Stable Adaptive Work-Stealing for Concurrent Many-core Runtime Systems

Yangjie Cao[1], Hongyang Sun[2], Depei Qian[3], and Weiguo Wu[1]

[1]School of Electronic and Information Engineering, Xi'an Jiaotong University, China
[2]School of Computer Engineering, Nanyang Technological University, Singapore
[3]School of Computer Science and Engineering, Beihang University, China
caoyj@stu.xjtu.edu.cn, sunh0007@ntu.edu.sg, depeiq@buaa.edu.cn, wgwu@mail.xjtu.edu.cn

Abstract: The proliferation of many-core architectures has led to the explosive development of parallel applications using programming models, such as OpenMP, TBB, and Cilk/Cilk++. With increasing number of cores, however, it becomes even harder to efficiently schedule parallel applications on these resources since current many-core runtime systems still lack effective mechanisms to support collaborative scheduling of these applications. In this paper, we study feedback-driven adaptive scheduling based on work stealing, which provides an efficient solution for concurrently executing a set of applications on many-core systems. To dynamically estimate the number of cores desired by each application, a stable feedback-driven adaptive algorithm, called SAWS, is proposed using active workers and the length of active deques, which well captures the runtime characteristics of the applications. Furthermore, a prototype system is built by extending the Cilk runtime system, and the experimental results, which are obtained on a Sun Fire server, show that SAWS has more advantages for scheduling concurrent parallel applications. Specifically, compared with existing algorithms A-Steal and WS-EQUI, SAWS improves the performances by up to 12.43% and 21.32% with respect to mean response time respectively, and 25.78% and 46.98% with respect to processor utilization, respectively.

Keywords: Many-core architectures, Many-core runtime systems, Feedback-driven adaptive scheduling

1 Introduction

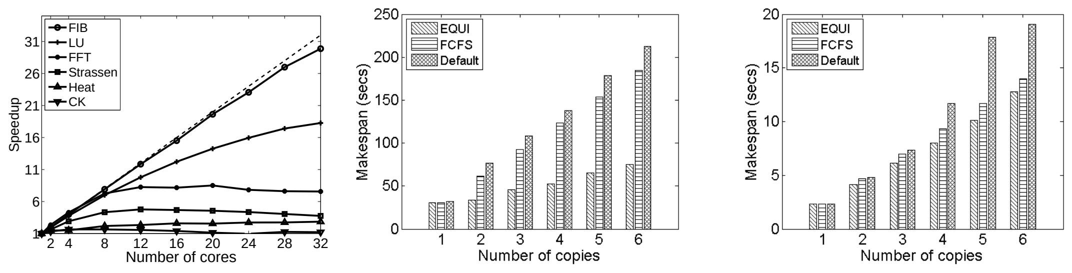
Recent developments in microprocessor design show a clear trend towards many-core architectures. In the near future, it will be common to have a many-core processor with hundreds or even thousands of cores on the chip [1]. Exploiting all the advantages offered by these abundant cores, however, will be a great challenge because it is not trivial to efficiently utilize the available computing power.

To exploit the hardware resources of modern processors, various programming models for many-core systems have been developed, such as OpenMP [2], TBB [3], Cilk/Cilk++ [4] which is recently extended to Intel Cilk Plus [5]. Compared with other parallel programming models, such as MPI and POSIX threads, these models, supported by their flexible runtime systems, provide good programmability, portability, and ability to manage dynamic parallelism for many-core systems. When using these programming models in practice, however, many issues remain to be addressed to efficiently utilize the increasing number of cores.

First, current many-core runtime systems may have poor scalability. It is a typical requirement in most many-core runtime systems to explicitly or implicitly (via function calls) specify the number of cores to use for the execution of an application. As more cores are becoming available, many applications will start to experience diminishing returns with increased processor allocation. Without knowing the execution characteristic of the application on a particular hardware platform, simply allocating all available cores to the application may not ensure satisfying performance. As we can see in Fig.1(a), which is conducted on a Sun Fire server using several Cilk applications[1], only a couple of applications, namely FIB and LU, have nearly linear speedup when increasing the number of allocated cores.

Second, competitions for processor resources are unavoidable in current many-core runtime systems. It is very common for multiple users or applications to share a high-performance computing platform nowadays. Using current solutions by themselves, the performance may not scale well with increasing number of cores, particularly in the presence of concurrently running parallel applications. To demonstrate this with an example, Fig.1(b) and Fig.1(c) give the results of running multiple copies of two Cilk applications on a Sun Fire server**. As shown in these figures, the overall running time (makespan) of both applications under the default scheduler, which runs each copy of the Cilk application on

[1]The detailed information related to this experiment can be found in Section 5.



(a) Speedup (b) CK (c) LU

Figure 1: Speedup and makespan comparison of different scheduling strategies using Cilk applications.

all available cores, becomes much worse than that of the FCFS (First-Come First-Serve) scheduler when increasing the number of concurrently running copies.

Aiming at addressing these problems in the current many-core runtime systems, adaptive scheduling algorithms are studied in this paper. Compared to scheduling all applications in a time-sharing manner as described above, adaptive scheduling based on space-sharing seems to provide a more efficient solution for simultaneously executing a set of applications. Since the parallelism of most applications often changes over time, adaptive scheduling takes advantage of the application malleability by dynamically allocating a variable number of processors to each job during runtime, thus it is able to achieve better utilization of the available resources. Fig.1(b) and Fig.1(c) also show the results of running the same set of applications as described previously, but with the simple space-sharing scheduler EQUI (Equi-Partitioning) [6] which at any time divides the total number of cores evenly among all running jobs. The results demonstrate that EQUI has much better performance in terms of makespan, especially when the applications have sublinear speedups. While this simple example shows the benefit of adaptive scheduling, in the rest of this paper we will study more effective mechanisms that can better capture and explore the parallelism variations of the jobs.

Although some existing work have studied adaptive scheduling, most results are based on theoretical analysis and simulation approaches [7, 8, 9, 10, 11]. Unlike these results, in this paper we study the benefits of adaptive scheduling based on solid experiments conducted on practical systems and actual workloads. The adaptive runtime system we build is based on the well-known work-stealing strategy, which has been shown to have good performances from both theoretical and practical perspectives [4, 12]. The main contributions of the paper are the following:

- An adaptive runtime system is implemented based on the work-stealing load balancing strategy. The runtime system has the ability to dynamically change the number of cores allocated to each job so that it can effectively exploit the runtime characteristics of the jobs, and more importantly it eliminates the need of manually specifying the number of cores required by most existing many-core runtime systems.

- To dynamically estimate the number of cores desired by each job, a stable feedback algorithm, called SAWS, is proposed using active workers and the length of active deques. Compared to existing algorithms, SAWS captures more precisely the parallelism of the jobs, and more importantly it solves the desire instability problem of an existing algorithm.

- A prototype system is built by heavily modifying the original Cilk runtime system. The experimental results show that feedback-driven algorithms have more advantages for scheduling parallel applications with dynamic changing parallelism, and better overall performance will be achieved with more accurate and stable feedback mechanism.

The rest of this paper is organized as follows: Section 2 briefly introduces adaptive scheduling based on work stealing. Section 3 describes how to obtain stable parallelism feedback using active workders and the length of active deques. Section 4 gives the detailed implementation of the adaptive scheduling framework. Our experimental results are presented in Section 5 and Section 6 concludes the paper.

2 Adaptive Scheduling Based on Work Stealing

In order to present our adaptive runtime system and the feedback-driven algorithm, it is necessary to review adaptive scheduling and work stealing. In this section, we will first define the basic concepts in the two scheduling paradigms. We then discuss challenges in adaptive work stealing.

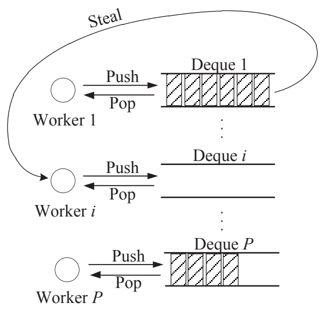


Figure 2: The paradigm of working stealing.

2.1 Work Stealing

Work stealing [12] is a popular thread-level scheduling mechanism to schedule parallel computations with dynamic parallelism. Because of the good performance and ease of implementation, it has been successfully applied to runtime systems in Cilk [4], Cilk Plus [5], TBB [3] and OpenMP [13].

In traditional work stealing, as shown in Fig. 2, an application is given a fixed set of P processors throughout execution. Each processor (or worker) maintains a double-ended queue, called *deque*, which contains the ready threads of the job. A worker treats its own deque as a stack and treats the deque of another worker as a queue. At any time, each worker works as follows: (1) when the thread it is currently running spawns a new thread, the worker pushes the parent onto the bottom of its deque and starts working on the child thread; (2) when the running thread completes or blocks, the worker checks its own deque. If the deque is not empty, it pops the thread from the bottom of the deque and starts working on it. In case the deque is empty, e.g., for Worker i in Fig. 2, the worker becomes a "thief" and starts work stealing. In this process, the thief randomly chooses another worker, called "victim", e.g., Worker 1 in Fig. 2, and removes the thread from the top of victim's deque if it is not empty. If the victim's deque is empty, the thief restarts the stealing process by randomly choosing another victim until its finds a thread to work on. Clearly, when an application first starts to run, all of its allocated processors have empty deques except one worker that works on the job's root thread.

Work stealing has been shown to have provably-efficient performances in terms of both time and space bounds [12]. Moreover, unlike centralized schedulers based on work sharing such as the Greedy scheduler [9], a work stealing scheduler operates in a decentralized manner without knowing all the available threads of a job at any time. Therefore, due to ease of implementation, it has also been shown to be an effective thread scheduling mechanism in practice.

2.2 Adaptive Scheduling

Adaptive scheduling provides an efficient solution to better utilize the available processor resources for simultaneously executing a set of applications, thus has gained popularity recently [7, 8, 9, 10, 11, 14, 15, 16, 17]. Since the parallelism of most applications often changes over time, adaptive scheduling takes advantage of the application malleability and gives a variable processor allocations to the jobs.

One common approach used in adaptive scheduling is the two-level scheduling framework [7]. In this framework, the executions of the jobs are divided into regular intervals, called scheduling quantum, and the processors are reallocated based on the interaction between the job-level thread scheduler and the global-level resource allocator or processor controller. Specifically, at the beginning of each scheduling quantum q, a thread scheduler for each job calculates its processor desire $d(q)$, that is, how many processors the job needs, in this quantum. The processor controller at the global level then based on the processor desires of all jobs and its scheduling policy decides a processor allocation $a(q)$ for the job in quantum q. This process, called *request-allocation protocol* [9], will repeat after each scheduling quantum until the completion of all jobs.

One important aspect of two-level adaptive scheduling is how to calculate processor desires from the thread scheduler. Since the future parallelism of the job is usually unknown, the desire calculation is usually based on the execution history of the job in the previous quantum, such as measurements about the job's processor utilizations or average parallelism [7, 10]. Another aspect is for the processor controller to decide the processor allocation of each job. In this paper, we use the well-known dynamic equi-partitioning (DEQ) policy [18], which we will describe in detail in Section 4.

2.3 Adaptive Work Stealing

Compared with conventional thread schedulers that use only a fixed set of processors at any time, adaptive scheduling has the additional challenge of dealing with variable processor allocations at different times. When the thread scheduler uses distributed work stealing, this task becomes even more challenging since the scheduler does not possess global information on the deques of the processors.

To handle processor changes, we adopt the concept of *mugging* [14]. While the number of processors and therefore the number of deques is fixed for an application in traditional work stealing, it is no longer the case in adaptive work stealing. In particular, when the processor allocation decreases from quantum q to $q+1$, the job loses $a(q) - a(q+1)$ processors, who may have non-empty deques. These deques, which contain ready threads for the job and therefore still belong to the job, are however not associated with any processor at this time, and thus become muggable. When any processor of the job runs out of work during quantum $q+1$, instead of immediately stealing work from another processor, it will first look for muggable deques. If there are indeed deques waiting to be mugged, it will claim any such deque as its own and starts working on its bottom-most thread. Otherwise, if there is no muggable deque, it will start stealing as normal. On the other hand, when the processor allocation increases from quantum q to $q+1$, the job gains $a(q+1) - a(q)$ additional processors with empty deque. Again, each of these processors will first look for a muggable deque, which may be available from previous quantum, before stealing work as described before.

Moreover, besides dealing with processor changes, another very important challenge in adaptive work stealing is how to calculate processor desires for a job in each scheduling quantum. In the next section, we will design a novel desire calculation strategy that directly utilizes the lengths of the active deques, which solves the desire instability problem of an existing scheduler.

3 Stable Desire Calculation Using Both Processor Utilization and Length of Active Deques

In this section, we propose an novel desire calculation algorithm, called SAWS, based on the utilization of active workers and the length of active deques. We show that the processor desires calculated by SAWS well reflects the parallelism of the job, and more importantly, it solves the desire instability problem of an existing scheduler.

3.1 A Novel Algorithm: SAWS

SAWS works based on both processor utilization and length of active deques in each scheduling quantum. Intuitively, the status of the processors in terms of whether they are busy or idle indicates the utilization of the resources allocated to a job, thus it can be used to determine the number of processors in the next quantum. Moreover, the total length of the active deques of the job at any time gives the number of ready threads that can be stolen when the job is provided with sufficient processors to execute, thus it can indicate the unexploited parallelism of the job. SAWS explores both of these indicators and computes the processor desire of the job as described in the following.

Since the processor allocation can be changed dynamically in adaptive scheduling, only a worker that is associated with a physical processor is called an active worker; otherwise it is called an inactive worker. Suppose that quantum q starts at time t_q and lasts L units of time. Since an active worker is either working, mugging, or stealing at any time $t \in [t_q, t_q + L]$, let $X_j(t)$ denote the status of the jth processor at time t, where $1 \leq j \leq a(q)$. Specifically, if processor j is either working or mugging at t, we have $X_j(t) = 1$. Otherwise, if processor j is stealing at t, we have $X_j(t) = 0$. As mugging is a result of reduced processor allocation, the time spent on mugging is considered as not wasted [8]. Apparently, $\frac{1}{L}\int_{t_q}^{t_q+L}\sum_{j=1}^{a(q)} X_j(t)dt$ represents the average number of processor cycles not wasted at any time in quantum q, thus it reflects the processor utilization in the quantum.

Let $e(t)$ denote the number of active deques of the job at time $t \in [t_q, t_q + L]$, including the muggable ones that are not attached to any processor. For the jth active deque, let $Q_j(t)$ denote its length, or the number of ready threads on the top of the deque waiting to be stolen at time t. Hence, $\frac{1}{L}\int_{t_q}^{t_q+L}\sum_{j=1}^{e(t)} Q_j(t)dt$ represents the average length of all active deques at any time in quantum q, which reflects the potential parallelism of the job not explored in the quantum.

The processor desire for the job in next quantum $q+1$ is then calculated based on both $X_j(t)$ and $Q_j(t)$ as follows:

$$d(q+1) = \frac{1}{L}\int_{t_q}^{t_q+L}\left(\sum_{j=1}^{a(q)} X_j(t) + \beta\sum_{j=1}^{e(t)} Q_j(t)\right)dt, \tag{1}$$

where $\beta \geq 1$ is the exploration parameter that controls how aggressively the scheduler exploits the job's parallelism.

For instance, suppose a processor j is busy working at time t and has one more ready thread on its current deque, that is, $X_j(t) = Q_j(t) = 1$. From this deque's perspective, an extra processor would be able to steal its ready thread, thus explores the available parallelism of the job. Setting $\beta = 1$ will satisfy this requirement. However, since the ready thread is in higher level of the job's structure, it is more likely to spawn more threads in the future. Thus, having a larger value for β, such as setting $\beta = 2$, will further explore the unexposed parallelism of the job. To explore the entire parallelism of the job and to smooth out the processor desire, this calculation is taken from all processors and deques, and is averaged

```
for (i = 0; i < N; i++) {
   Spawn Work(i);
}
sync;
```

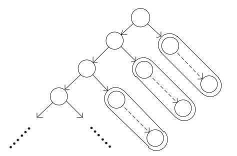


Figure 3: A simple data-parallel program written in Cilk and its DAG representation.

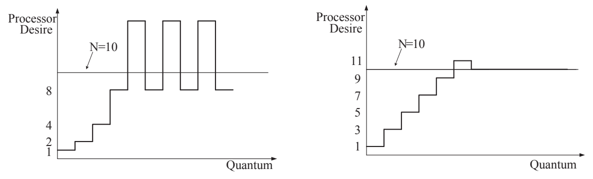


Figure 4: Processor desires calculated by (a) A-Steal and (b) SAWS, when the parallelism of the job is constant at $N = 10$.

over the entire quantum as shown in Eq (1). In case that no more thread is spawned by the extra processors, the processor desire will be reduced to the number of busy processors in the following quantum.

3.2 An Existing Algorithm: A-Steal

We now describe an existing adaptive work stealing algorithm, called A-Steal [14], which calculates the processor desire for a job in each quantum based on only the utilization of the job's allocated processors in the previous quantum. The calculation uses a simple multiplicative-increase multiplicative-decrease strategy first introduced in [7].

Recall that $X_j(t)$ denotes the status of the jth processor at time t, where $1 \leq j \leq a(q)$. The usage of the allocated processors in quantum q is then given by $w(q) = \int_{t_q}^{t_q+L} \sum_{j=1}^{a(q)} X_j(t)dt$. Since maximum possible usage of the quantum is $a(q)L$, the utilization of the processors is $u(q) = w(q)/(a(q)L)$. The quantum is said to be "efficient" if the utilization satisfies $u(q) \geq \delta$, where δ is a threshold usually set in the range of 80% to 95%. Otherwise, the quantum is said to be "inefficient". In addition, the quantum is said to be "satisfied" if we have $a(q) \geq d(q)$. Otherwise, the quantum is "deprived". The processor desire for the job in next quantum $q + 1$ is calculated depending on whether quantum q is efficient or inefficient and whether it is satisfied or deprived as follows:

$$d(q+1) = \begin{cases} d(q) \cdot \rho & \text{if } q \text{ is efficient and satisfied,} \\ d(q)/\rho & \text{if } q \text{ is inefficient,} \\ d(q) & \text{if } q \text{ is efficient and deprived,} \end{cases}$$

where ρ is a responsiveness parameter usually set in the range of 1 to 3. In both SAWS and A-Steal, the processor desire for the first quantum is fixed to be 1, since the job usually starts with a single thread.

Note that A-Steal also actively explores the potential parallelism of the job by increasing its processor desire by a multiplicative factor ρ each time. Since such calculation is blind to the actual parallelism of the job, it can result in desire instability as we will show in the next subsection. SAWS, on the other hand, performs such exploration with more precision and stability, as it directly makes use of the information about the length of active deques, which is a strong indicator on job's actual parallelism.

3.3 Desire Stability of SAWS and A-Steal

It was shown in [16, 10] that another adaptive scheduler based on centralized work sharing, called A-Greedy [7], exhibits desire instability problem, even when the parallelism of the job is constant. Since both A-Steal and A-Greedy use multiplicative-increase multiplicative-decrease strategy to calculate processor desires, such instability problem can also be observed in A-Steal. In this section, we use a simple data-parallel program to demonstrate the desire instability of A-Steal, and to compare it with SAWS.

Suppose that we have a data-parallel application written in Cilk [4] as shown in Fig.3, where N children threads are spawned by the parent thread at almost the same time[2], and each child contains a large amount of work to be done in the

[2]The N threads are spawned with a small delay after each iteration of the *for* loop. Compared to the large amount of time to complete the function *Work*(), however, such delay is negligible.

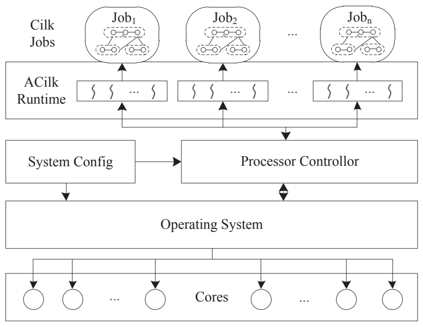


Figure 5: The adaptive scheduling framework of ACilk.

Work() function. The graph at the right of Fig.3 shows the DAG (Directed Acyclic Graph) that represents the structure of the program. The parallelism of this application is therefore constant at N for a long period of time.

To schedule this application with SAWS or A-Steal, we make the following two assumptions. First, we assume that the desires of the job can be satisfied by the global-level processor controller as much as possible. This corresponds to light to medium workloads, in which two-level adaptive schedulers tend to work better compared to non-adaptive schemes [17, 10]. Second, we assume that the ready threads of a deque can be stolen as quickly as possible by steal attempts from other processors. Since the victims are chosen uniformly at random, this is usually true for a reasonable number of processors and when the quantum is set to be sufficiently long. Given these two assumptions, the processor desire and hence the processor allocation of the job can be shown to exhibit unstable behavior as shown in Fig.4(a), where the responsiveness parameter of A-Steal is set to be $\rho = 2$, the utilization threshold is set to $\delta = 0.8$, and the parallelism of the job is at $N = 10$.

Although Fig.3 only gives a simple example, it is not hard to see that such instability problem of A-Steal will remain in many other data-parallel programs like this. Varying parameters ρ and δ can alleviate the problem for a specific parallelism. However, it will inevitably affect the responsiveness of the desires or the utilization of the processors for other sections of the job with different parallelism.

Fig.4(b), on the other hand, shows the processor desires calculated by SAWS for the same application when its exploration parameter is set to be $\beta = 2$. Compared to A-Steal, which catches up with the job's parallelism in about $\log_\rho N$ steps, but never converges to N, SAWS converges to the target parallelism in about N/β steps, and exhibits no desire oscillation afterwards. With comparable values in ρ and β, A-Steal tends to have better convergence for large N initially, but its desire instability will delay job execution and cause resource waste for the majority of time steps in the steady state. SAWS, on the other hand, is more conservative in estimating the processor desires, but guarantees stability, no steady-state error and as shown in Fig.4(b) a small amount of transient overshoot[3]. These properties not only ensure more efficient job execution and resource utilization, but also help to reduce scheduling overheads in practice caused by context switching and cache reloading when adjusting processor allocations for a job [16, 10].

4 Implementations

In this section, we present an adaptive scheduling framework called ACilk, which provides the ability to feedback the processor desires and to support dynamical processor reallocation in runtime. We also present more efficient implementations of the desire calculation algorithms based on sampling methods that approximate the required statistics.

4.1 The framework of ACilk

To implement the feedback-driven scheduling algorithms, we build an adaptive scheduling framework ACilk (Adaptive Cilk), as shown in Fig. 5, which is an extension to the Cilk runtime system [4]. Cilk is a language for multithreaded parallel programming based on ANSI C and it employs the work-stealing scheduler in its runtime system. Based on POSIX threads library, ACilk is built on top of operating systems, such as Linux, and includes three main components: System Config, Processor Controller, and ACilk Runtime, as shown in Fig. 5. The System Config component provides the ability to collect the hardware information, such as available cores in system, and to specify user-oriented configuration information such as scheduling quantum, scheduling algorithms to be used by ACilk. The obvious benefit provided by System Config is to enhance the system scalability and to eliminate the drawback of explicitly specifying the number

[3]The desire overshoot is because of the parent thread that continues after the *for* loop, but immediately blocks when executing the *sync* statement. Since SAWS does not have advanced information about the program structure, it requests for more processors to explore the potential parallelism. The extra processor is immediately released in the next quantum when the parent blocks and no longer spawns more threads.

of cores by users. The Processor Controller is located in the center of ACilk, which provides two main functions: 1) coordination with runtime systems by employing a request-allocation protocol [9] to control processor allocations and feedback processor desires of jobs; 2) reallocation of processor resources among running jobs, which currently supports two processor reallocation strategies, namely, EQUI and DEQ. The Processor Controller is implemented as a daemon process on operating systems, and the Shared Memory technique is used as the Inter-Process Communication (IPC) between the controller and ACilk runtime systems. ACilk Runtime extends the original Cilk runtime system but has the following major improvements: 1) supporting dynamic readjustment of processor allocations without interrupting the execution of the jobs; 2) providing an efficient approximating method to collect and feedback processor desires of jobs at runtime. The more detailed implementation information of ACilk is given in the following subsections.

4.2 Sampling Methods for Desire Calculation

As described in Section 3, the desire calculation algorithms in SAWS and A-Steal require utilization information of the active workers in a quantum, and SAWS also needs the length of its active deques at any time during a quantum. Gathering these information can be very expensive in practice, which will incur a large amount of overhead in the implementation. In this subsection, we will present a more efficient implementation of the algorithms based on sampling methods that approximate the required statistics.

4.2.1 Approximating Processor Utilization

To approximate the processor utilization in a quantum, we adopt the technique used in [19], which takes the ratio between the total number of purely unsuccessful steal attempts and the total number of all steal attempts. Specifically, for each job in quantum q, let $total\_steal_j$ denote the total number of steal attempts by the jth allocated processor or active worker, where $1 \leq j \leq a(q)$. Among all steal attempts, let $purely\_unsucc\_steal_j$ denote the total number of purely unsuccessful steal attempts. A steal attempt is called purely unsuccessful if the victim itself is attempting to steal work from other processors. The processor utilization $u(q)$ of the job in quantum q can then be approximated by

$$u(q) = 1 - \frac{\sum_{j=1}^{a(q)} purely\_unsucc\_steal_j}{\sum_{j=1}^{a(q)} total\_steal_j},$$

which can be used to calculate the processor desires of A-Steal. As shown in Section 3.2, we can also get $\frac{1}{L}\int_{t_q}^{t_q+L}\sum_{j=1}^{a(q)} X_j(t)dt = u(q)a(q)$, which can be used to calculate the processor desires of SAWS in Eq (1).

The intuition for the above approximation of the processor utilization is the following. Since a processor at any time is either working, mugging or stealing and the victim is chosen uniformly at random, the ratio between the number of purely unsuccessful steal attempts and the total number of all steal attempts gives a reasonable approximation for the inefficiency, that is $1 - u(q)$, of the processors in quantum q. From a sampling perspective, the approximation is more accurate if there is a larger number of steal attempts. Furthermore, since the work-stealing scheduler of Cilk runtime already has built-in counters to measure the steal attempts, collecting these information would incur very little extra overhead.

4.2.2 Approximating Active Deques Length

To approximate the length of active deques in a quantum to be used in the desire calculation of SAWS, we again use the technique for approximating processor utilization, but combine it with the length of the deques sampled at the end of the quantum for better accuracy.

We introduce a new counter in ACilk to accumulate the length of the victims' deques at every steal attempt for each active worker j, and denote the accumulated length at the end of quantum q by $length_j$. The approximated length of all active deques is then given by

$$Q(q) = e(t_q + L)\frac{\sum_{j=1}^{a(q)} length_j}{\sum_{j=1}^{a(q)} total\_steal_j},$$

where $e(t_q + L)$ denotes the number of active deques when quantum q ends at time $t_q + L$. Since each steal attempt will collect the deque length of the victim processor, the ratio between the total accumulated deque length from all active workers and the total number of steal attempts intuitively gives the average length of any single deque in the quantum. Multiplying this ratio by the number of active deques then gives a natural approximation for the total length of all active deques.

In addition, we use the length of the deques sampled at the end of the quantum as another approximation, and it is given by $Q'(q) = \sum_{j=1}^{e(t_q+L)} Q_j(t_q + L)$, where $Q_j(t_q + L)$ denotes the length of the jth deque of the job at time $t_q + L$ when quantum q ends.

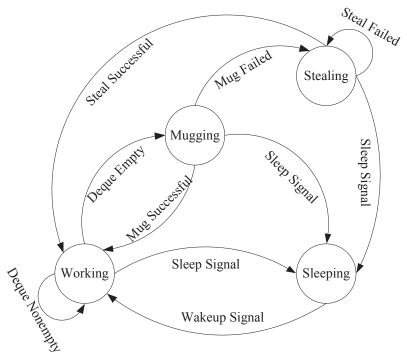


Figure 6: State diagram of a worker's execution in adaptive work stealing.

The final approximation of the length of active deques then takes a linear combination of the two approximations and is given by $\frac{1}{L}\int_{t_q}^{t_q+L}\sum_{j=1}^{e(t)} Q_j(t)dt = \alpha Q(q) + (1-\alpha)Q'(q)$, which can be used to complete the desire calculation of SAWS shown in Eq (1). Intuitively, the first approximation is more accurate when there are more samples in steal attempts, thus should have higher weight. In our implementation, we set α to be the ratio between the total number of steal attempts in the quantum and the maximum possible steal attempts. Hence, the second approximation is always used in the calculation, and when no steal attempt occurs in the quantum, the first approximation is simply ignored.

4.3 Processor Reallocation

For implementation of adaptive work stealing, one of important challenges in ACilk is how to deal with the dynamic processor reallocations in runtime without interrupting the execution of jobs. To support adaptive work stealing, ACilk introduces four different states, namely working, stealing, mugging, and sleeping to each processor or a worker in Cilk runtime system. ACilk ensures that the number of active workers used by a job always matches the number of physical processors assigned to it by controlling the state of the workers. The detailed process is depicted in Fig.6. At the initialization stage, ACilk creates as many workers as the total number of physical processors for each job. After getting its first processor allocation (which is usually 1), ACilk puts the extra workers into the sleeping state. After each scheduling quantum, whenever the allocation of the job increases, some workers of the job are waken up. When the allotment decreases, the corresponding number of workers are put into sleeping state. Unlike the original work-stealing mechanism, whenever a worker runs out work in ACilk, that is, its local deque becomes empty, it first enters the mugging state to look for muggable deques instead of immediately stealing work from another worker.

In the Processor Controller, two different resource allocation strategies EQUI and DEQ are implemented. EQUI (Equi-partitioning) [6] is one of the well-known resource allocation strategies, which at any time divides the total number of all available cores evenly among all running jobs. Obviously, only when a new job is released or when a job completes, EQUI starts to readjust the processor allocation among the running jobs, and any feedback from the jobs is not considered in this algorithm.

DEQ [18] is a variant of EQUI, which can take advantage of the parallelism feedbacks. Compared with EQUI, DEQ never allocates more processors to a job than the job's processor desire, hence it is better known for its efficiency and fairness in processor allocation [15]. Let $\mathcal{J}(q)$ denote the set of active jobs when a new quantum q begins. Based on the processor desires of all jobs collected by ACilk runtime, DEQ allocates the processors as shown in Algorithm 1, where $a_i(q)$ and $d_i(q)$ denote the processor allocation and the processor desire of job J_i in quantum q respectively, and P denotes the total number of available cores in the system.

5 Experiments

The experiments are carried out on a Sun Fire X4600 M2 server which is equipped with eight AMD Opteron(TM) 8384 quad-core processors, each with 2.7 GHz clock speed, 128 KB L1 cache, 512 KB L2 cache per core, 6 MB L3 cache, and 256GB main memory. The operating system is Ubuntu 9.10 (Linux kernel 2.6.28), and the compiler is GCC 4.4.1, with the compiling option "-g -O2". Six computation-intensive benchmarks are selected from the official released Cilk-5.4.6 for the experiments. The brief description and input sets of these benchmarks are listed in Table 1.

To compare the performances of different scheduling algorithms, we use the following metrics: makespan, mean response time, and processor utilization. The makespan is defined as the completion time of the last completed job in the job set. The response time of a single job is the time elapsed from when the job arrives to when it completes, and the mean response time of the job set is used in our experiments. The utilization of the job's allocated processors is collected

Algorithm 1 $DEQ(\mathcal{J}(q), P)$

```
1:  if $\mathcal{J}(q) = \emptyset$ then
2:      return
3:  $S = \{\mathcal{J}_i \in \mathcal{J}(q) : d_i(q) \leq P/|\mathcal{J}(q)|\}$
4:  if $S = \emptyset$ then
5:      for each $J_i \in \mathcal{J}(q)$ do
6:          $a_i(q) = P/|\mathcal{J}(q)|$
7:      return
8:  else
9:      for each $J_i \in S$ do
10:         $a_i(q) = d_i(q)$
11:     $DEQ(\mathcal{J}(q) - S, P - \sum_{J_i \in S} a_i(q))$
```

Table 1: The description and input sets of the benchmarks

Benchmark	Description	Input sets
CK	Rudimentary checkers	-b 10 -w 13
Fib	Fibonacci numbers	46
FFT	Fast Fourier Transform	-n 2^{26}
LU	LU decomposition	-n 4096
Heat	Jacobi-type iteration to Solve a finite-difference	-g 10 -nx 4096 -ny 4096 -nt 500
Strassen	Multiplies two n $\times$ n matrices	-n 4096

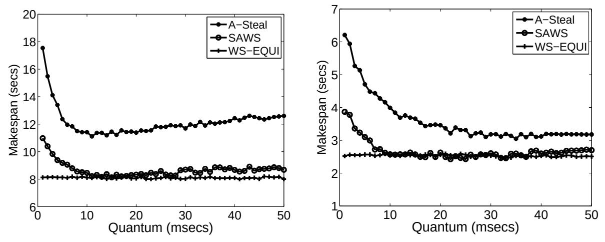
by counting the time of each processor during a quantum when the processor is doing useful work. Note that the time a processor spends on stealing is considered as wasted, because although the processor is not idle during stealing, it is not contributing towards the work of the job. In the experiments, the responsiveness parameter ρ and the utilization threshold δ of A-Steal are set to be 2 and 80% respectively, and the exploration parameter β of SAWS is set to be 2.

As pointed out in previous sections, the WS (Work Stealing) algorithm implemented by original Cilk runtime system does not support dynamically readjusting the jobs' processor allocations at runtime. Therefore, manually specifying a fixed number of processors may easily lead to degraded performance when concurrently running multiple Cilk jobs. In our experiment, we implement WS as a two-level scheduling algorithm by combining it with algorithm EQUI and name the new algorithm WS-EQUI. Compared with WS, WS-EQUI does not need explicit specification on the number of cores by each user and it can automatically share all available cores among the running jobs equally. Since WS-EQUI can be considered as a special type of two-level adaptive scheduler with variable quantum length (a quantum only expires if a job completes or a new job is released) and an oblivious parallelism feedbacks (which always divides the processors equally among the active jobs regardless of each job's processor desire), we use it as a reference to evaluate the performances of the feedback-driven scheduling algorithms, such as A-Steal and SAWS in the following experiments.

5.1 Scheduling Quantum and Overhead

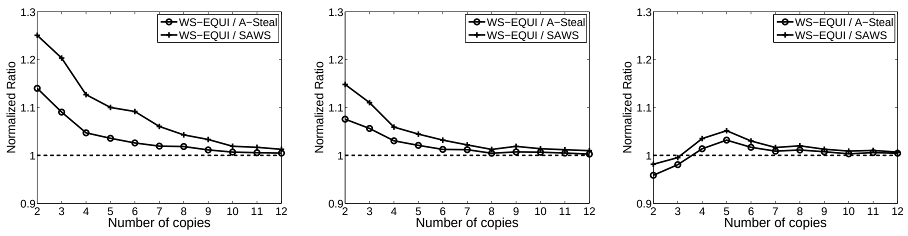
In adaptive scheduling, the length of the scheduling quantum is an important system parameter, which may significantly affect the performance of a scheduling algorithm. Intuitively, smaller quantum length may lead to more efficiency for capturing changes in a job's parallelism, but it inevitably incurs more scheduling overhead, including the cost of processor reallocation. In this subsection, we conduct a set of experiments to examine the impact of scheduling quantum and corresponding overhead on the performances of different scheduling algorithms. Specifically, only one job in Table 1 is used for each experiment. The quantum length is varied from 1ms to 50ms. The experimental results for Strassen and LU are shown in Fig.7 while the other benchmarks have similar results and are omitted.

The results demonstrate that, compared with WS-EQUI, the performance of A-Steal and SAWS are impacted by varying scheduling quantum, especially that of A-Steal due to its unstable parallelism feedbacks. As can be seen in Fig. 7, the makespans of A-Steal and SAWS are much worse when the quantum length is set to 1ms, as the overhead incurred by the feedback-driven algorithms is too large to be ignored in this case. With increasing quantum length, however, the makespans of A-Steal and SAWS become smaller and tend to get closer to that of WS-EQUI since the scheduling overhead is now better amortized over the entire scheduling quantum. Based on the experimental results, the length of the scheduling quantum is set to be 10ms in all following experiments, which seems to provide a good tradeoff between the responsiveness and the scheduling overhead.



(a) Strassen (b) LU

Figure 7: Impact of scheduling quantum and corresponding overhead on the performances of different scheduling algorithms



(a) CK (b) Strassen (c) LU

Figure 8: Performance comparison of different algorithms using different types of batched jobs.

5.2 Performance Comparison of Different algorithms

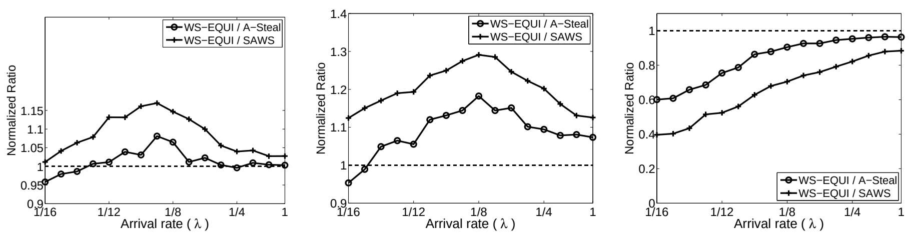
In this subsection, we evaluate and compare the performances of different scheduling algorithms using two sets of experiments. The first set uses different types of batched workloads, where each type of workload is represented by a particular job shown in Table 1 and several copies of the same job are released simultaneously. This corresponds to typical burst arrivals of jobs with the same characteristics. The system load is set to be proportional to the number of simultaneously submitted copies, which is varied from 2 to 12. The second set uses the general non-batched workloads, where jobs are randomly chosen from Table 1, and they are released into the system according to the Poisson process, where the inter-arrival time follows exponential distribution $f(t; \lambda) = \lambda e^{-\lambda t}(t \geq 0)$. The system load is proportional to the arrival rate λ of the jobs, which is varied from 1/16 to 1, and the total number of jobs is fixed to be 16.

Batched jobs The experimental results, as shown in Fig.8, demonstrate that the feedback-driven adaptive scheduling algorithms A-Steal and SAWS generally outperform WS-EQUI with respect to Makespan. In addition, better performance tend to be achieved when the jobs have lower parallelism such as CK and Strassen, as shown in Fig.1(a). The reason is that feedback-driven scheduling strategies take advantage of the parallelism feedback based on the information of execution history and thus can more precisely captures the running characteristics of jobs, while WS-EQUI is oblivious to the job's parallelism and thus wastes many processor resources. On the other hand, as shown in Fig.8(c), when the jobs have sufficient parallelism, such as LU, the performances of all algorithms tend to eventually converge to each other, since the jobs can efficiently utilize all available cores regardless of the algorithm. Nevertheless, it shows that feedback-driven adaptive schedulers have more advantages than WS-EQUI, particularly when the runtime characteristics of the jobs are unknown in advance.

Non-batched jobs The non-batched experiments represent more realistic scenarios when running parallel jobs in practice. The results as shown in Fig.9 suggest that SAWS generally achieves better performance than A-Steal and WS-EQUI with respect to makespan, mean response time, and utilization. Specifically, the mean response time improvements of SAWS over A-Steal and WS-EQUI are 12.43% and 21.32% respectively and the corresponding utilization improvements are 25.78% and 46.98% respectively. The makespan improvements of SAWS, however, seem small, which are only 3.02% and 7.18%, as it could be easily dominated by one large job in the job set with long execution time. The main advantage of SAWS is that it directly benefits from its more accurate and stable parallelism feedbacks, as described in Section 3.3. The experimental results also show that WS-EQUI seems to have better performance when the system load is light, but it is at the cost of wasting more processor resources, as clearly shown in Fig.9(c).

6 Conclusion

In this paper, we studied feedback-driven adaptive scheduling based on a work-stealing load-balancing strategy, which provides an efficient solution to better utilize the available processor resources and to improve efficiency when concur-



(a) Makespan (b) Mean response time (c) Utilization

Figure 9: Performance comparison of different algorithms using nonbatched jobs.

rently executing parallel applications on many-core platforms. The benefit of adaptive scheduling is reflected not only in eliminating the need of manually specifying the number of cores required by most existing many-core runtime systems, but also in enhancing the overall system performance by exploiting the runtime characteristics of individual parallel applications. The experimental results demonstrated that feedback-driven adaptive scheduling algorithms achieve better performance with respect to makespan, mean response time and processor utilization, especially when more accurate and stable feedback mechanism is applied. For our future work, we plan to integrate our adaptive scheduling algorithm into the Linux kernel, which will provide more benefits for efficiently controlling and collaborating with many-core runtime systems.

Acknowledgment

This work is supported in part by Sino-Italian Joint Research Program under the grant No. 2009DFA12110, Natural Science Foundation of China under the grant No. 61073011, 61133004, 61173039 and China National Hi-tech Research and Development Program (863 Program) under the grants No. 2008AA01A202, 2011AA01A204.

References

[1] S. Borkar, and A. A Chien. The future of microprocessors. *Communications of the ACM*, 2011, 54(5):67–77.

[2] B. Chapman, and L. Huang. Enhancing OpenMP and its implementation for programming multicore systems. Parallel computing: architectures, algorithms, and applications. IOS Press Inc, Amsterdam, 2008.

[3] J. Reinders. Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. Sebastopol, CA: O'Reilly Media, 2007.

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

[5] Intel Cilk Plus http://software.intel.com/en-us/articles/intel-cilk-plus/

[6] J. Edmonds. Scheduling in the dark (improved result). *Theoretical Computer Science*, 2007, 235(1):109–141.

[7] K. Agrawal, Y. He, W.-J. Hsu, and C. E. Leiserson. Adaptive scheduling with parallelism feedback. In *PPoPP*, 2006.

[8] K. Agrawal, Y. He, and C. E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *ICDCS*, 2006.

[9] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient online non-clairvoyant adaptive scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2008, 19(9): 1263–1279.

[10] H. Sun, Y. Cao, and W.-J. Hsu. Efficient adaptive scheduling of multiprocessors with stable parallelism feedback. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):594–607, 2011.

[11] Y. Cao, H. Sun, W.-J. Hsu and D. Qian. Malleable-Lab: A tool for evaluating adaptive online schedulers on malleable jobs. In *PDP*, 2010.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[13] A. Duran, J. Corbal*á*n, and E. Ayguad*é*. Evaluation of OpenMP task scheduling strategies, *OpenMP in a New Era of Parallelism*, 5004:100–110, 2008.

[14] K. Agrawal, Y. He, and C. E. Leiserson. Adaptive work stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 2008, 26(3):1–32.

[15] Y. He, W.-J. Hsu, and C. E. Leiserson. Provably efficient two-level adaptive scheduling. In *JSSPP*, 2006.

[16] H. Sun and W.-J. Hsu. Adaptive B-Greedy (ABG): A simple yet efficient scheduling algorithm. In *IPDPS*, 2008.

[17] H. Sun, Y. Cao, and W.-J. Hsu. Competitive two-level adaptive scheduling using resource augmentation. In *JSSPP*, 2009.

[18] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

[19] S. Sen. Dynamic processor allocation for adaptively parallel jobs. Master's thesis, Massachusetts Institute of technology, 2004.