LAWS: Locality-Aware Work-Stealing for Multi-socket Multi-core Architectures

Quan Chen
Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University
chen-quan@sjtu.edu.cn

Minyi Guo*
Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University
guo-my@cs.sjtu.edu.cn

Haibing Guan
Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University
hbguan@sjtu.edu.cn

ABSTRACT

Modern mainstream powerful computers adopt Multi-Socket Multi-Core (MSMC) 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, which can degrade the performance of memory-bound applications seriously. To solve the problem, we propose a Locality-Aware Work-Stealing (LAWS) scheduler, which better utilizes both the shared cache and the NUMA memory system. In LAWS, a load-balanced task allocator is used to evenly split and store the data set of a program to all the memory nodes and allocate a task to the socket where the local memory node stores its data. Then, an adaptive DAG packer adopts an auto-tuning approach to optimally pack an execution DAG into many cache-friendly subtrees. Meanwhile, a triple-level work-stealing scheduler is applied to schedule the subtrees and the tasks in each subtree. Experimental results show that LAWS can improve the performance of memory-bound programs up to 54.2% compared with traditional work-stealing schedulers.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Run-tume environments

Keywords

Shared cache; NUMA; Auto-tuning; DAG packing

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 Multi-Socket Multi-Core (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* [3] and *work-stealing* [4] are two best-known dynamic load balancing policies. For instance, TBB [25], XKaapi [14], Cilk++ [22] and X10 [21] use work-stealing, OpenMP [3] 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 [16]). 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 extreme-scale supercomputing systems often employ NUMA-based (*Non-Uniform Memory Access*) memory system, 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 very inefficient in this architecture.

In 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 (the problem will be discussed in detail in Section 2).

To reduce both remote memory accesses and shared cache

*Minyi Guo is the correspondence author of this paper.

ICS'14, June 10–13, 2014, Munich, Germany.

http://dx.doi.org/10.1145/2597652.2597665.

misses, this paper 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 cache friendly manner. LAWS targets iterative divide-and-conquer applications that have tree-shaped execution DAG. 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 paper 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.
- 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.

The rest of this paper is organized as follows. Section 2 explains the motivation of LAWS. Section 3 presents locality-aware work-stealing, including balanced data allocator, adaptive DAG packer and triple level work-stealing scheduler. Section 4 gives the implementation of LAWS. Section 5 evaluates LAWS and shows the experimental results. Section 6 discusses the related work. Section 7 draws conclusions.

2. MOTIVATION

Similar to many popular work-stealing schedulers (e.g., Cilk [5] and CATS [10]), this paper targets iterative *Divide-and-Conquer* (D&C) programs that have tree-shaped execution DAG. Most stencil programs [26] and algorithms based on jacobi iteration (e.g., *Heat distribution* and *Successive Over Relaxation*) are examples of iterative D&C programs.

Fig. 1 gives a general execution DAG for iterative D&C programs. In a D&C program, its data set is recursively divided into several parts until each of the leaf tasks only processes a small part of the whole data set.

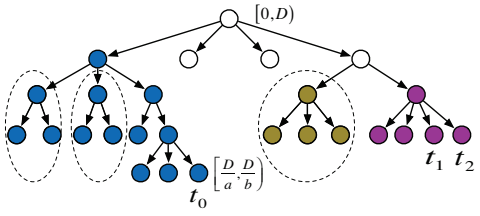


Figure 1: A general execution DAG for iterative D&C programs.

Suppose the execution DAG in Fig. 1 runs on an MSMC architecture with a NUMA memory system as shown in Fig. 2. 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 *first touch strategy*. In this work, we take advantage of this strategy of memory allocation.

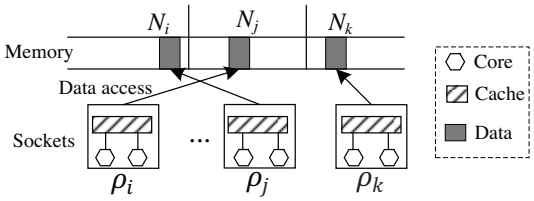


Figure 2: The data access pattern in traditional work-stealing.

For a parallel program, its data set is often first accessed by tasks in the first iteration or in an indepedent initialization phase. By scheduling these tasks to different sockets, the whole data set of the program that has the execution DAG in Fig. 1 is split and stored in different memory nodes as shown in Fig. 2 due to the first touch strategy.

However, traditional work-stealing suffers from two main problems when scheduling the execution DAG in Fig. 1 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 data set of the program in Fig. 1 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., task t_1 and task t_2 in Fig. 1) 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 Fig. 1, 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 *Locality-Aware Work-Stealing* (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 data set 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 *Cache-Friendly Subtrees* (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 verify the effectiveness of LAWS through theoretical analysis.

3.1 Design of LAWS

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

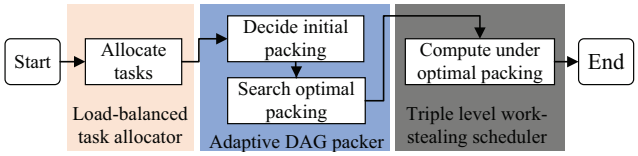


Figure 3: The processing flow of an iterative D&C program in LAWS.

In every iteration, the task allocator carefully allocates tasks to different sockets to evenly distribute the data set 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 Fig. 1, 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 data set 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 victim 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 and cache misses can be reduced.

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 data set 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 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 schedule 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 data set is D, to balance workload, the tasks allocated to each socket need to process $\frac{1}{M}$ of the whole data set. Without loss of generality, LAWS makes sure that the tasks allocated to the i-th ($1 \le i \le M$) socket should process the part of the whole data set 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 the above objective, we need to find out each task processes which part of the whole data set. For a task α_2 in Fig. 4, to find out it will process which part of the data set, LAWS analyzes the structure of the dynamically generated execution DAG when α_2 is spawned. Suppose task α_2 is task α_1's i-th sub-task and the branching degree of α_1 is b. If α_1 processes the part of data $[D_s, D_e)$, Eq. 1 gives the part of data that α_2 will process.

$$[(i-1) \times \frac{D_e - D_s}{b} + D_s, i \times \frac{D_e - D_s}{b} + D_s) \qquad (1)$$

Fig. 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 Eq. 1.

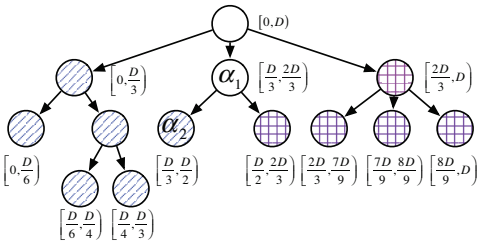


Figure 4: Allocate the tasks to the two sockets of a dual-socket architecture.

In the dual-socket architecture, the tasks that process the data set $[0, \frac{D}{2})$ and $[\frac{D}{2}, D)$ should be allocated to the first socket and the second socket respectively. For instance, in Fig. 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. Due to the same reason, the slash-shaded tasks should be allocated to the first socket and the mesh-shaded tasks should be allocated to the second socket.

Note that, 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 load-balanced task allocator allocates a task according to the range of its data set, in the following iterations, the tasks processing the same part of the whole data set 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 are 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.

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 into the shared cache of a socket. Note 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 above objective, we need to know the size of 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 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 due to 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. SOSC of α is calculated in the bottom-up manner. Suppose α has m direct child tasks α_1, ..., α_m and their SOSCs are S_1, ..., S_m respectively. SOSC of α (denoted by S_α) can be calculated in Eq. 2, where M_α equals to 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 Eq. 2 precisely equals to 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 due to 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 Eq. 2. On the other hand, if some data stored in the shared cache has already been pre-fetched into the private cache before, it does not incur last level private cache misses and the size of the pre-fetched data is missed in Eq. 2. The multiple accumulation of shared data and the pre-fetching make S_α of Eq. 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 the shortest makespan as the optimal packing.

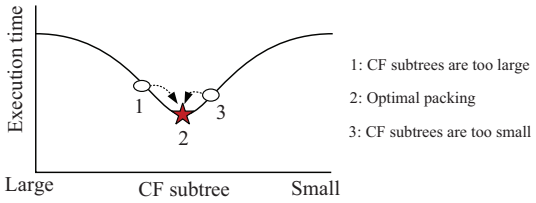


Figure 5: Execution time of an iteration when the execution DAG is packed differently.

Fig. 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 Fig. 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 Fig. 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 Fig. 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 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.

Since the initial packing is already near-optimal, LAWS can find the optimal packing in a few iterations. Theoretically, it has 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 subtrees independently in a small number of iterations at run-

Algorithm 1: Algorithm for searching the optimal way to pack the 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 Fig. 5
8          T_c = T_n ;
9          Save new CF root tasks ;
10         EvalLarger = 0 ;
11     else break ;
12 if EvalLarger == 1 then // Point 3 in Fig. 5
13     Restore CF root tasks to {α_1, ..., α_m} ;
14     T_c = T ;
15     while CF root tasks have parent tasks do
16         Set parent tasks of the current CF root tasks as
           the new CF root tasks ;
17         Execute an iteration under the new packing ;
18         Record the execution time T_n ;
19         if T_n > T_c then break ;
20         else T_c = T_n ; Save new CF root tasks ;
```

time. To simplify the problem, we increase or decrease the size of all the CF subtrees at the same time in Algorithm 1 of this paper. Actually, according to the experiment in Section 5.2, our current auto-tuning strategy in Algorithm 1 works efficiently.

The approach of packing DAG into CF subtrees in LAWS partially origins from CATS [9], which also packs the execution DAGs of parallel programs into subtrees for optimizing shared cache usage. However, once an execution DAG is packed in CATS, the packing cannot be adjusted even if the packing is not optimal. Experiment in Section 5.2 shows that the performance of applications can be further improved with the auto-tuning algorithm described in Algorithm 1 for searching the optimal packing. 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 5.

3.4 Triple-level work-stealing scheduler

Fig. 6 gives the architecture of LAWS on an M-socket multi-core architecture, and illustrates the triple-level work-stealing policies in LAWS. In Fig. 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 CF task pool of ρ. Core c is not allowed to steal a task from other sockets because the data set of a task will be stored into the wrong memory node if it is stolen in the first iteration due to the first touch strategy.

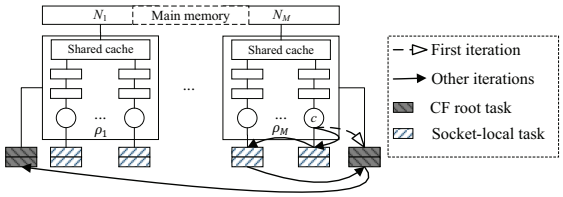


Figure 6: Architecture of LAWS on an M-socket multi-core architecture.

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 CF task pools of other sockets. Although ρ needs 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 data sets. Also, a socket is only allowed to steal entire CF subtrees from other sockets for optimizing shared cache usage.

3.5 Theoretical Validation

A memory-bound D&C program has three features. First, only leaf tasks physically access the data while other tasks divide the data set recursively into smaller pieces. Second, each leaf task only processes a small part of the whole data set of the program. Third, the execution time of a leaf task is decided by its data access time. 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 node 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 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 Eq. 3.

$$T_R = \frac{S+B}{V_l} \times \frac{1}{M} + \frac{S+B}{V_r} \times \frac{M-1}{M} \quad (3)$$

If we adopt LAWS to schedule the program, benefit from the task allocator, α can access its own part of data from 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 Eq. 4, because α also has a high chance to access its boundary data from local memory node.

$$T_L \leq \frac{S}{V_l} + \frac{B}{V_r} \quad (4)$$

Deduced from Eq. 3 and Eq. 4, we can get Eq. 5.

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

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

Because leaf tasks need 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.

4. IMPLEMENTATION

We implement LAWS by modifying MIT Cilk, which is one of the earliest parallel programming environments that implement work-stealing [13].

Existing work-stealing schedulers adopt either parent-first policy or child-first policy when generating new tasks. In parent-first policy (called *help-first policy* in [17]), a core continually executes the parent task after spawning a new task. In child-first policy (called *work-first policy* in [5]), a core continually executes the spawned new task once the child is spawned. Parent-first policy works better when the steals are frequent, while child-first policy works better when the steals are infrequent [17].

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 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 compiler of MIT Cilk to support both the parent-first and child-first task-generating policy while the original Cilk only support 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. EVALUATION

We use a server that has four AMD Quad-core Opteron 8380 processors to evaluate the performance of LAWS. Each socket has a 512K private L2 cache for each core and a 6M L3 cache shared by all four cores. The server has 16GB RAM and runs Linux 3.2.0-14. Therefore, each socket has a 4GB memory node.

We compare the performance of LAWS with the performance of Cilk [5] and CATS [9]. 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 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.

Table 1: Benchmarks used in the experiments

Name	Description
Heat/Heat-ir	2D heat distribution (regular/irreg.)
SOR/SOR-ir	Successive Over-Relaxation (regular/irreg.)
GE/GE-ir	Gaussian elimination alg. (regular/irreg.)
9p/9p-ir	2D 9-point stencil comp. (regular/irreg.)
6p/6p-ir	3D 6-point stencil comp. (regular/irreg.)
25p/25p-ir	3D 25-point stencil comp. (regular/irreg.)

In order to evaluate the performance of LAWS in different scenarios, we use benchmarks listed in Table 1 that have both regular execution DAG and irregular execution DAG in the experiment. 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. *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 [9]. If all the nodes (except the leaf tasks) in the execution DAG have the same branching degrees, the execution DAG is regular. All benchmarks are compiled with "-O2".

5.1 Performance of LAWS

Fig. 7 shows the performance of all the benchmarks in Cilk, CATS and LAWS.

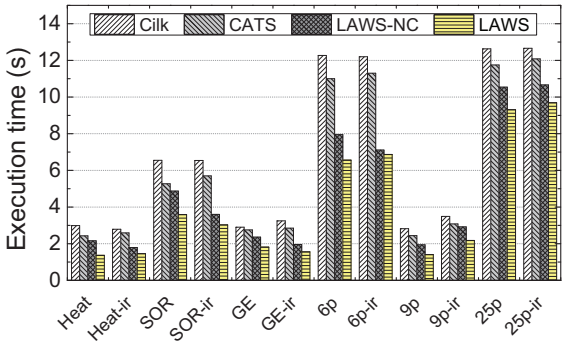


Figure 7: The performance of all the benchmarks in Cilk, CATS and LAWS.

In this experiment, for *Heat*, *Heat-ir*, *SOR* and *SOR-ir*, the input data used is a 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 a $8096 \times 64 \times 64$ 3D matrix.

As we can see from Fig. 7, LAWS can significantly improve the performance of benchmarks compared with Cilk while 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.

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

To explain why LAWS outperforms both Cilk and CATS for memory-bound applications, we collect the shared cache misses (Event 4E1H) and the local memory accesses (Event 1E0H) of the straggler socket. The information about the events of hardware performance units can be found in [2]. For each benchmark, Table 2 lists its shared cache misses and the local memory accesses of the straggler socket in Cilk, CATS and LAWS.

Observed from Table 2, 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 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 shared cache, the shared cache misses are also reduced.

Only for *GE* and *GE-ir*, the local memory accesses of the straggler socket are 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 any more. Because the L3 cache misses are prominently reduced, LAWS can still significantly improve the performance of *GE* and *GE-ir*.

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 2.

Surprisingly, CATS can also slightly increase the local memory accesses of the straggler socket. As mentioned before, we have improved CATS so that the adjacent data is stored in the same memory node. Although tasks have the same possibility ($\frac{1}{M}$ on an M-socket architecture) to find its data in the local memory node in Cilk and CATS, if a task can find its own data in the local memory node in CATS, it has higher possibility to also find its boundary data in the local memory node. The local memory accesses of the straggler socket in CATS are increased in consequence.

Careful readers may find that CATS performs much worse here than in the original paper [9]. While CATS can only improve the performance of benchmarks up to 19.6% here, it can improve their performance up to 74.4% in [9]. The reduction of performance improvement of CATS comes from the much larger input data set used in this paper. This result matches with the findings in [9]. That is, with the increasing of the size of the input data set, the percentage of shared data among tasks decreases and the effectiveness of CATS degrades in consequence.

5.2 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 Fig. 7 we find that LAWS-NC performs better than both 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.

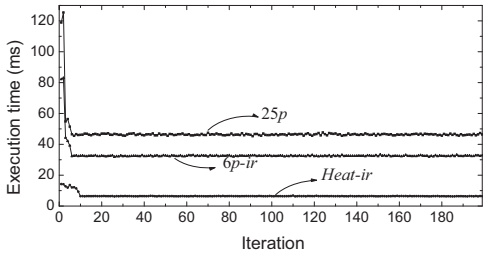


Figure 8: Execution time of each iteration in 25p, 6p-ir and Heat-ir.

To evaluate the auto-tuning approach (Algorithm 1) proposed to optimally pack tasks into CF subtrees, Fig. 8 gives the execution time of 200 iterations of *Heat-ir, 25p* and *6p-ir* in LAWS. 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 works also fine.

5.3 Scalability of LAWS

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

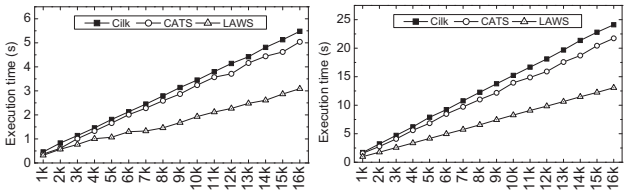
During the execution of all the benchmarks, every task divides its data set into several parts by rows to generate

Table 2: Shared cache misses and local memory accesses of the straggler socket (*1E6)

		Heat	Heat-ir	SOR	SOR-ir	GE	GE-ir	6p	6p-ir	9p	9p-ir	25p	25p-ir
L3 Cache Misses	Cilk	572	574	1151	1013	220.3	230.1	2518	2543	573.2	577	2484	2477
	CATS	531	541.8	1070	886	147.4	113.3	2420	2361	539.1	469.2	2383	2372
	LAWS	462	504.5	1005	876	29.1	28.7	2375	2345	504.5	446.03	2340	2354
Local Memory Accesses	Cilk	16.1	17.2	32.8	29	6.1	5.64	81.5	74.4	17.2	15.3	83.2	81.5
	CATS	21.3	18.6	41.4	30.4	4.5	3.58	100.5	97.3	21.9	19.3	90.6	85.8
	LAWS	25.8	27.5	57.1	39.3	0.65	0.47	151.9	134.7	27.2	24.8	125	117.7

child tasks unless the task meets the cutoff point (i.e., the rows of a leaf task, and 8 rows is used in the experiment). Since the data set size of the leaf tasks affects the measurement of scalability, we ensure that the data set size of the leaf tasks is constant by using a constant cutoff point for the leaf tasks. If the input data is an $x \times y$ 2D matrix, we set $y = 1024$ 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. 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* as benchmarks to evaluate the scalability of CATS in scenario that applications with a regular execution DAG and an irregular execution DAG. All the other benchmarks have similar results.



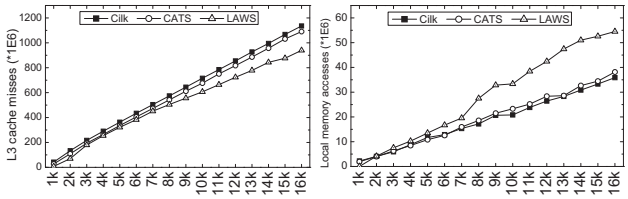
(a) Heat-ir (b) 6p

Figure 9: Performance of Heat-ir and 6p with different input data sizes.

Fig. 9 shows the performance of *Heat-ir* and *6p* with different input data sizes in Cilk, CATS and LAWS. We can find that *Heat-ir* and *6p* 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. 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.

In Fig. 9, 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.

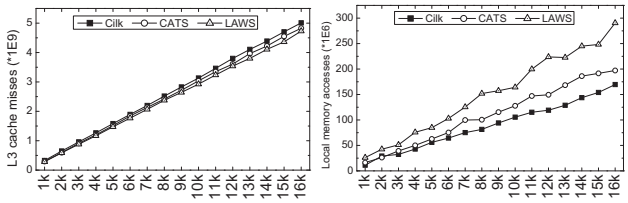
Corresponding to Fig. 9, Fig. 10 and Fig. 11 show the L3 cache misses and the local memory accesses of the straggler socket in executing *Heat-ir* and *6p* with different input data sizes. Observed from the figure, we can find that the shared cache misses are reduced, while the local memory accesses of



(a) L3 cache misses (b) Local memory accesses

Figure 10: L3 cache misses and local memory accesses of the straggler socket in Heat-ir.

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.



(a) L3 cache misses (b) Local memory access

Figure 11: L3 cache misses and local memory accesses of the straggler socket in 6p.

Fig. 10 and Fig. 11 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.

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 efficient when the input data size is small but performs poor when the input data size is large. On the contrary, because LAWS can increase more local memory accesses when input data size gets larger, it performs even better when the input data is large as shown in Fig. 10 and Fig. 11. This feature of LAWS is

promising as the data size of a problem is becoming larger and larger.

5.4 Overhead of LAWS

Because LAWS aims to reduce remote memory accesses and 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 WATS [7, 8] scheduler to improve the performance of CPU-bound programs by balancing workloads among cores.

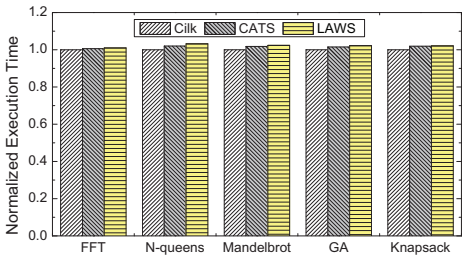


Figure 12: Performance of CPU-bound benchmarks in Cilk, CATS and LAWS.

Fig. 12 shows the performance of several CPU-bound applications in Cilk, CATS and LAWS. The applications in this experiment are examples in Cilk package. By comparing the performance of CPU-bound applications in Cilk, CATS and LAWS, we can find the extra overhead of LAWS.

Observed from Fig. 12, we see the extra overhead of LAWS is negligible compared with Cilk and CATS. The extra overhead of LAWS mainly comes from the overhead of distributing data to all the memory node evenly and 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.

5.5 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 from local memory node since the load-balanced task allocator allocates tasks to sockets in each iteration independently according to their data set 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 the above assumption is not satisfied, LAWS can improve the performance of memory-bound programs due to the increased local memory accesses.

6. RELATED WORK

Work-stealing is increasing popular due to its automatic load balancing feature and high performance. There are many works have been done to improve its performance [18, 9] and energy efficiency [27] on various hardwares. However, most existing work-stealing schedulers are designed for single-socket architectures and degrade the performance of memory-bound programs in MSMC architectures with NUMA memory system. There are two main approaches for improving the performance of memory-bound programs in MSMC architectures: *increasing local memory accesses* and *reducing shared cache misses*.

Many studies have been done to improve the performance of a particular application [26, 6] or general applications [28, 23] by increasing local memory accesses in NUMA memory system (i.e., the first approach).

In [26], nuCATS and nuCORALS improved the performance of iterative stencil computations for 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. In [28], a dynamic work-stealing strategy is proposed for on-chip NUMA multi-core processors based on the topology of underlying hardware. Based on Charm++ [19], NUMALB [23] is proposed to improved the performance of parallel programs. NUMALB balances the workload while avoiding unnecessary migrations and reducing across-core communication. While the above schedulers only increase local memory accesses, LAWS can further reduce the shared cache misses and thus performs better for memory-bound programs.

Using the second approach, there are also several work-stealing schedulers [1, 18, 24, 15] are proposed to tackle the cache-unfriendly problem in various parallel architectures (e.g., multi-CPU and multi-GPU architectures [15]). In [12], the authors analyzed the cache misses of algorithms using traditional task-stealing, focusing on the effects of false sharing. In SLAW [18], 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 did not consider the NUMA memory systems and did not pack tasks for optimizing shared cache usage as LAWS does.

Similar to LAWS, HWS [24] and CAB [11] 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. However, 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 the above burden, CATS [9] 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 above schedulers cannot optimally partition an execution DAG. In addition, they did not consider the NUMA memory system at all. Our experiment results also show that LAWS significantly outperforms CATS.

In [29], an offline graph-based locality analysis framework based on METIS [20] 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 optimally pack tasks into CF subtrees based on online collected information and auto-tuning.

7. CONCLUSIONS

Traditional work-stealing schedulers pollute shared cache and increase remote memory accesses in MSMC architectures with NUMA-based memory system. To solve the 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 task allocator evenly distributes the data set of a program to all the memory nodes and allocates a task to the socket where the local memory node stores its data. Based on auto-tuning, for each socket, the adaptive DAG packer can optimally pack the allocated tasks into CF subtrees. 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. In this way, the shared cache misses are greatly reduced and the local memory accesses are prominently increased. Experimental results demonstrate that LAWS can achieve up to 54.2% performance gain for memory-bound applications compared with traditional work-stealing schedulers.

8. ACKNOWLEDGMENTS

This work was supported by Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT) China, NSFC (Grant No. 61272099) and Scientific Innovation Act of STCSM (No.13511504200). This work was also partially supported by NRF Singapore under its Campus for CREATE Program.

9. REFERENCES

[1] U. Acar, G. Blelloch, and R. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] AMD. *BIOS and Kernel Developer Guide (BKDG) For AMD Family 10h Processors*. AMD, 2010.

[3] 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 TPDS*, 20(3):404–418, 2009.

[4] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, MIT, Sept. 1995.

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

[6] M. Castro, L. G. Fernandes, C. Pousa, J.-F. Méhaut, and M. S. de Aguiar. NUMA-ICTM: A parallel version of ICTM exploiting memory placement strategies for NUMA machines. In *IPDPS*, pages 1–8, 2009.

[7] Q. Chen, Y. Chen, Z. Huang, and M. Guo. WATS: Workload-aware task scheduling in asymmetric multi-core architectures. In *IPDPS*, pages 249–260, 2012.

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

[9] Q. Chen, M. Guo, and Z. Huang. CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *ICS*, pages 163–172, 2012.

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

[11] Q. Chen, Z. Huang, M. Guo, and J. Zhou. CAB: Cache-aware bi-tier task-stealing in multi-socket multi-core architecture. In *ICPP*, pages 722–732, 2011.

[12] R. Cole and V. Ramachandran. Analysis of randomized work stealing with false sharing. In *IPDPS*, pages 985–989, 2013.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[14] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *IPDPS*, pages 1299–1308, 2013.

[15] T. Gautier, J. V. F. Lima, N. Maillard, B. Raffin, et al. Locality-aware work stealing on Multi-CPU and Multi-GPU architectures. In *MULTIPROG*, 2013.

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

[17] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS*, pages 1–12, 2009.

[18] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a scalable locality-aware adaptive work–stealing scheduler. In *IPDPS*, pages 1–12, 2010.

[19] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*. ACM, 1993.

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

[21] J. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *PPoPP*, pages 25–36, 2010.

[22] C. Leiserson. The Cilk++ concurrency platform. In *DAC*, pages 522–527, 2009.

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

[24] J.-N. Quintin and F. Wagner. Hierarchical work-stealing. In *EuroPar*, pages 217–229, 2010.

[25] J. Reinders. *Intel threading building blocks*. Intel, 2007.

[26] M. Shaheen and R. Strzodka. NUMA aware iterative stencil computations on many-core systems. In *IPDPS*, pages 461–473, 2012.

[27] S. Sridharan, G. Gupta, and G. S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *ICS*, pages 337–348, 2013.

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

[29] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis. Locality-aware task management for unstructured parallelism: a quantitative limit study. In *SPAA*, pages 315–325, 2013.