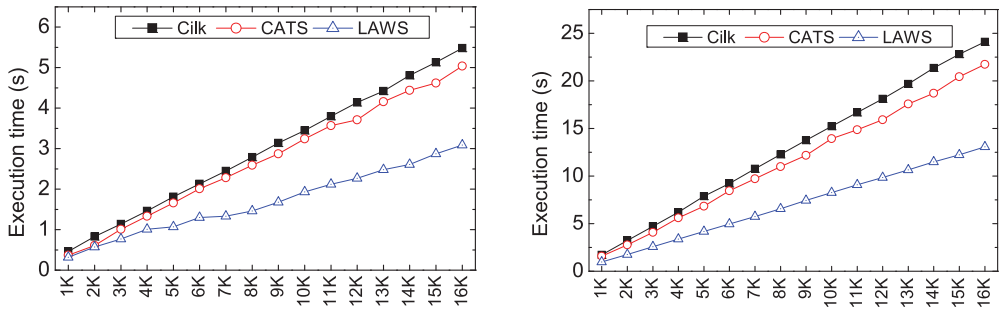
To evaluate the auto-tuning approach (Algorithm 1) proposed to optimally pack tasks into CF subtrees, Figure 8 gives the execution time of 200 iterations of all the benchmarks with irregular execution DAGs in LAWS on the AMD server and the execution time of 200 iterations of all the benchmarks with regular execution DAGs in LAWS on the Intel server. All the other benchmarks give a similar result. From the figure, we find that the execution time of an iteration in all the benchmarks is significantly reduced after the optimal packing is found in several iterations.

In summary, the adaptive DAG packer in LAWS is effective and the auto-tuning algorithm for searching the optimal packing of tasks in Algorithm 1 also works fine.

6.4. Scalability of LAWS

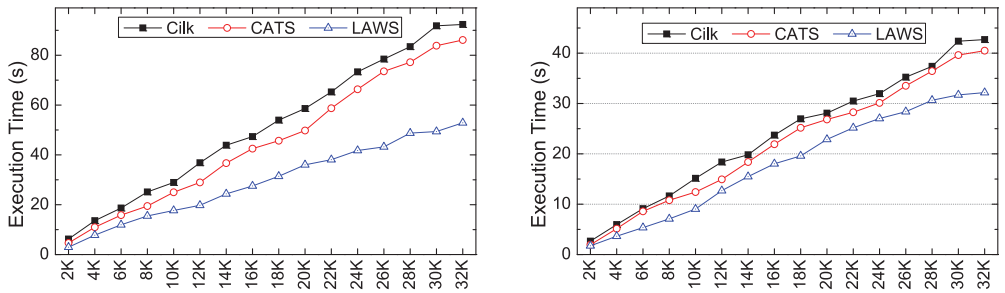
To evaluate the scalability of LAWS, we compare the performance of benchmarks with different input data sizes in Cilk, CATS, and LAWS.

During the execution of all the benchmarks, every task divides its dataset into several parts by rows to generate child tasks unless the task meets the cutoff point (i.e., the rows of a leaf task, and eight rows are used in the experiment). Since the dataset size of the leaf tasks affects the measurement of scalability, we ensure that the dataset size of the leaf tasks is constant by using a constant cutoff point for the leaf tasks. On the AMD server, if the input data is an $x \times y$ 2D matrix, we set $y = 1{,}024$ for all the input 2D matrix. If the input data is an $x \times y \times z$ 3D matrix, we set $y = 64$ and $z = 64$ for all the input 3D matrix. On the Intel server, if the input data is an $x \times y$ 2D matrix, we set $y = 4{,}096$ for all the input 2D matrix. If the input data is an $x \times y \times z$ 3D matrix, we set $y = 128$ and $z = 128$ for all the input 3D matrix.



(a) Heat-ir (b) 6P

Fig. 9. Performance of Heat-ir and 6P with different input data sizes on the AMD server.



(a) SOR-ir (b) 9P

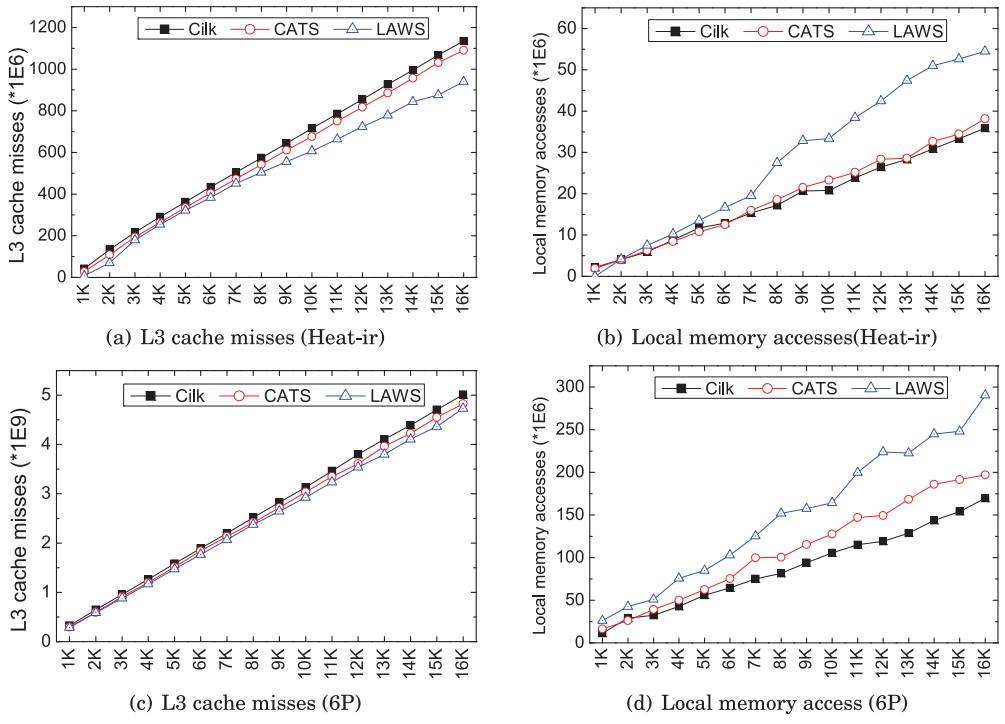
Fig. 10. Performance of SOR-ir and 9P with different input data sizes on the Intel server.

We only adjust the x of the input matrices in the experiment. In this way, we can measure the scalability of LAWS without the impact of the granularity of the leaf tasks. In all the following figures, the x-axis represents the x of the input matrixes.

We use *Heat-ir* and *6P* on the AMD server and *SOR-ir* and *9P* on the Intel server as benchmarks to evaluate the scalability of CATS in scenarios with applications with a regular execution DAG and an irregular execution DAG. All the other benchmarks have similar results. We omit them here due to the limited space.

Figure 9 and Figure 10 show the performance of benchmarks with different input data sizes in Cilk, CATS, and LAWS. We can find that *Heat-ir*, *6P*, *SOR-ir,* and *9P* achieve the best performance in LAWS for all input data sizes. When the input data size is small (i.e., $x = 1k$), LAWS reduces 30.4% execution time of *Heat-ir* and reduces 36.6% execution time of *6P* compared with Cilk on the AMD server. When the input data size is large (i.e., $x = 16k$), LAWS reduces 43.6% execution time of *Heat-ir* and reduces 45.8% execution time of *6P* compared with Cilk on the AMD server. When the input data size is small (i.e., $x = 2k$), LAWS reduces 52.5% execution time of *SOR-ir* and reduces 34.9% execution time of *9P* compared with Cilk on the Intel server. When the input data size is large (i.e., $x = 32k$), LAWS reduces 42.7% execution time of *SOR-ir* and reduces 24.7% execution time of *9P* compared with Cilk on the Intel server.

In Figure 9 and Figure 10, the execution time of benchmarks in Cilk, CATS, and LAWS increases linearly with the increasing of their input data sizes. Since their execution time increases much slower in LAWS than in Cilk and CATS, for all the input data sizes, LAWS can always reduce the execution time of memory-bound applications. In summary, LAWS is scalable in scheduling both regular execution DAGs and irregular execution DAGs.



(a) L3 cache misses (Heat-ir)

(b) Local memory accesses(Heat-ir)

(c) L3 cache misses (6P)

(d) Local memory access (6P)

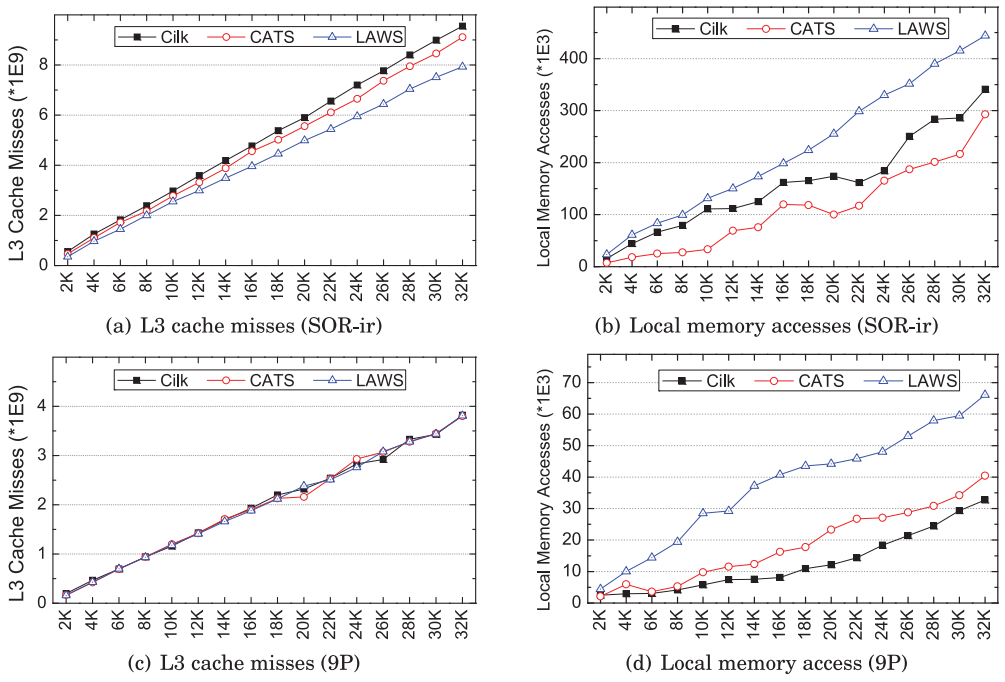
Fig. 11. L3 cache misses and local memory accesses of the straggler socket in Heat-ir and 6P on AMD server.

Corresponding to Figure 9 and Figure 10, Figure 11 and Figure 12 show the L3 cache misses and the local memory accesses of the straggler socket in executing *Heat-ir*, *6P*, *SOR-ir,* and *9P* with different input data sizes. Observed from the figure, we can find that the shared cache misses are reduced, while the local memory accesses of the straggler socket are increased in LAWS. When the input data size is small (i.e., $x = 1k$), LAWS can reduce 82% L3 cache misses and increase 132.1% local memory accesses compared with Cilk. When the input data size is large (i.e., $x = 16k$), LAWS can reduce 17.3% L3 cache misses and increase 70.6% local memory accesses compared with Cilk.

Figure 11 and Figure 12 further explain why LAWS performs much better than CATS. Since LAWS can optimally pack tasks into CF subtrees through auto-tuning, it can reduce more L3 cache misses of memory-bound benchmarks than CATS. In addition, since LAWS can schedule a task to the socket where the local memory node stores its data, it significantly increases local memory accesses. The two key advantages of LAWS result in the better performance of LAWS.

Careful readers may observe from Figure 12 that LAWS failed to reduce the last-level shared cache misses for *9P* on the Intel server. However, because LAWS significantly improved the local memory access, *9P* still performs much better than Cilk and CATS.

As we all know, if the input data of a memory-bound program is small, the shared cache is big enough to store the input data. In this case, if the shared cache misses are greatly reduced, the performance of memory-bound programs can be improved. If the input data is large, the performance bottleneck of the program is the time of reading data from main memory. Therefore, CATS performs efficiently when the input data size is small but performs poorly when the input data size is large. On the contrary, because LAWS can increase more local memory accesses when the input data size gets



(a) L3 cache misses (SOR-ir)

(b) Local memory accesses (SOR-ir)

(c) L3 cache misses (9P)

(d) Local memory access (9P)

Fig. 12. L3 cache misses and local memory accesses of the straggler socket in SOR-ir and 9P on the Intel server.

larger, it performs even better when the input data is large. This feature of LAWS is promising as the data size of a problem is becoming larger and larger.

6.5. Overhead of LAWS

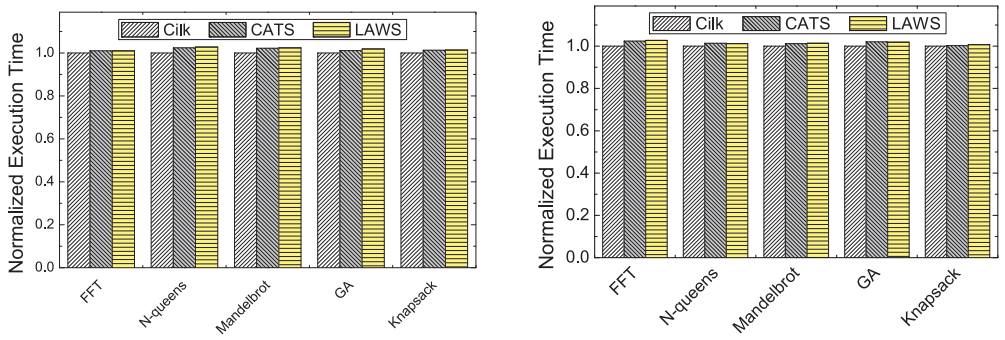
Because LAWS aims to increase local memory accesses and reduce shared cache misses, LAWS is neutral for CPU-bound programs. Based on the runtime information, if LAWS finds that a program is CPU bound, LAWS schedules tasks of the program in traditional work stealing. Another option is to use techniques in the WATS [Chen et al. 2012; Chen and Guo 2014] scheduler to improve the performance of CPU-bound programs by balancing workloads among cores.

Figure 13 shows the performance of several CPU-bound applications in Cilk, CATS, and LAWS on the AMD server and Intel server. The applications in this experiment are examples in the Cilk package. By comparing the performance of CPU-bound applications in Cilk, CATS, and LAWS, we can find the extra overhead of LAWS.

As can be observed from Figure 13, we see that the extra overhead of LAWS is negligible (less than 3% of the overall execution time) compared with Cilk and CATS. The extra overhead of LAWS mainly comes from the overhead of distributing data to all the memory nodes evenly and from the profiling overhead in the first iteration of a parallel program, when LAWS can determine if the program is CPU bound or memory bound based on the profiling information.

6.6. Discussion

LAWS assumes that the execution DAGs of different iterations in an iterative program are the same. The assumption is true for most programs. Even if a program does not satisfy this assumption, LAWS can still ensure that every task can access its data



(a) AMD server (b) Intel server

Fig. 13. Performance of CPU-bound benchmarks in Cilk, CATS, and LAWS on the AMD server and Intel server.

from the local memory node since the load-balanced task allocator allocates tasks to sockets in each iteration independently according to their dataset in the current iteration. However, in this situation, the optimization on shared cache utilization is not applicable since the optimal packing for the past iterations may not be optimal for future iterations due to the change of the execution DAG. In summary, even if this assumption is not satisfied, LAWS can improve the performance of memory-bound programs due to the increased local memory accesses.

As LAWS is neutral for CPU-bound programs, LAWS decides at runtime if an application is CPU bound based on profiled information. When LAWS collects cache misses in the first iteration, it also collects the the number of retired instructions of the task through a performance-monitoring counter. If the *cache miss intensity* (i.e., cache misses per instruction) of a task is smaller than a given threshold, the task is labeled as CPU bound. If most tasks of an application are CPU bound, the application is regarded as CPU bound by LAWS.

7. RELATED WORK

Many works have been done to improve the performance [Guo et al. 2010; Chen et al. 2012] and energy efficiency [Sridharan et al. 2013] of work stealing on various hardwares. Related to LAWS, there are two main approaches for improving the performance of memory-intensive programs in MSMC architectures: *increasing local memory accesses* and *reducing shared cache misses*.

Many works have been done to improve the performance of a particular application [Shaheen and Strzodka 2012; Yang et al. 2011; Castro et al. 2009] or general applications [Vikranth et al. 2013; Pilla et al. 2011; Muddukrishna et al. 2013] by increasing local memory accesses in the NUMA memory system (i.e., the first approach). nuCATS and nuCORALS [Shaheen and Strzodka 2012] improved the performance of iterative stencil computations for the NUMA memory system by optimizing temporal blocking and tiling. While nuCATS and nuCORALS focused on the tiling scheme for stencil programs, through online scheduling, LAWS can improve the performance of iterative stencil programs without changing the tiling scheme. A dynamic work-stealing strategy [Vikranth et al. 2013] is proposed for on-chip NUMA multicore processors based on the topology of underlying hardware. NUMALB [Pilla et al. 2011], a NUMA-aware load balancer, is proposed to improve parallel system performance based on Charm++ [Kale and Krishnan 1993]. NUMALB balances the workload while avoiding unnecessary migrations and reducing across-core communication. While these schedulers only

increase local memory accesses, LAWS can further reduce the shared cache misses and performs better for memory-intensive programs.

With the second approach, several work-stealing schedulers [Acar et al. 2002; Guo et al. 2010; Quintin and Wagner 2010; Gautier et al. 2013a] have been proposed to tackle the cache-unfriendly problem in various parallel architectures (e.g., multi-CPU and multi-GPU architectures [Gautier et al. 2013a]). From a theoretical perspective, a theoretical bound on the number of cache misses for random work stealing was presented and a locality-guided work-stealing algorithm was implemented on a single-socket SMP [Acar et al. 2002]. The effects of false sharing in algorithms using traditional work stealing are also analyzed [Cole and Ramachandran 2013]. In CONTROLLED-PDF [Blelloch et al. 2008], which is proposed for single-socket architectures, the DAG of a program is divided into *L2-supernodes* that are similar to CF subtrees in LAWS. By executing L2-supernodes sequentially, the cache misses can be reduced. However, the *L2-supernodes* are determined based on the space complexity function provided by users and cannot be adjusted at runtime even if the DAG is not divided appropriately. Furthermore, the scheduler does not consider the underlaying NUMA memory system at all.

In SLAW [Guo et al. 2010], workers are grouped into *places* and a worker is only allowed to steal tasks from other workers in the same place. The scheduling policy is similar to the triple-level work-stealing policy in LAWS. However, SLAW only considered the stealing policy and did not consider the NUMA memory systems and did not pack tasks for optimizing shared cache usage as LAWS does. Similar to LAWS, HWS [Quintin and Wagner 2010] and CAB [Chen et al. 2011] used a rigid boundary level to divide tasks into global tasks and local tasks (similar to socket-local tasks in LAWS). By scheduling local tasks within the same socket, the shared cache misses can be reduced. Users have to give the level manually in HWS or provide a number of command line arguments for the scheduler to calculate the boundary level in CAB. To relieve this burden, CATS [Chen et al. 2012] was proposed to divide an execution DAG based on the information collected online, without extra user-provided information. While the adaptive DAG packer in LAWS can find the optimal packing of tasks into CF subtrees through auto-tuning, all the previously mentioned schedulers cannot optimally partition an execution DAG. In addition, they did not consider the NUMA memory system. Our experiment results also show that LAWS significantly outperforms CATS.

Based on METIS [Karypis and Kumar 1998], an offline graph-based locality analysis framework [Yoo et al. 2013] is proposed to analyze the inherent locality patterns of workloads. Leveraging the analysis results, tasks are grouped and mapped according to cache hierarchy through recursive scheduling. Because the framework relied on offline analysis, a program has to be executed at least one time before it can achieve good performance in the framework. On the contrary, LAWS can improve the performance of programs online without any prerequisite offline analysis, because it can pack tasks into CF subtrees based on online-collected information and auto-tuning.

8. CONCLUSIONS

Traditional work-stealing schedulers suffer from shared cache pollution and the small number of local memory accesses in MSMC architectures with the NUMA-based memory system. To solve these two problems, we have proposed the LAWS scheduler, which consists of a load-balanced task allocator, an adaptive DAG packer, and a triple-level work-stealing scheduler. The load-balanced task allocator evenly distributes the dataset of a program to all the memory nodes and allocates a task to the socket where the local memory node stores its data for increasing local memory accesses. Based on auto-tuning, for each socket, the adaptive DAG packer can optimally pack the allocated tasks into CF subtrees to optimize shared cache usage. The

triple-level work-stealing scheduler schedules tasks in the same CF subtree among cores in the same socket and makes sure that each socket executes its CF subtrees sequentially. Experimental results show that LAWS can improve the performance of memory-bound programs up to 54.2% on the AMD-based experimental platform and up to 48.6% on the Intel-based experimental platform compared with traditional work-stealing schedulers. Furthermore, the extra overhead of LAWS for CPU-intensive applications is negligible.

REFERENCES

U. A. Acar, G. E. Blelloch, and R. D. Blumofe. 2002. The data locality of work stealing. *Theory of Computing Systems* 35, 3 (2002), 321–347.

E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. 2009. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (2009), 404–418.

G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. 2008. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, 501–510.

R. D. Blumofe. 1995. *Executing Multithreaded Programs Efficiently*. Ph.D. Dissertation. MIT.

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing* 37, 1 (1996), 55–69.

M. Castro, L. G. Fernandes, C. Pousa, J.-F. Méhaut, and M. S. de Aguiar. 2009. NUMA-ICTM: A parallel version of ICTM exploiting memory placement strategies for NUMA machines. In *The 23rd International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

Q. Chen, Y. Chen, Z. Huang, and M. Guo. 2012. WATS: Workload-aware task scheduling in asymmetric multi-core architectures. In *The 26th International Parallel and Distributed Processing Symposium*. IEEE, 249–260.

Q. Chen and M. Guo. 2014. Adaptive workload aware task scheduling for single-ISA multi-core architectures. *ACM Transactions on Architecture and Code Optimization* 11, 1 (2014), 8.

Q. Chen, M. Guo, and H. Guan. 2014. LAWS: Locality-aware work-stealing for multi-socket multi-core architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing*. ACM, 3–12.

Q. Chen, M. Guo, and Z. Huang. 2012. CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th Annual International Conference on Supercomputing*. ACM, 163–172.

Q. Chen, M. Guo, and Z. Huang. 2013. Adaptive cache aware bi-tier work-stealing in multi-socket multi-core architectures. *IEEE Transactions on Parallel and Distributed Systems* 24, 12 (2013), 2334–2343.

Q. Chen, Z. Huang, M. Guo, and J. Zhou. 2011. CAB: Cache-aware bi-tier task-stealing in multi-socket multi-core architecture. In *The 40th International Conference on Parallel Processing*. IEEE, 722–732.

R. Cole and V. Ramachandran. 2013. Analysis of randomized work stealing with false sharing. In *The 27th International Parallel and Distributed Processing Symposium*. IEEE, 985–989.

Hypertransport Technology Consortium. 2010. HyperTransport I/O Link Specification, Revision 3.10c edition. (2010).

M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 212–223.

T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. 2013a. Locality-aware work stealing on multi-CPU and multi-GPU architectures. In *The 6th Workshop on Programmability Issues for Heterogeneous Multicores*.

T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. 2013b. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *The 27th International Parallel and Distributed Processing Symposium*. IEEE, 1299–1308.

A. Gerasoulis and T. Yang. 1992. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing* 16, 4 (1992), 276–291.

Y. Guo, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *The 23th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–12.

Y. Guo, J. Zhao, V. Cave, and V. Sarkar. 2010. SLAW: A scalable locality-aware adaptive work–stealing scheduler. In *The 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–12.

Intel. 2009. Introduction to the Intel Quickpath Interconnect. *White Paper* (2009).

L. V. Kale and S. Krishnan. 1993. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 91–108.

G. Karypis and V. Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

J. K. Lee and J. Palsberg. 2010. Featherweight X10: A core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 25–36.

C. E. Leiserson. 2009. The Cilk++ concurrency platform. In *The 46th ACM/IEEE Design Automation Conference*. ACM, New York, 522–527.

A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson. 2013. Locality-aware task scheduling and data distribution on NUMA systems. In *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 156–170.

L. L. Pilla, C. P. Ribeiro, D. Cordeiro, A. Bhatele, P. O. A. Navaux, J.-F. Méhaut, and L. V. Kalé. 2011. Improving parallel system performance with a NUMA-aware load balancer. *TR-JLPC-11-02* (2011).

J.-N. Quintin and F. Wagner. 2010. Hierarchical work-stealing. In *The 16th International Euro-Par Conference*. Springer, 217–229.

J. Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media.

M. Shaheen and R. Strzodka. 2012. NUMA aware iterative stencil computations on many-core systems. In *The 26th International Parallel and Distributed Processing Symposium*. IEEE, 461–473.

S. Sridharan, G. Gupta, and G. S. Sohi. 2013. Holistic run-time parallelism management for time and energy efficiency. In *Proceedings of the 27th Annual International Conference on Supercomputing*. ACM, 337–348.

B. Vikranth, R. Wankar, and C. R. Rao. 2013. Topology aware task stealing for on-chip NUMA multi-core processors. *Procedia Computer Science* 18 (2013), 379–388.

R. Yang, J. Antony, A. Rendell, D. Robson, and P. Strazdins. 2011. Profiling directed NUMA optimization on Linux systems: A case study of the Gaussian computational chemistry code. In *The 25th International Parallel and Distributed Processing Symposium*. IEEE, 1046–1057.

R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis. 2013. Locality-aware task management for unstructured parallelism: a quantitative limit study. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 315–325.

Received March 2015; accepted April 2015