2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems

SAWS: Selective Asymmetry-aware Work-Stealing for Asymmetric Multi-Core Architectures

Haodong Guo*, Quan Chen†, Minyi Guo‡ and Liting Xu§
Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China
Email: *sjtu.ghd@sjtu.edu.cn, { †chen-quan, ‡guo-my, §xu-lt}@cs.sjtu.edu.cn

***Abstract*—Asymmetric Multi-Core (AMC) architectures, where cores in different CPUs have different performance and power consumption, have been widely used from large-scale datacenters to mobile smart-phones for their high performance as well as energy efficiency. However, existing task scheduling policies often result in the poor performance of parallel programs on emerging AMC architectures due to the unbalanced workload, the severe shared cache misses and remote memory accesses. To solve this problem, we propose a Selective Asymmetry-aware Work-Stealing (SAWS) runtime system, which can reduce remote memory accesses while balancing workload across asymmetric cores. SAWS consists of an asymmetric-aware task allocator and a selective work-stealing scheduler. The asymmetric-aware task allocator properly distributes the tasks to asymmetric CPUs so that most tasks can access data from local memory node and the workload is balanced according to the computational ability of different CPUs. After that, the selective work-stealing scheduler is used to further balance the workload at runtime and adjust the frequencies of asymmetric cores. Our real-system experimental results show that SAWS improves the performance of memory-bound programs up to 59.3% compared with traditional work-stealing schedulers in AMC architectures.**

I. Introduction

In order to fulfill the urgent requirement for high computational capacity, emerging computer technology demands increasingly on parallel computing, such as multi-core, which integrates multiple cores in a CPU. Nowadays, multicore processors have become mainstream in both research and public settings, from datacenters to personal laptops to smartphones, since they demonstrate superior performance per watt and larger computational capacity than single-core processors.

For multi-core architectures, it is crucial to balance workloads among all the cores so that they can be utilized most effectively. Researchers have found that dynamic task scheduling in runtime systems is an effective solution. MIT Cilk [1], Cilk++ [2], TBB [3] and LAWS [4] adopt work-stealing strategy [5], while OpenMP [6] introduces work-sharing strategy [6]. Both of them are dynamic task scheduling strategies.

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. They perform good on traditional symmetric multi-core architectures.

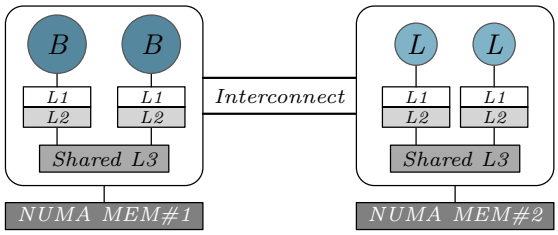


Fig. 1: An example asymmetric multi-core (AMC) architecture with NUMA-based memory system, in which each CPU has its local memory node. B = big (beefy) core; L = little (weak) core; L1 = L1 cache; L2 = L2 cache; Shared L3 = shared L3 cache; NUMA MEM = non-uniform memory access (NUMA) memory node.

However, vendors start to produce Asymmetric Multi-Core (AMC) architectures [7], [8], [9] to meet the high performance and low power consumption requirements for processing different types of workloads. In AMC architectures, such as Arm Big-Little [10] and Intel Quick-IA [11], "big" (beefy) cores can be used to execute complex computing tasks and slow "little" (weak) cores deal with simple transactions. As suggested in Quick-IA [11], in order to build an AMC architecture, it is more efficient to integrated multiple asymmetric CPUs in the same shared-memory computer, where cores in the same CPU are symmetric but cores in different CPUs are asymmetric. Figure 1 shows an example of AMC architecture. As shown in the figure, when multiple CPUs are integrated in the same computer, the memory is often organized as *Non-Uniform Memory Access* (NUMA) structure. In AMC architecture with NUMA-based memory system, each CPU, a.k.a NUMA node has its own local memory node. It is much faster for a core to access data from local memory node than from remote memory node through interconnect link, such as Intel Quick-Path Inter-connection (QPI). Traditional work-stealing is very inefficient in the AMC architectures due to the unawareness of asymmetric cores and the large amount of remote memory accesses (to be explained in Section II).

Targeting this problem, there are a large amount of prior work [12], [13], [14] aims to improve the performance of parallel programs on AMC architecture. For AMC architec-

978-1-5090-4297-5/16 $31.00 © 2016 IEEE
DOI 10.1109/HPCC-SmartCity-DSS.2016.20

tures where the frequencies of cores are fixed (called "static asymmetry"), PFT [12] and WATS [13] attempt to adopt their own spawning policies to allocate tasks appropriately. For AMC architectures where the frequencies of cores is changeable through DVFS (called "dynamic asymmetry"), AAWS [14] proposes work-pacing, work-sprinting and work-mugging techniques that dynamically adjust the frequencies of big and little cores to obtain good performance. AAWS considers the computational ability of cores when making schedule decisions and adjusting the frequencies of cores. However, as observed from Figure 1, besides the computational ability of cores, the latency of data access also significantly affect the performance of parallel applications, especially memory-bound applications. As prior techniques, such as WATS and AAWS, overlook data access latency when making scheduling decision, they often results in the poor performance of memory-bound applications.

In order to improve the performance of applications considering both the computational ability of cores and the latency of data access in AMC architecture, we propose *Selective Asymmetric-aware Work-Stealing* (SAWS) runtime system that consists of an *asymmetry-aware task allocator* and a *selective work-stealing scheduler*. The asymmetry-aware task allocator is a heuristic task-spawning policy based on program's topology and iteration's history, and used to effectively allocate tasks in AMC architecture. The selective work-stealing scheduler prefers to steals intra-socket tasks and then takes advantage of power slack provided by waiting cores to steal inter-socket tasks or adjust the frequencies of executing cores with DVFS technology. In our target scenarios, system's power consumption is restricted to power constraints.

The primary contributions of our work are as follows.

- We propose an asymmetry-aware task allocator that heuristically allocates tasks into asymmetry sockets according to iterations' history, so that the workload is balanced across asymmetric sockets.
- We propose a selective work-stealing scheduler to schedule tasks accordingly so that most tasks can access data from fast local memory node. In more detail, the scheduler schedules tasks within sockets first and then automatically select between power slack monopoly and power slack distribution strategies for the purpose of higher performance and energy efficiency.

Our real-system experimental results show that SAWS can improve the performance of memory-bound programs up to 59.3% compared with traditional work-stealing schedulers in AMC architecture. This paper is organized as follows. Section 2 explains the motivation of SAWS. Section 3 gives an overview of SAWS. Section 4 and Section 5 present the asymmetry-aware task allocator and selective work-stealing scheduler respectively. Section 6 introduces the implementation of SAWS. Section 7 evaluates SAWS. Section 8 discusses the related work. Section 9 draws conclusions.

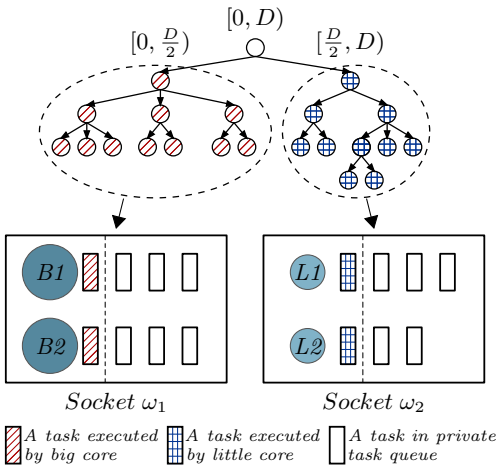


Fig. 2: Task allocation of a general task graph for iterative D&C programs in the first iteration.

II. Problems in Emerging Work-stealing

Similar to many popular work-stealing schedulers (e.g., Cilk [1], and LAWS [4]), this paper targets iterative Divide-and-Conquer (D&C) programs that have tree-shaped task graph. Most stencil programs and algorithms based on jacobi iteration (e.g., Heat distribution simulation and Successive Over Relaxation) are examples of iterative D&C programs. In D&C programs, as shown in Figure 2, the root task process the whole dataset $[0, D)$ of the program, and after several recursions of task spawning, each leaf task is allocated with a small portion of the whole dataset.

Suppose the iterative D&C program in Figure 2 runs on an AMC architecture with a NUMA-based memory system. For NUMA, Linux memory management for NUMA adopts *first touch strategy*, so when a chunk of data is first accessed by a core of socket i, then a physical page of the local memory of socket i is allocated to the data. In our work, we utilize this feature of Linux memory management.

Observed from Figure 2, emerging work-stealing schedulers encounter two main problems when scheduling tasks in AMC architectures. First, most tasks have to access their data from remote memory nodes in all the iterations due to the random task stealing. Second, the workload is not balanced across asymmetric sockets.

As for the first problem, in the scenario shown in Figure 2, suppose task t is the first task that accesses the part of the data $[a, b)$ ($a > 0$, $b < D$). If task t is scheduled to socket w_1, then the part of the data is stored in the memory node attached to w_1. Due to the random scheduling in work-stealing, it is highly possible that the task t is scheduled to another socket w_2 in the following iterations. In this case, t executed by cores in socket w_2 has to access data from remote memory node attached to socket w_1, which significantly degrades the performance of memory-bound applications. LAWS [4] partly solved the first problem by always evenly distributing the

data set of a parallel program to all the memory nodes, and schedule a task to the socket where its data is stored. However, due to the unawareness of performance asymmetry of sockets, the workload is not balanced across sockets, which in turn degrades the performance of memory-bound applications.

As for the second problem, because different sockets have different computational abilities in AMC architectures, it is challenging to appropriately balance the tasks to the asymmetric sockets. Given an AMC architecture, tasks in different programs have different speed up ratios on the big core compared with the little core because different programs have different features. Therefore, it is not reasonable to find a static optimal task allocation that can always balance the workload across asymmetric sockets. Targeting this problem, AAWS [14] tried to balance the workload between asymmetric cores by adjusting the computational abilities (frequencies) of asymmetric cores accordingly. However, the tasks are still randomly scheduled between sockets and memory-bound applications still suffer from severe remote memory accesses, which seriously degrade their performance in consequence in real-system AMC architectures.

To solve the above problems, we propose *Selective Asymmetry-aware Work-Stealing* (SAWS) runtime system composed of an asymmetry-aware task allocator and a selective work-stealing scheduler. The asymmetry-aware task allocator makes use of the first touch strategy and asymmetry of AMC architecture to automatically schedule tasks to proper sockets according to its dataset distribution and heuristically adjust task allocation by iterations' history to balance the workload. However, absolute load balance cannot be guaranteed due to system noise and variations. To solve this problem, the selective work-stealing scheduler takes power slack provided by waiting cores in w into consideration, and introduces *power slack monopoly (PSM)* and *power slack distribution (PSD)* strategies to further improve the performance of parallel programs. PSM strategy is more commonly used and PSD strategy works better when intra-socket work-stealing fails in memory-bound applications, which will be further discussed in Section V.

III. OVERVIEW OF SELECTIVE ASYMMETRY-AWARE WORK-STEALING RUNTIME

This section presents an overview of our selective asymmetry-aware work-stealing runtime system. Figure 3 shows the processing flow of a parallel program in SAWS.

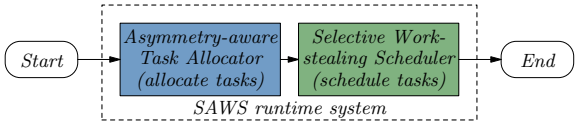


Fig. 3: The processing flow of a parallel program in SAWS.

In the first iteration/initialization phase, the asymmetry-aware task allocator first distributes the whole dataset evenly according to the number of sockets. In the following iterations, it carefully adjusts the task distribution decided in the first iteration or initialization phase on the basis of the previous iteration's execution status. For instance, when iteration i completes, SAWS analyzes the execution status and adjusts the task allocation in iteration $i+1$ (to be presented in Section IV).

In the selective work-stealing scheduler, in order to improve the performance under power constraint, we attempt to use DVFS technology to dynamically lower the frequencies of waiting cores. And we find that the power slack obtained by waiting cores can be utilized mainly in two ways. First, the waiting cores can use up their own power slack by work-mugging, work-stealing or work-sharing techniques. In our work, work-sharing is not suitable for NUMA-based memory system and work-mugging may cause extra overhead. Thus, we adopt work-stealing technique for waiting cores to steal tasks and transform back to executing state. Second, other executing cores can share the power slack, and waiting cores remain waiting. This strategy works when most task queues are empty but executing cores are still processing, or the cost of inter-socket work-stealing is much greater than the profit earned by waking up these waiting cores. Therefore, our selective work-stealing scheduler prefers to steal intra-socket tasks so as to better utilize data locality, then provide a selective solution to monopoly or distribute the power slack provided by waiting cores, and when most task queues turn to be empty, maximize all the executing cores by means of power slack distribution (to be presented in Section V).

It is worthing noting that SAWS provides a heuristic solution without any extra information offline profiled by users.

IV. ASYMMETRY-AWARE TASK ALLOCATOR

In a multi-socket multi-core architecture with a NUMA-based memory system, Cilk[1]'s random task scheduler can result in severe shared cache misses and remote memory access in a NUMA node. With respect to memory-bound applications, remote memory access can significantly worsen the performance. LAWS[4] proposes a load-balanced task allocator which divides tasks according its data set.

However, the ratio of memory cycles to total runtime is not likely to be 100%, for example approximately 70% for a memory-bound benchmark mcf[15]. Thus, in an asymmetry Multi-Socket Multi-Core (MSMC) architecture, a load-balanced task allocation policy should take into account shared cache hit rate and asymmetry processing capabilities of big and little cores.

The knowledge of task workloads is essential to optimal task scheduling in asymmetry multi-core architecture[13]. It's acceptable to obtain such workload information by means of offline profiling. Nevertheless, in iterative Divide-and-Conquer (D&C) programs, we can take advantage of shared cache misses in each iteration to heuristically adjust next iteration's task allocation.

In our work, we define *a balancing factor* α as the load ratio of big and small cores, and α determines the size of each socket's dataset. Supposing that a parallel program with the whole dataset [*0,D*) runs on asymmetry MSMC

Algorithm 1 Algorithm for heuristically updating balancing factor α

Require: $MAX\_ITER > 0$

```
f_i = 0, f_{i-1} = 0      // frequencies of iteration i & i-1
i ← 0                     // iteration i
α ← 1                     // α initialization
for iteration  i ← 0  to  MAX_ITER do
    Spawn tasks and execute iteration i under the new α;
    Record the frequency f_i of inter-socket work-stealing
    in this iteration;
    if f_i ≤ f_{i-1} then
        α ← α + 2^i / (f_{i-1} − f_i)      // update the alpha
        f_{i-1} ← f_i                      // update the last frequency
    else
        α ← α − 2^(i-1) / (f_{i-1} − f_i)  // set back to the last alpha
        break;
    end if
end for
```

architecture with two big sockets and one small socket, two big sockets occupy the dataset [*0*, $\frac{\alpha}{\alpha+1}\cdot$ *D*) and then the two big sockets evenly split dataset [*0*, $\frac{\alpha}{\alpha+1}\cdot$ *D*) while the small socket occupies the last $\frac{1}{\alpha+1}$ of the whole dataset.

In the first iteration, α is initialized as 1 so the whole dataset is evenly split into two parts for big and small cores. SAWS runtime records the frequency of inter-socket work-stealing between big and small sockets in every iteration. It can be easily validated that the total execution time changes as α scales, and there exists an α value where the total execution time reaches its minimum. But when the α increases to greater than some threshold, the performance will dramatically degrade because an over-large α makes small cores out of work. Thus, in our task allocator, we target the local optimum of the total execution time rather than taking risk of a over-large α. In Algorithm 1, we obtain the frequency $f_{(i-1)}$ of inter-socket work-stealing between big small sockets in last iteration i-1, and if the frequency f_i of current iteration *i* is not more than $f_{(i-1)}$, we attempt to increase α by a given step until $f_i > f_{(i-1)}$. It's worth noting that the step length decides convergence speed. It consumes a relatively long time to converge if the step length is too short, but it may cause over-convergence or missing the local optimum with the step length too long. Thus, in our updating algorithm, we adopt an incremental method in which the α is updated by an incremental step length 2^i in iteration i and in order to control the size of α, the step length is divided by the difference between the frequencies of last iteration and current iteration. When $f_i > f_{(i-1)}$, α stops increasing and minus a step in last iteration since the α in last iteration finds the local optimum of total execution time. The approach of our task allocator partially origins from LAWS[4], but our asymmetry-aware task allocator is based on the above heuristic algorithm to reach the local optimal of load balance in asymmetry environment.

V. Selective Work-Stealing Scheduler

In a multi-socket multi-core system with N_B big cores and N_L little cores, we assume that each core's frequency can be scaled respectively, and call the cores out of work, *waiting cores* and other cores executing tasks, *executing cores*. We suppose that energy model of such systems is a third-order model [16]:

$$\begin{aligned} P_{Bi} &= \lambda_B \cdot f_{Bi}^3 \qquad (i = 1, 2, ..., N_B) \\ P_{Li} &= \lambda_L \cdot f_{Li}^3 \qquad (i = 1, 2, ..., N_L) \end{aligned} \tag{1}$$

where P_{Bi} is the power of big core i, f_{Bi} is the frequency of big core i, and so on.

We use N_{BW} and N_{LW} to refer to the number of big waiting cores and little waiting cores. When $N_{BW} \neq 0 \vee N_{LW} \neq 0$ is true, there is at least one waiting core. In a asymmetry multicore system, we assume that the frequencies of big cores can be scaled from $f_{B,max}$ to $f_{B,min}$, and the normal frequency of big cores is $f_{B,norm}$. Likewise, the frequencies of little cores can be scaled from $f_{L,max}$ to $f_{L,min}$ ($f_{B,max} \geq f_{L,max}, f_{B,min} \geq f_{L,min}$), and the normal frequency of little cores is $f_{l,norm}$ ($f_{B,norm} \geq f_{L,norm}$).

A. Power Slack

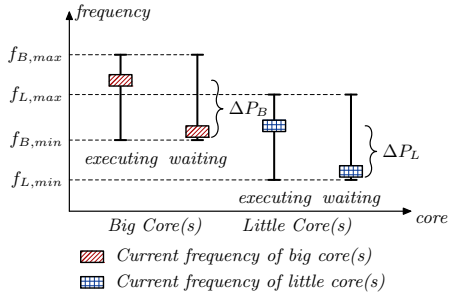


Fig. 4: Power slack from big and little core(s).

One of our goals is to maximize performance with normal power consumption constraints $P_N = N_B \cdot P_{BN} + N_L \cdot P_{LN}$. And in memory-bound applications, work-stealing strategy will cause remote memory access, which implies that it is an option to adopt work-stealing technique or adjust core's frequency by DVFS technique. Figure 4 shows a moment of $N_{BW} \neq 0 \vee N_{LW} \neq 0$ where task queues of some big and/or small cores happen to be empty. Under the consideration of energy saving, these waiting cores will be set to be the minimal frequencies, and executing cores are still at the normal frequencies. We call the energy saving from waiting cores' lower frequencies, *power slack*, noted as ΔP.

$$\begin{aligned} \Delta P &= \Delta P_B + \Delta P_L \\ &= N_{BW} \cdot (P_{B,norm} - P_{B,min}) \\ &+ N_{LW} \cdot (P_{L,norm} - P_{L,min}) \end{aligned} \tag{2}$$

When waiting cores exist, i.e. $N_{BW} \neq 0 \vee N_{LW} \neq 0$, there are two strategies to utilize their power slack,

TABLE I: States of an AMC system

N_{BW} N_{LW}	State
$N_{BW} = 0 \wedge N_{LW} = 0$	S(0,0)
$N_{BW} = 0 \wedge N_{LW} \neq 0$	S(0,1)
$N_{BW} \neq 0 \wedge N_{LW} = 0$	S(1,0)
$N_{BW} \neq 0 \wedge N_{LW} \neq 0$	S(1,1)

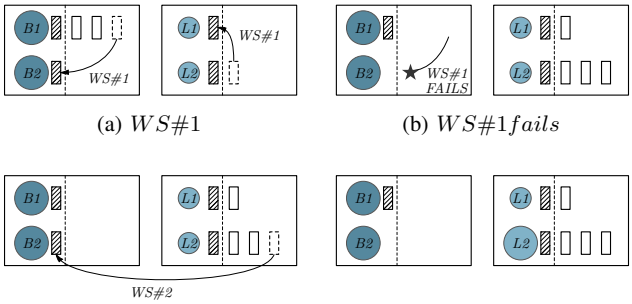
power slack monopoly and power slack distribution. First, *power slack monopoly* (PSM) strategy means that the core ω which devotes the power slack uses up its own power slack, in other words steals tasks from another core's private task queue and transforms back to an executing core. And *power slack distribution* (PSD) strategy is that other cores can share the power slack devoted by ω to higher their frequencies, but waiting cores remain. The decision of which strategy is determined by the cost of shared cache misses. Taking data locality into account, when all waiting cores are big cores or small cores, i.e. $N_{BW} \neq 0 \wedge N_{LW} = 0$ (*S(1,0)* in Table I) or $N_{BW} = 0 \wedge N_{LW} \neq 0$ (*S(0,1)* in Table I), intra-socket work-stealing strategy is perferred which actually is PSM, because it doesn't arise remote memory access. When $N_{BW} \neq 0 \wedge N_{LW} \neq 0$ (S(1,1)) and then intra-socket work-stealing cannot transform this state into State(0,0), it implies that all the task queues are empty but some executing cores are still processing their remaining tasks. In such a state, work mugging and work stealing techniques make no sense. So to utilize the power slack, it performs better to scale up executing cores' frequencies which actually is PSD.

B. Work-stealing

In our selective work-stealing scheduler, we introduce both power slack monopoly and power slack distribution so as to make an optimal work-stealing decision.

First, we prefer to use PSM strategy in most situations. In the first iteration when our asymmetry-aware task allocator must not converge, it's likely to cause load imbalance between cores. Then our scheduler schedules tasks by means of intra-socket work-stealing with the highest priority. As shown in Figure 5a, intra-socket work-stealing is denoted by *WS#1*, and when an executing core transforms to the waiting state and its private task queue is out of work, the scheduler will detect other private task queues in the same socket first. If an intra-socket task queue is targeted, the top task of it will be stolen for the waiting core, which monopolies its own power slack and turns back to an executing core. In the following iterations, with our task allocator gradually reaching the local optimum of task allocation, load imbalance between big sockets and little sockets is mitigated and *WS#1* still works but its frequency reduces.

As mentioned above, intra-socket work-stealing, i.e. *WS#1* may fail since in the case of severe load imbalance between big and small sockets in Figure 5b all the private task queues in the big sockets are empty and the waiting big core *B2* fails to steal task from *B1*. When this situation occurs, there are two optional strategies: power slack monopoly and power slack distribution. As for the former, due to load imbalance, private task queues in other sockets are likely to be nonempty and therefore inter-socket work-stealing denoted by *WS#2* may succeed to steal tasks from other sockets as Figure 5c. As for the latter, *B2* remains waiting and *L2* utilizes the power slack to higher its own frequency so without task and data migrations *L2* executes its tasks faster. When the size of dataset allocated to socket ω is smaller than that of shared cache in socket ω, PSM strategy should be adopted because although shared cache misses may occur due to inter-socket work-stealing, the data of the stolen task will be cached in the thief's shared cache in the following iteration. Otherwise, PSD strategy is preferred since the stolen tasks have to access remote memory after inter-socket work-stealing in PSM strategy.



(a) $WS\#1$ (b) $WS\#1fails$

(c) PSM strategy (d) PSD strategy

Fig. 5: Four examples to illustrate the situations where SAWS runtime works – (a) intra-socket work-stealing; (b) fails to steal intra-socket tasks; (c) power slack monopoly strategy; (d) power slack distribution strategy.

Moreover, when most of the private task queues turn out to be empty in both big and small sockets, PSD strategy is adopted. In such case, intra-socket work-stealing must fail and then inter-socket work-stealing makes no sense and work-mugging technique also fails to improve the performance. Therefore, our scheduler directly decides to distribute the power slack provided by waiting cores and use DVFS technology to higher the frequencies of executing cores. This method is progressive. executing cores can obtain the frequency increments decided by the number of waiting cores and stop the progress when reaching their maximal frequency.

VI. EVALUATION

In the evaluation section, we describe our implementation, experiment environment, performance evaluation compared to MIT Cilk[1], AAWS[14] and LAWS[4], and effectiveness of SAWS.

MIT Cilk[1] is one of the earliest work-stealing runtimes, and SAWS is implemented on the basic of Cilk. In our implementation, we have majorly modified the runtime of Cilk, including its runtime initialization and scheduler. In the runtime initialization, we need to initialize α, inter-socket frequencies of each socket and so on for the heuristic algorithm

of updating α and per-socket information of task queues for scheduler. In the scheduler, according to above per-socket information, task is spawned according to its dataset and scheduled first by intra-socket work-stealing. And in every iteration, scheduler adjust its packing decision according to the updated α.

In order to implement both PSM and PSD strategies described in last section, We use the "libpfm" and "libcpufreq" libraries in Linux kernel to collect private cache misses of tasks and dynamically adjust the current frequencies of cores with DVFS technology.

TABLE II: System configuration

Big Core	Intel Xeon CPU X7560@2.27GHz, 2.26GHz maximal frequency, 1.06GHz minimal frequency, 2.00GHz normal frequency.
Little Core	Intel Xeon CPU X7560@2.27GHz, 2.26GHz maximal frequency, 1.06GHz minimal frequency, 1.33GHz normal frequency.
Caches	4-ways 32KB L1I, 8-ways 32KB L1D, 8-ways 256KB L2 per core; 24-ways shared 24MB L3.
DRAM	Samsung DDR3 PC3-10600.

We use a server which has four Intel 8-core Xeon X7560 processors with hyper-threading technology to evaluate the performance and effectiveness of SAWS. Table II includes details on the architectures and memory system of our server. The server adopts NUMA memory system, so the term a NUMA node also denotes a socket in our paper. Each core has a 256KB private L2 and each NUMA node is equipped with a shared 24MB L3, which are shared by all the cores in the same socket. And the version of Linux on the server is 3.13.0-32-generic. In our evaluation, half of the processors are big cores with 2GHz normal frequency $f_{B,norm}$ and the other half are little cores with 1.33GHz normal frequency $f_{L,norm}$ as Table II illustrates. The maximal frequencies of big and little cores are same, so are the minimal frequencies of them. All of them are scaled by means of DVFS technology.

We compare the performance of SAWS runtime system with MIT Cilk, AAWS and LAWS. Cilk spawns and schedules tasks based on the child-first policy. AAWS uses a marginal-utility-based approach to narrow the frequency gap in high parallel region and mugs tasks from little cores to big cores in low parallel region. LAWS packs the execution DAG of a parallel program into subtrees and adjust the packing for the optimal to reduce shared cache misses. But LAWS does not take the asymmetry architecture into account.

SAWS is implemented by modifying MIT Cilk out of fairness in comparison and we also evaluate Cilk without any modification and AAWS on basic of MIT Cilk.

TABLE III: Benchmarks in the experiments

Name	Description
heat/heat-ir	2D heat distribution (regular/irregular)
gauss/gauss-ir	Gaussian elimination (regular/irregular)
9p/9p-ir	9-point 2D stencil computation (regular/irregular)
25p/25p-ir	25-point 3D stencil computation (regular/irregular)
6p/6p-ir	6-point 3D stencil computation (regular/irregular)

Table III gives a list of benchmarks in our experiments so that we can evaluate the performance of SAWS compared to Cilk, AAWS and LAWS in different scenarios. Each benchmark has a regular and an irregular execution DAG versions. The irregular execution DAG version of each benchmark is implemented with the same algorithm except their irregular execution DAGs as suggested in [17]. And all the benchmarks are compiled with option "-O2".

A. *Performance evaluation*

Figure 6 illustrates the performance of all the benchmarks in MIT Cilk, AAWS, LAWS and SAWS.

In this experiment, for *heat* and *heat-ir*, the input data is a 8192×2048 matrix. For *gauss* and *gauss-ir*, the input data is a 2048×2048 constrained by algorithm. For *9p* and *9p-ir*, the input data is a 4096×2048 matrix. And for *25p, 25p-ir, 6p* and *6p-ir*, the input data is a $2048 \times 128 \times 128$ matrix.

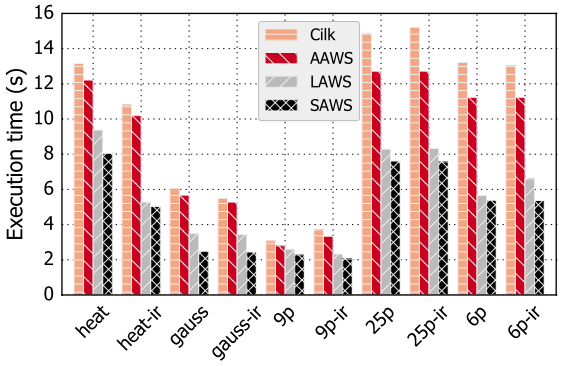


Fig. 6: The performance of all the benchmarks in MIT Cilk, AAWS, LAWS and SAWS.

When compared to Cilk in Figure 6, we can observe that SAWS performs much better than Cilk, and make a total execution time reduction of 25.89% to 59.29%. AAWS improves the performance of benchmarks from 3.88% to 16.89%, and LAWS also obtains the performance improvement which ranges from 15.52% to 57.25%.

During the implementation of SAWS runtime, we have attempted to adopt NUMA page migration technique to migrate data from the *victim* socket to the *thief* after inter-socket work-stealing occurs. Before our experiments, we once held the view that if inter-socket work-stealing just steals tasks to achieve load balancing, remote memory access still worsens the performance. However, NUMA page migration technique

TABLE IV: Shared cache misses of all the benchmarks (*1E6)

	heat	heat-ir	gauss	gauss-ir	9p	9p-ir	25p	25p-ir	6p	6p-ir
Cilk	799	667	275	264	162	160	2861	2865	1030	1031
AAWS	752	653	264	261	156	156	2826	2842	995	972
LAWS	477	577	35	35	105	108	2479	2469	471	452
SAWS	460	565	27	30	103	107	2445	2438	465	445

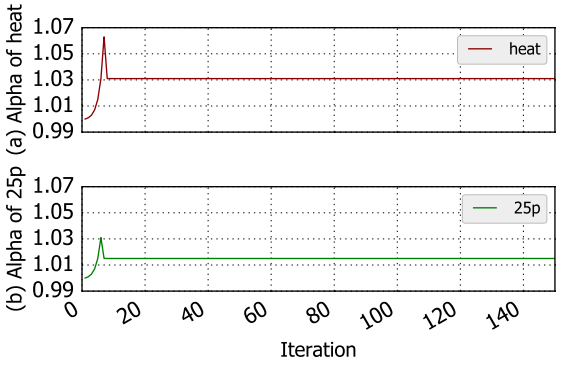


Fig. 7: Tracing the dynamic alpha values in iterative programs –(a) alpha values of heat; (b) alpha values of 25p.

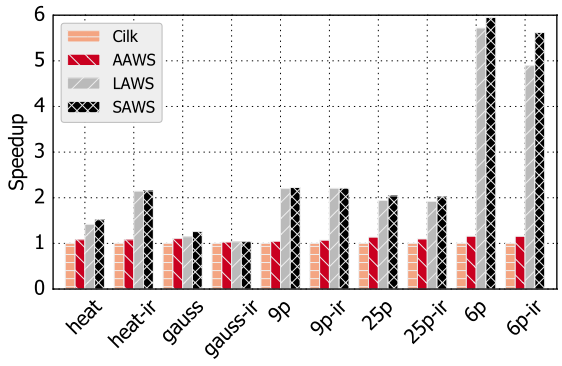


Fig. 8: The speedup of all the benchmarks in Cilk, AAWS, LAWS and SAWS.

moves all pages in a process to another set of nodes. In our experiments, it only slightly improves the performance of benchmarks, although it indeed reduces the remote memory access of the benchmarks, especially *9p/9p-ir*. This is mainly because the penalty of page migrations cancels the gains of remote memory access reduction.

As discussed in previous sections, load balancing makes a difference in the performance of memory-bound applications and depends on effective reduction in low parallel region.

In order to explain the performance of SAWS, We have collected shared cache misses in all the benchmarks and traced the dynamic alpha values in each iteration. As Table IV illustrates, shared cache misses of each benchmark are reduced in SAWS when compared to Cilk, AAWS and LAWS. SAWS's asymmetry-aware task allocator allocates tasks to the sockets where its data locate as far as possible and take system's asymmetry into consideration, so with load-balancing task allocation and less inter-socket work-stealing, tasks are likely to access local memory nodes and neighbor tasks in the same socket take advantage of shared cache to preserve shared data. As Figure 7 shows, *Heat* and *25p* reach their optimal alphas in 20 iterations. Since our heuristic updating algorithm is based on the frequency of inter-socket work-stealing, from the figure we can find that inter-socket wort-stealing effectively reduces as α reaches its local optimum.

B. Scalability of SAWS

In order to evaluate the scalability of SAWS, we increase the data sizes of benchmarks in Cilk, AAWS, LAWS and SAWS.

In scalability evaluation, for *heat*, *heat-ir*, *9p* and *9p-ir*, the input data is a 8192×4096 matrix. And for *25p, 25p-ir, 6p* and *6p-ir*, the input data is a $8192 \times 128 \times 128$ matrix.

As we can see from Figure 8, the speedups of SAWS over Cilk are greater than that of other runtimes, and the max speedup of SAWS can reach 5.95 in *6p* evaluation. The reasons why the speedups of LAWS and SAWS in *6p* are both greater than 5 is that the performance of *6p* in Cilk and AAWS suffers a lot from the load imbalance caused by the lack of proper task spawning policy, and both LAWS and SAWS respectively adopts their own task spawning policies based on dataset distribution.

VII. Related work

The importance of reducing shared cache misses and increasing local memory access in MSMC architecture with NUMA-based memory system has been widely accepted. And as for asymmetry architecture, load balancing make a difference because the wider the low parallel region is, the more task re-scheduling occurs, and consequently data locality can not be guaranteed.

Work-stealing technique is one of popular techniques to schedule tasks in order to reach load balancing on asymmetry architecture. There are plenty of works have adopted work-stealing so as to improve the performance or achieve energy efficiency in their runtime systems [1], [4], [14], [13], [17], [18], [19], [20]. With respect to NUMA-based memory system, many studies have achieved good results in specific applications [21], [22] or targeted general applications [23], [24].

LAWS [4] has adopted work-stealing technology and an adaptive DAG packer to reduce share cache misses and remote memory access. However LAWS did not consider the asymmetry architecture. AAWS [14] has proposed work-sprinting, work-pacing and work-mugging techniques to improve the performance of asymmetry architecture, but AAWS did not

take NUMA-based memory system into account. So shared cache misses can be severe when AAWS executing memory-bound applications. In CAB [25] and HWS [26] have used a rigid boundary level to divide tasks and schedule tasks into their local sockets, but extra information provide by users is needed for HWS and CAB does not achieve the optimal packing as LAWS does. As for the above problems, we target memory-bound applications and correspondingly propose an asymmetry-aware task allocator to evenly allocate task in asymmetry arhitectures by means of a heuristic tasking allocation algorithm without any extra information from users, and a selective work-stealing scheduler to reduce shared cache misses in MSMC architecture with NUMA-based memory system.

VIII. Conclusion

Traditional work-stealing schedulers suffer from poor data locality and load imbalance problems in AMC architectures with NUMA-based memory system. In order to solve the two problems, we present SAWS, which consists of an asymmetry-aware task allocator and a selective work-stealing scheduler, to automatically realize load balancing during the execution runtime. The task allocator allocates tasks to specific sockets with the knowledge of their dataset and adopts a heuristic algorithm to dynamically adjust task allocation out of consideration of system's asymmetry. The selective work-stealing scheduler improves the parallel programs' performance by means of power slack monopoly and power slack distribution strategies with energy efficiency guaranteed. For typical iterative Divide-and-Conquer algorithms, SAWS runtime can achieve 59.3% performance improvement compared with traditional work-stealing schedulers.

Acknowledgment

This work was partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC)(No. 61602301, 61261160502, 61272099), the Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), the Scientific Innovation Act of STCSM (No. 13511504200).

References

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

[2] C. E. Leiserson, "The Cilk++ Concurrency Platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[3] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. " O'Reilly Media, Inc.", 2007.

[4] Q. Chen, M. Guo, and H. Guan, "LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures," in *Proceedings of the 28th ACM international conference on Supercomputing*, ser. ICS '14. ACM, 2014, pp. 3–12.

[5] R. D. Blumofe, "Executing Multi-threaded Programs Efficiently," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

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

[7] M. Dean and P. Norman, "Heterogeneous Chip Multiprocessors," *IEEE Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[8] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," in *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, 2004, pp. 64–75.

[9] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The Impact of Performance Asymmetry in Emerging Multicore Architectures," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 506–517.

[10] P. Greenhalgh, "Big.Little Processing With Arm Cortex-A15 & Cortex-A7," *ARM White paper*, pp. 1–8, 2011.

[11] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih *et al.*, "QuickIA: Exploring Heterogeneous Architectures on Real Prototypes," in *HPCA*. IEEE, 2012, pp. 1–8.

[12] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[13] Q. Chen, Y. Chen, Z. Huang, and M. Guo, "WATS: Workload-Aware Task Scheduling in Asymmetric Multi-Core Architectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 249–260.

[14] C. Torng, M. Wang, and C. Batten, "Asymmetry-Aware Work-Stealing Runtimes," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 40–52.

[15] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System Level Analysis of Fast, Per-Core DVFS Using on-Chip Switching Regulators," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 2008, pp. 123–134.

[16] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya, "Some Observations on Optimal Frequency Selection in DVFS-Based Energy Consumption Minimization," *Journal of Parallel and Distributed Computing*, vol. 71, no. 8, pp. 1154–1164, 2011.

[17] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache-Aware Task-Stealing Based on Online Profiling in Multi-Socket Multi-Core Architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. ACM, 2012, pp. 163–172.

[18] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable Work Stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC'09. ACM, 2009, p. 53.

[19] D. Chase and Y. Lev, "Dynamic Circular Work-stealing Deque," in *Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2005, pp. 21–28.

[20] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A Scalable Locality-Aware Adaptive Work-stealing Scheduler," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[21] M. Shaheen and R. Strzodka, "NUMA-Aware Iterative Stencil Computations on Many-Core Systems," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 461–473.

[22] 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 *Parallel & Distributed Processing, 2009. (IPDPS 2009). IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[23] L. Pilla, C. Ribeiro, D. Cordeiro, A. Bhatele, P. Navaux, J. Méhaut, and L. Kalé, "Improving Parallel System Performance With a NUMA-aware Load Balancer," 2011.

[24] B. Vikranth, R. Wankar, and C. R. Rao, "Topology-Aware Task Stealing for On-chip NUMA Multi-Core Processors," *Procedia Computer Science*, vol. 18, pp. 379–388, 2013.

[25] Q. Chen, Z. Huang, M. Guo, and J. Zhou, "Cab: Cache-Aware Bi-tier Task-stealing in Multi-Socket Multi-Core Architecture," in *ICPP*. IEEE, 2011, pp. 722–732.

[26] J.-N. Quintin and F. Wagner, "Hierarchical Work-stealing," in *European Conference on Parallel Processing*, 2010, pp. 217–229.