Int J Parallel Prog (2018) 46:173–197
https://doi.org/10.1007/s10766-016-0484-8

Scheduling Parallel Computations by Work Stealing: A Survey

Jixiang Yang[1] · **Qingbi He**[2]

Received: 2 September 2016 / Accepted: 29 December 2016 / Published online: 6 January 2017

Abstract Work stealing has been proven to be an efficient technique for scheduling parallel computations, and has been gaining popularity as the multiprocessor/multicore-processor load balancing technology of choice in both industry and academia. A review on the work stealing scheduling is provided from the perspective of scheduling algorithms, optimization of algorithm implementation and processor architecture oriented optimization. The future research trends and recommendations driven by theory, emerging applications and motifs, architecture and heterogeneous platforms are also provided.

Keywords Work stealing · Load balancing · Task parallelism · Task scheduling · Optimization · Multicore/manycore

1 Introduction

For many decades, Moore's law has bestowed a wealth of transistors that hardware designers and compiler writers have converted to usable performance, without changing the sequential programming interface. The main techniques for these performance benefits—increased clock frequency and smarter but increasingly complex architectures—are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster.

✉ Jixiang Yang
jixiang_yang@126.com

1 School of Mathematics and Statistics, Chongqing Jiaotong University, Chongqing 400074, China

2 School of Information Science and Engineering, Chongqing Jiaotong University, Chongqing 400074, China

This historic shift to multicore processors changes the programming interface by exposing parallelism to the programmer, after decades of sequential computing.

The emergence of multicore chips makes computer programming model facing huge pressure due to the shift from traditional serial programming mode to new parallel programming mode. Before multicore chips appear, with the continuous improvement of the performance of single core processor, serial application execution speed will be accelerated. However, at present, the pursuit of high performance single core processor era has ended and the free lunch is over. The performance of serial applications can only be improved with parallelism, and programmers are on the way to parallel programming.

There are two driving forces for traditional parallel programming models. One is the endless pursuit of performance and problem scale of scientific computing applications (mostly physical simulation), and generally there exists a superior, static computation partitioning method for this type of applications which have obvious data locality and are easy to achieve load balancing. The corresponding languages and interfaces include the MPI on distributed storage system and the OpenMP (before version 2.5) on shared memory system. These language models are mapping methods that allow users to describe computing on a virtual processor set, which can be extended to cluster and traditional shared memory multiprocessor (SMP) machines. But OpenMP (before version 2.5) can't adapt to more extensive irregular applications, MPI allows users to manually handle synchronization and communication between parallel processes. The second driving force is to improve the performance of key non-scientific computing applications via concurrency and the common programming interface is Pthread. Here, users can directly manipulate the creation, synchronization and communication of threads and thus the programming level is lower than others.

It can be seen that traditional parallel programming model target the expert level and senior programmers, or can only adapt to the application of rules. In multicore era, it is paramount to provide parallel programming tools that are easy to program with high development productivity for a wide range of application domains. In recent years, many new parallel programming models have emerged, among which the task-parallel programming model has become the preferred parallel programming model for multicore platform. Note that a task is an abstract terms, which may be an entire program or each successive invocation of a program, more precise alternative term indicates process or thread. The process can be defined as program that is being executed within a computer system, which generally is allowed to be further divided into threads. The threads allowed operate depending on the processes running them. Once the processes are terminated, the threads are also terminated. Runtime system is responsible for task scheduling, each core corresponds to a physical thread, and each physical thread will perform a lot of logical tasks. In order to improve multicore utilization, runtime system uses work stealing scheduling algorithm to achieve load balancing.

One of the core technologies for task-parallel computing is work stealing scheduling [1,2]. Runtime system uses work stealing scheduling algorithm to map and schedule logical task to thread to execute, and thus can achieve load balancing. In general, the implementation of work stealing scheduling algorithm is that each processor core corresponds to a thread, and each thread maintains a double-ended queue (deque) for storing task status information. This status information includes the local variables, PC values, and the number of sub-tasks of this task and so on, and is used to restore suspended tasks or to perform stolen tasks. Each thread pushes the ready tasks onto the bottom of its deque or popups the tasks that have been executed; when the deque is empty, this thread steals tasks from the top of the deque on other threads. It firstly restores the status of the stolen tasks, and then jumps to the instruction after *spawn* or *sync* and starts execution according to the status. Stealing threads by means of deque will not interrupt the execution of work thread and thus can obtain tasks, meanwhile can achieve load balancing.

Work stealing is an algorithm based on directed acyclic graph (DAGs) scheduling, each node in the DAG represents a task and each edge in DAG represents a dependency relation. If all ancestor-nodes' tasks of a task have been completed, then this task is ready. Work stealing scheduling is responsible for allocating the ready task to the processor core to prepare for execution. The execution time of random work stealing scheduling algorithm based on DAG on P processors is $T_P = T_1/P + O(T_\infty)$, where T_1 is the minimum serial execution time of the multithreaded computation and T_∞ is the minimum execution time with an infinite number of processors. Moreover, the space required by the execution is at most $S_1 P$, where S_1 is the minimum serial space requirement [3].

Work stealing scheduling makes it uncertain whether the parent and child task will be executed in parallel. For example, when a thread creates, or spawns a child task B during the course of executing parent task A, the thread pushes A onto the bottom of its deque and then starts to execute the child task B. If the parent task A is stolen by other threads, then the child task B and the parent task A are executed in parallel in this case. If the parent task A isn't stolen by other threads, then the parent task A is popped from the bottom of task deque and the child task B and parent task A are executed serially in this case.

In summary, work stealing has the following advantages. (1) It dynamically schedules tasks according to busy situation of threads and thus achieves load balancing. (2) At the synchronization point, if a task A has to wait for a child task that is not completed, then the thread stalls the task A and executes other tasks. The thread having completed last child task will enable the stalled task A. This kind of data-flow-driven method can efficiently utilize computing resources. (3) Effective combination of recursive parallel with cache oblivious algorithms makes locality-sensitive applications can efficiently utilize cache. For example, the BLAS library uses recursive cache oblivious algorithms. Task parallel programming model provides implicit task mapping mechanism. Runtime system uses work stealing scheduling algorithm to map logical task to physical thread to execute, and this can improve efficiency of execution. A great deal of work related to work stealing can be divided into three stages: theoretical research, optimization of algorithm implementation and processor architecture oriented optimization.

The rest of this paper is organized as follows. The theoretical results of work stealing scheduling are reviewed in Sect. 2, the optimization-related research on the implementation of work stealing scheduling algorithm is reviewed in Sect. 3, and the architecture-oriented optimization research is also reviewed in Sect. 4. Conclusions and recommendations are provided in Sect. 5.

2 Work Stealing Scheduling Algorithms (WS)

The idea of work stealing can be traced back to the work of Burton and Sleep [4] on parallel execution of functional programs, as well as the implementation of Multilisp (a multiprocessor extension of the Lisp programming language) by Halstead [5,6]. When deciding whether to spawn a new task for a function or to execute it inline, it puts potential new task onto the bottom of a task queue and allows other processors to steal from the queue. If the task is not stolen, then it is executed on original machine. In the 1990s and twenty-first century, the work stealing scheduling multithreaded computations for shared memory multiprocessor architecture have achieved breakthrough in all aspects.

2.1 Randomized Work Stealing Algorithm (RWS)

Blumofe and Leiserson [3] presented the first provably good randomized work stealing (RWS) scheduler for multithreaded computations with dependencies; they prove existentially optimal worst-case bounds on expected execution time, space required, and total communication cost, and show that work stealing has much lower communication cost than work sharing. The work stealing algorithm has one deque per processor, and processor is treated as stack but other processors can steal from other end. They assumed that processor will work on its own tasks if possible, but steals from a randomly chosen processor if the deque is empty. They modeled contention by assuming that steal requests are serially queued by a worst-case analysis. They have shown that the expected time of executing a fully-strict computation, which is one where the data dependencies of a worker go to its parent only, with P processors, using their work stealing based scheduler, is given by $t = \frac{t_1}{P+O(t_\infty)}$. Here t_1 denotes the minimum execution time of the computation with a single processor, t_∞ denotes the minimum execution time with infinite number of processors. The expected total communication cost of the algorithm is $P \times t_\infty(1 + N_d) \times S_{\max}$, where N_d is the maximum number of times a thread synchronizes with its parent, and $S_{\max}$ is the size of largest activation record of any thread. The storage space required by the algorithm is given by $s_1 P$, where s_1 is the minimum space required with a single processor. The work stealing algorithm is efficient in terms of time, space and communication. This algorithm is used in Cilk [1]. While these theoretical bounds hold for fully-strict computations, work stealing has also been shown to be efficient for those programs which are not fully-strict [7].

2.2 Variants of RWS

Many variants of randomized work stealing (RWS) are presented in various directions, such as those analyzed by Squillante and Nelson [8], Mitzenmacher [9] and Rudolph et al. [10]. Work stealing has also been investigated in a variety of other contexts, including migration cost [8,11,12], affinity scheduling [13–17], heterogeneous systems [18–20] and other theoretical results [21–29]. See Table 1.

Table 1 Examples of the variants of RWS

Context	Section	References
Two basic models	2.2.1	Squillante et al. [8], Mitzenmacher [9], Rudolph et al. [10]
Migration cost	2.2.2	Squillante et al. [8], Eager et al. [11], Mirchandaney et al. [12]
Affinity scheduling	2.2.3	Squillante and Lazowska [13], Squillante et al. [14], Acar et al. [15], Narang and Shyamasundar [16], Suksompong et al. [17]
Heterogeneous systems	2.2.4	Mirchandaney et al. [18], Bender and Rabin [19], Gast and Bruno [20]
More theoretical results	2.2.5	Blelloch et al. [21], Fatourou and Spirakis [22], Berenbrink et al. [23], Arora et al. [24], Tchiboukdjian et al. [25,26], Agrawal et al. [27], Cole et al. [28], Agrawal et al. [29]

2.2.1 Two Basic Models

Squillante et al. [8] presented a threshold-based queueing model of shared-memory multiprocessor scheduling. They assume identical processors, distributions of arrival rates λ, and distributions of processing time μ. Also, time to probe and migrate and waiting task is exponentially distributed, and processing time that is also exponential distribution but with parameter $\alpha < \mu$ at new processor is high because of affinity. Queueing model results in discrete state space and continuous-time Markov process. The large and complex state space is decomposed by assuming processor states stochastically independent and identical. It is approximate for finite number of processors, so compared with simulation. The general form allows modeling of degradation in system performance due to task migration. Even when migration costs are large, and contention compounds these costs by degrading system performance, task migration may still be beneficial. Threshold policies prevent processor thrashing: instability when tasks passed back and forth and most of system time spent on migration.

Mitzenmacher [9] used differential equations to study work stealing. His method is an approximation for a finite number of processors, though it is exact as the number of processors goes to infinity. The basic model assumes: identical processors, task arrivals Poisson with parameter λ, service time exponential with parameter μ, work stealing algorithm, stealing instantaneous. The extensions considers threshold stealing, preemptive stealing, repeated steal attempts, varying service and arrival distributions, transfer time for tasks, multiple choices, stealing multiple tasks such as in Rudolph et al. [10], varying processor speeds, varying arrival rates.

2.2.2 Migration Cost

Task migrations naturally will incur additional costs or overheads. Squilante et al. [8] argued that the costs of moving a task are very different in shared-memory systems: in a distributed system, costs are incurred by the processor from which the task is migrated, possibly with an additional network delay. For shared-memory systems, an idle processor can search the queues of other processors and remove a task without disturbing the busy processors. The time required for probing and removal of processes is small. The major cost of task migration is a larger service demand at the processor that migrate the task, reflecting the time needed to establish the cache's working set at this processor. Thus the direct costs of migration are shifted from the busy processor to the idle processor, and work-stealing has much greater potential benefits. Secondly, indirect cost of task migration in a shared memory multiprocessor is increased contention for the communication medium and the shared memory itself. In distributed systems, on the other hand, contention for the communication network is unlikely. Finally, shared-memory multiprocessors have no difficulty in migrating a task which has already started execution. This is very different than the results of Eager et al. [11] for the distributed case.

Work stealing contrasts with work sharing, another popular scheduling approach for dynamic multithreading, where each thread is attempted to underutilized processor when it is spawned. Compared to this approach, underutilized processors attempt to take threads from other processors, and migration of threads occurs less frequently with work stealing than work sharing, especially when load is high and no one wants to steal [3]. Eager et al. [11] argued that work-sharing outperforms work-stealing at light to moderate system loads, while work-stealing outperforms work-sharing at high loads, if the costs of task transfer under the two strategies are comparable. However, they argued that costs are likely to be greater under work-stealing, making work-sharing preferable. This is because work-stealing policies must transfer tasks which have already started to execute, while work-sharing policies can transfer prior to beginning execution. Mirchandaney et al. [12] have similar results, assuming that the cost of task transfer results from network delays. Analysis techniques for both: decomposition of Markov chain, matrix-geometric approach, validation by simulations. They all neglect overhead due to probing.

2.2.3 Affinity Scheduling Algorithms

Affinity scheduling is the allocation, or scheduling, of computing tasks on the computing nodes where they will be executed more efficiently. Such affinity of a task for a node can be based on any aspects of the computing node or computing task that make execution more efficient, and it is most often related to different speeds or overheads associated with the resources of the computing node that are required by the computing task.

In a shared-memory multiprocessor system, it may be more efficient to schedule a task on one processor than another. One reason for this is that the processors, or their associated resources, may be heterogeneous. Squillante and Lazowska [13], Squillante et al. [14], and Acar et al. [15] considered this case in more detail. Another reason is processor-cache affinity: when a task returns for execution and is scheduled on a processor, it experiences an initial burst of cache misses. However, if a significant portion of the task's working set is already in the cache, this penalty is reduced. Thus they have a tradeoff between load balancing and using locality.

Squillante and Lazowska [13] proposed and compared various scheduling algorithms which trade off load balancing and processor-cache affinity. They used a detailed cache model to examine cache reload time, as well as examining the effects of increased bus traffic. Analysis techniques include a combination of mean value analysis, bounding approximations, and simulation. A shared pool of tasks is assumed: idle processors search this pool and choose a task associated with that processor if possible. This outperforms a simple FIFO queue (good load balancing, bad locality) or a "fixed processor" model where each processor has its own queue and tasks are not shared (good locality, bad load balancing). No work stealing since pool of tasks is shared.

Squillante et al. [14] assumed a generic form of affinity in which the service rates μ_{ij} are higher for jobs in a processors own queue ($i = j$). However, some queues have higher priorities c_j than others, so processor i will choose the job with highest $u_{ij}c_j$. Also, we can assume that some processors are not allowed to process some queues by setting $u_{ij} = 0$. They showed that a threshold-based priority algorithm works well: a threshold T_j is set for each queue j, and queues with a number of tasks exceeding this threshold are given priority. They presented an algorithm which determines where the thresholds should be set. Analysis techniques involve queueing theory to give approximate results for the two-queue two-server case, and simulations for all cases.

Acar et al. [15] presented a work-stealing algorithm that uses locality information, and thus outperforms the standard work-stealing algorithm on benchmarks. Each process maintains a queue of pointers to threads that have affinity for it, and attempts to steal these first. They also bounded the number of cache misses for the work stealing algorithm, using a "potential function" argument.

Naranga et al. [16] addressed affinity driven distributed scheduling for hybrid parallel computations which contain tasks that have pre-specified affinity to a place and also tasks that can be mapped to any place in the system. Specifically, they addressed two scheduling problems of the type $P_m|M_j, \text{prec}|C_{\max}$. They presented online distributed scheduling algorithms for hybrid parallel computations assuming both unconstrained and bounded space per place. They also presented the time and message complexity for distributed scheduling of hybrid computations. This is the first time that distributed scheduling algorithms for hybrid parallel computations have been presented and analyzed for time and message bounds under both unconstrained space and bounded space.

Suksompong et al. [17] investigated a variant of the work-stealing algorithm that they call the localized work-stealing algorithm. They showed that the expected running time of the algorithm is $T_1/P + O(T_\infty P)$, and that under the even distribution of free agents assumption, the expected running time of the algorithm is $T_1/P + O(T_\infty \lg P)$. In addition, they obtained another running-time bound based on ratios between the sizes of serial tasks in the computation. If M denotes the maximum ratio between the largest and the smallest serial tasks of a processor after removing a total of $O(P)$ serial tasks across all processors from consideration, then the expected running time of the algorithm is $T_1/P + O(T_\infty M)$.

2.2.4 Heterogeneous Systems

There has been a recent increase of interest in heterogeneous computing systems, due partly to the fact that a single parallel architecture may not be adequate for exploiting all of a programs available parallelism. Mirchandaney et al. [18] presented an adaptive algorithm in heterogeneous systems. Type 1 systems have processors identical with respect to processing capabilities and speeds, but different arrival rates λ. Type 2 systems may also have different processing rates μ. As in their earlier paper, they assumed that job transfers encounter significant delays due to network processing at source and destination, and transmission time. Again, load-sharing and load-stealing algorithms are considered. Analysis techniques include Markove chain and decomposition into simpler chains then exact solution using matrix-geometric techniques. Models are validated with simulations. The most interesting idea is biased probing. Biased probing can result in significant performance gains for work-stealing though not for work-sharing. Bender and Rabin [19] focused on the case where processors have different speeds, each processor maintains an estimate of its own speed, and communication between processors has a cost. They proposed a version of the work stealing algorithm where a faster processor can interrupt a slower processor and steal its current task, and they bounded the execution time of this algorithm.

Gast and Bruno [20] presented a Markovian model for performance evaluation of work stealing in large-scale heterogeneous systems. Using mean field theory, they showed that when the size of the system grows, it converges to a system of deterministic ordinary differential equations that allows one to compute the expectation of performance functions (such as average response times) as well as the distributions of these functions. They first studied the case where all resources are homogeneous, showing in particular that work stealing is very efficient, even when the latency of steals is large. They also considered the case where distance plays a role: the system is made of several clusters, and stealing within one cluster is faster than stealing between clusters. They compared different work stealing policies, based on stealing probabilities and showed that the main factor for deciding where to steal from is the load rather than the stealing latency.

2.2.5 More Theoretical Results

More theoretical results are presented from different perspectives, such as space bounds [21], strict multi-threaded computations [22], and stability [23]. Work stealing has also been investigated in a variety of other contexts, including applications to thread scheduling [24], list scheduling [25,26], Fork–Join parallel programming [27], false sharing [28] and parallel batched data structures [29]. See Table 2.

Blelloch et al. [21] improved the space bounds of Blumofe et al. [3] for a global shared-memory multiprocessor system. Fatourou et al. [22] extended the Blumofe et al. [3] model to strict multi-threaded computations (thread dependent on ancestor, not just parent). Berenbrink et al. [23] showed that the work stealing algorithm is stable even under a very unbalanced distribution of loads.

Arora et al. [24] assumed that the scheduler maps threads onto processes while an adversarial kernel maps processes to processors, and bounded the expected execution time. Tchiboukdjian et al. [25] presented a complete analysis of the cost of distribution in list scheduling. They proposed a new framework, based on potential functions, for analyzing the complexity of distributed list scheduling algorithms. In all variants of the problem, they succeeded to characterize precisely the overhead due to the decentralization of the list. In particular, in the case of independent tasks, the overhead

Table 2 Examples of other theoretical results

Context	References
Space bounds	Blelloch et al. [21]
Strict multi-threaded computations	Fatourou and Spirakis [22]
Stability	Berenbrink et al. [23]
Applications to thread scheduling	Arora et al. [24]
List scheduling	Tchiboukdjian et al. [25,26]
Fork–Join parallel programming	Agrawal et al. [27]
False sharing	Cole and Ramachandran [28]
Parallel batched data structures	Agrawal et al. [29]

due to the distribution is small and only depends on the number of tasks and not on their weights. In addition, this analysis improves the bounds for the classical work stealing algorithm of Arora et al. [24] from 32 to 5.5 D. The work helps to clarify the links between classical list scheduling and work stealing. Furthermore, the framework to analyze DLS algorithms described in this paper is more general than the method of Arora et al. [24]. Furthermore, Tchiboukdjian et al. [26] derived a bound on the deviation from the mean and applied this technique to show that the expected makespan for scheduling W unit independent tasks on m processors is equal to *W/m* with an additional term in 3.65 $\log_2 W$. Indeed, the work does not assume a specific rule to manage the local lists. Moreover, the work doesn't refer to the structure of the DAG but on the work contained in each list, and thus this analysis can be extended to the case of general precedence graphs.

Agrawal et al. [27] provided theoretical completion-time and space-usage bounds for a design of HELPER based on work stealing. Their theoretical work extends the results given in [3,24] for work stealing schedulers, showing that for a computation E, HELPER completes E on P processors in expected time $O(T_1/P + T_\infty + PV)$, where T_1 is the work of E, T_∞ is E's "aggregate span" which is bounded by the sum of spans (critical-path lengths) of all regions, and V is the number of parallel regions in E. Their completion-time bounds are asymptotically optimal for certain computations with parallel regions and helper locks. In addition, the bounds imply that HELPER produces linear speedup provided that all parallel regions in the computation are sufficiently parallel. Roughly, if for every region A, the nonnested work of region A is asymptotically larger than P times the span of A, then HELPER executes the computation with speedup approaching P. They also showed that HELPER completes E using only $O(PS_1)$ stack space, where S_1 is the sum over all regions A of the stack space used by A in a serial execution of the same computation E.

Cole et al. [28] analyzed the overhead due to false sharing when parallel tasks are scheduled using randomized work stealing (RWS) [3]. They obtained high-probability bounds on the cache miss overhead, including the overhead due to false sharing, for several parallel cache-efficient algorithms when scheduled using RWS. These include algorithms for fundamental problems, such as matrix computations, FFT, sorting, basic dynamic programming, list ranking and graph connected components. Their

Table 3 Examples of optimization of algorithm implementation

Context	Section	References
Task granularity	3.1	Sanchez et al. [30], Kulkarni et al. [31], Hill and Marty [32], Chen et al. [33], Faxén [34]
	3.1.1	Mohr et al. [35], Loidl et al. [36], Duran et al. [37], Cong et al. [38], Duran et al. [39], Acar et al. [40]
	3.1.2	Wang et al. [41], Tzannes et al. [42], Cao et al. [43], Hoffmann and Rauber [44]
	3.1.3	Lee et al. [45], Zhao et al. [46]
Local sensitivity	3.2.1	Robison et al. [47]
	3.2.2	Chen et al. [48,49,52], Olivier et al. [50,51]
Improvement of task queue	3.3	Tzannes et al. [53], Chase et al. [54], Lê et al. [55], Traoré et al. [56], Dinan et al. [57], Michael et al. [58], Kumar et al. [59], Quintin et al. [60], Wang et al. [61], Tsai et al. [62], Dijk et al. [63], Hendler et al. [64]
Other methods	3.4	Guo et al. [65,66], Paudel et al. [67], Cao et al. [68], Adnan et al. [69], Acar et al. [70], Herlihy et al. [71]

main technical contribution is the derivation of nontrivial high probability bounds on the number of steals incurred by these algorithms in the presence of false sharing, when using RWS.

Agrawal et al. [29] extended a randomized work-stealing scheduler and guarantees provably good performance to parallel algorithms that use parallel batched data structures. In particular, suppose a parallel algorithm has T_1 work, T_∞ span, and n data-structure operations. Let $W(n)$ be the total work of data structure operations and let $s(n)$ be the span of a size-P batch, then BATCHER executes the program in $O(T_1 + W(n) + ns(n) = P + s(n)T_\infty)$ expected time on P processors. For higher-cost data structures like search trees and large enough n, this bound becomes $(T_1 + n\lg n = P + T_\infty \lg n)$, provably matching the work of a sequential search tree but with nearly linear speedup, even though the data structure is accessed concurrently. The BATCHER runtime bound also readily extends to data structures with amortized bounds.

3 Optimization of Algorithm Implementation

The optimization research in terms of algorithm implementation has been done since 2006, focusing on implementation optimization of work stealing scheduling algorithm on multicore platform. It mainly includes research on task scheduling considering task granularity and local sensitivity, improvement of task queue, and other methods. See Table 3.

3.1 Scheduling Algorithms Considering Task Granularity

Fine-grain parallelism has several advantages [30]. First, it can expose more parallelism in many applications, and for some applications parallelism is more easily expressed under this model [31]. This is particularly important for Chip-MultiProcessors (CMPs) with hundreds of cores, for which parallelism becomes a precious resource [32]. Second, it makes the underlying runtime system much more freedom in distributing and reassigning work among cores in order to avoid load imbalance in irregular computations, and to exploit constructive cache interference among certain tasks [33], or to adapt to environment changes such as cores becoming unavailable due to faults, thermal emergencies, or multiprogramming. Task-parallel programming model is fine-grain parallel, allowing programmers to express all the available parallelism. Runtime system is responsible for mapping and scheduling these tasks to physical threads to execute. This programming method will produce a large number of fine-grain logical tasks to ensure load balacing, and uses logical tasks to replace physical threads, which is independent of the number of processors, and thus can improve the level of parallel programming.

Implementing work stealing scheduling algorithm in software on runtime system comes at a price: a large number of generated fine-grain tasks will incur high system overhead. On the contrary, a small amount of coarse-grain tasks will result in load imbalance, thus reducing performance. It can be seen that the system overhead is proportional to the number of tasks, and the load balancing is inversely proportional to the number of tasks. The appropriate task granularity needs to be balanced between two factors of system overhead and load balancing. How to determine the appropriate task granularity is an important problem for work stealing scheduling.

A task corresponds to a function. For example, when executing *spawn foo*() statement, thread pushes the status information of task *foo* onto the bottom of its task queue. At the same time, it allocates a stack frame for foo function in stack space. If it is just a function call, then there will be no overhead generated by tasks. So the serial function can reduce the system overhead, parallel tasks can balance the load, and the reciprocity method for serial functions and parallel tasks is the main means of controlling task granularity. At present, the implementation method of work stealing can only convert to serial function from parallel task, but not to parallel task from serial function. The following methods mainly aim to control task granularity from the perspective of serial and parallel reciprocity. It can only be converted to serial functions from parallel tasks for cut-off policy, just needs to be careful to choose conversion point. Adaptive task granularity can complete switching between parallel task and serial function. As there is no consideration of locality, the performance may not be distinctly improved for locality-sensitive application. The runtime stack and task queue combined into one can reduce maintenance overheads of task queue, but needs to modify the operating system.

In order for improving the performance of fine grain task parallelism, it is often either cumbersome or impossible to increase the grain size of such programs [34]. Increasing core counts exacerbates the problem; a program that appears coarse-grained on eight cores may well look a lot more fine-grained on sixty-four .

3.1.1 Cut-Off Policy

Mohr et al. [35], Loidl and Hammond [36], and Duran et al. [37] used cut-off policy to control recursive depth of function call to reduce number of tasks, and thus can reduce overhead incurred by tasks. Meanwhile, task granularity can also be controlled. Cut-off policy usually specifies a recursive depth of the derived tree (or function call tree) and no task is generated when the depth is greater than the specified number. This strategy works well for a balanced derivation tree. But for unbalanced derived trees, the cut-off strategy will cause system to be "hunger". That is, some threads are forced to be idle because of no task to work, which can lead to unbalanced load, thus reducing execution efficiency.

The five implementation methods of cut-off strategy are shown as following: (1) Programmers provide a cut-off recursive depth, or run-time system sets a default depth. This method is simple, but can't adapt to environmental changes. (2) Batching [38]. Runtime system sets the cut-off depth on the basis of the size of current task queue, thus controling task granularity. But this method requires programmers to set the serial program threshold, and the performance tuning need to be manually conducted. (3) Profiling [33]. Profile information of work set is collected and the collected information is then used to perform cut-off. This method is more effective for some applications having work sets. However, this method can do nothing for those applications having no definite work set such as backtracking search, branch and bound search and game tree. (4) Adaptive cut-off technique supported by runtime system [39]. The runtime system collects the number of tasks generated at each level. If the number of tasks is greater than two times the number of threads then cut off, and new tasks are no longer generated. But for irregular applications, this adaptive cut-off technology can't predict which branch has more tasks. So it will cause load imbalance, and thus can affect the performance. (5) User guided adaptive cut-off technique [40] expects to be able to predict the size of each branch, and then cuts off till this branch is not too big. The concrete implementation method is that programmers provide a cost calculation statement for each function, and then the runtime system predicts the size of each branch and decides whether to perform cut-off.

3.1.2 Adaptive Task Granularity

The key idea of adaptive task granularity is that no task is generated when all threads are busy and these busy threads start to generate tasks when idle threads need tasks. Traditional implementation method of work stealing scheduling algorithm makes parallel task can call serial function, but serial function can't generate parallel task so that no task can be generated after entering the serial function. In order to achieve adaptive task granularity, the difficulty lies in how to generate parallel tasks in serial function.

Wang et al. [41] can obtain adaptive task granularity of nested parallelization. Their method introduces a special task for generating parallel tasks again in a serial function. The execution status information of serial function is stored for the special task that can't be stolen and must be continued to execute after having completed all sub-tasks. This adaptive task generating strategy can switch between parallel task and serial function according to actual implementation.

As the specific parallel implementation of cilk_for cycle adopts divide and conquer method to segment iterative space and loop parallelization is converted to nested parallelization, so task granularity problem can also be solved with this method. Tzannes et al. [42] can obtain adaptive task granularity of parallel loop. It needs to check whether the task queue is empty before executing an iterative. If it is empty then divides the remaining and unexecuted iterative space into two, and pushes tasks to be executed onto the bottom of task deque to execute. Otherwise, the tasks aren't pushed onto the bottom of task deque and continue to execute. This method segments the leaf nodes of derivation tree and thus can solve the problem of task granularity for parallel loop.

Considering that the best strategy controlling task granularity usually depends on the application characteristics, Cao et al. [43] proposed an adaptive task granularity (AdaptiveTG) strategy. First, they applied breadth-first policy until all the available threads are busy, and then if some thread becomes idle, the work stealing technique, which provokes the victim to create a task with sufficient size at the oldest spawnable point, is introduced. The strategy doesn't only maximize the task granularity but also efficiently solves the unbalanced computation loads.

Task pools have been shown to provide efficient load balancing for irregular applications on heterogeneous platforms. Often, distributed data structures are used to store the tasks and the actual load balancing is achieved by task stealing where an idle processor accesses tasks from another processor. Hoffmann and Rauber [44] extended the concept of task pools to adaptive task pools which are able to adapt the number of tasks moved between the processor to the specific execution scenario, thus reducing the overhead for task stealing significantly.

3.1.3 Other Methods

The following research aims to solve the runtime system cost problem from the perspective of the operating system, hardware structure and compiler optimization.

Sanchez et al. [30] observed that the software implementation of scheduling method is very flexible, but the cost is high. The hardware implementation of scheduling method has the advantages of low cost but lacks flexibility. They combined hardware method with software method: software maintains task queue, and hardware performs work stealing. This method can reduce the overheads incurred by generating tasks and managing task queue, thus improving performance. Lee et al. [45] provided thread-private memory mapping (thread-local memory mapping, TLMM) supported by modifying operating system. The task queue is combined with function call stack by using TLMM mechanism, thus implementing the conversion from serial functions to parallel tasks and accordingly improving performance. TLMM may specify a virtual address space area of a process as being thread-private. That is, for the virtual address space, each thread has the same virtual addresses, but has different physical memory pages. For the stack space of each thread implemented with TLMM, the task and stack frame of its ancestors are stolen, thus combining the task queue with function call stack and reducing the overhead generated by task. For a double loop, the parallel_for is swapped from inner layer to outer layer by using compiler-optimizated method, the number of tasks can be reduced [46], thus reducing the cost generated by creating and ending this task, and improving performance.

3.2 Locality-Sensitive Scheduling Algorithm

Work-stealing uses work-first policy. Its depth-first execution strategy can improve the utilization rate of cache. It randomly selects a thread for stealing work without considering the characteristics of processor architecture, having an impact on the locality-sensitive applications. Locality-sensitive scheduling algorithm is mainly concerned with how to select a thread to be stolen.

3.2.1 Cache-Affinity Scheduling

Acar et al. [15] proposed a cache-affinity scheduling method to improve the utilization rate of cache. The specific method is that each thread has a task queue and mailbox that is used for storing the tasks being affinity with this thread. When thread 1 generates a task being affinity with thread 2, the pointer pointing to the task is put into the mailbox of thread 2 by thread 1. When thread 2 completes these tasks in its task queue, it first checks its mailbox and executes tasks in the mailbox, and then steals tasks.

Considering most of threads are idle and keep stealing tasks when starting to execute a cycle, Robison et al. [47] improved the cache-affinity scheduling method, and provided that the tasks in mailbox of an idle thread are not to be stolen, and thus can solve the problem that can't guarantee the cache-affinity scheduling at its initial stage.

3.2.2 Locality-Sensitive Scheduling in Multi-socket Multicore Architecture

Current servers basically adopt multi-socket multi-core architecture. The processor includes a plurality of multicore chips: the intra-chip multicores share L3 cache, and the inter-chip multicores share memory and provide cache consistency. The work stealing scheduling strategy randomly selects a thread to steal tasks, without distinguishing intra-CMP and inter-CMP threads. For locality-sensitive applications, the random work stealing scheduling strategy can reduce the utilization rate of cache, thus reducing its performance.

Chen et al. [48,49] presented a Cache Aware Bi-tier (CAB) scheduling strategy. According to hardware structure, the intra-CMP threads are grouped into a group. Each thread has an intra-group task queue, and each group has an inter-group task queue. The runtime system divides tasks into intra-group tasks and inter-group tasks along call tree, and then puts them into responding task queues. After having completed its own tasks in the intra-group task queue, the thread randomly chooses another thread in the same group to steal; when all intra-group task queues in a group are empty, they start to steal tasks in inter-group task queue of this group; when all intra-group and inter-group task queues in a group are empty, they then steal tasks from inter-group task queues of other groups.

Olivier et al. [50] proposed a hierarchical scheduling strategy, it is not that each thread has a task queue, but it is that intra-Chip threads share a task queue, thus only can perform work stealing scheduling between chips. In this method, the shared task queue may become a bottleneck. Olivier et al. [51] leveraged different scheduling methods at different levels of the hierarchy. By allowing one thread to steal work on behalf of all of the threads within a single chip that share a cache, the scheduler limits the number of costly remote steals. For cores on the same chip, a shared LIFO queue allows exploitation of cache locality between sibling tasks as well as between a parent task and its newly created child tasks.

Chen and Guo [52] proposed a locality-aware work-stealing (LAWS) scheduler. In LAWS, a load-balanced task allocator is used to evenly split and store the dataset of a program to all the memory nodes and allocate a task to the socket where the local memory node stores its data for reducing remote memory accesses. Then, an adaptive DAG packer adopts an auto-tuning approach to optimally pack an execution DAG into cache-friendly subtrees. After cache-friendly subtrees are created, every socket executes cache-friendly subtrees sequentially for optimizing shared cache usage. Meanwhile, a triple-level work-stealing scheduler is applied to schedule the subtrees and the tasks in each subtree. Through theoretical analysis, they showed that LAWS has comparable time and space bounds compared with traditional work-stealing schedulers.

3.3 Improvement of Task Queue

Since work thread and stealing thread directly operate the task queue of work thread, when a stealing thread attempts to steal a task which is to be popped up by work thread, it will incur data races. The work stealing mechanism can solve the race by using THE protocol [2]. The THE protocol requires atomic operations with high cost. Tzannes et al. [53] replaced the traditional THE protocol with duplicate queue method to avoid data race. The duplicate queue requires no atomic operations, thus can improve its performance.

When runtime system starts, it creates a fixed size of task queue for each thread, and this may cause task queue overflow. Chase and Lev [54] proposed dynamic cyclic task queue and solved the problem of task queue overflow. Lê et al. [55] provided the first correctness proof of an optimized implementation of Chase and Lev's deque on top of the POWER and ARM architectures: these provide very relaxed memory models, which they exploited to improve performance but considerably complicate the reasoning.

Traoré et al. [56] showed that, in the case of independent tasks, a whole subpart of an array of tasks can be represented in a compact way by the range of the corresponding indices, each cell containing the effective description of a task. Task queue organization impacts all the other aspects of task scheduling, and there are some interesting designs of task queue proposed in recent years. For example, some approaches split a task queue into two parts and only allow tasks in one part to be stolen [57]; some techniques expand the size of a task queue with automatic garbage collection [58]; hardware task queue is used to improve the performance in [59]; hierarchical task queues are studied on different parallel systems [60], which enable locality-aware victim selection. Considering the two levels of hierarchy on multi-core cluster, cluster nodes and multiple cores within a node, their task queues are hierarchically organized and victim is selected inside a node before crossing the node boundary during work-stealing [61]. Tsai and Huang [62] proposed a generalized parallelization framework for dynamic workload scheduling using adaptive work-stealing of thread pool and dynamic circular lock-free double-ended queue. Dijk and Pol [63] proposed a new non-blocking work-stealing deque based on the split task queue. Their design uses a dynamic split point between the shared and the private portions of the deque, and only requires memory fences when shrinking the shared portion.

Considering that the ABP algorithm's synchronization protocol [24] is strongly based on the use of fixed size arrays, which are prone to overflows, especially in multiprogrammed environments which they are designed for, Hendler et al. [64] presented the first dynamic memory work-stealing algorithm, which is based on a novel way of building non-blocking dynamic memory ABP deques by detecting synchronization conflicts based on "pointercrossing" rather than "gaps between indexes" as in the original ABP algorithm.

3.4 Other Methods

Guo et al. [65] proposed Scalable Locality-Aware Work stealing (SLAW). It is a scalable task scheduler that applied the adaptive locality-aware work-stealing technique and supported both work-first and helpfirst policies [66] at runtime on a per-task basis. Paudel et al. [67] explored the selection of tasks that are favourable for migration across nodes in a distributed memory cluster, a lesser-explored dimension to distributed work-stealing. The selection of tasks is guided by the application-level task locality rather than hardware memory topology as is the norm in the literature. Cao et al. [68] studied feedback-driven adaptive scheduling based on work stealing, which provides an efficient solution for concurrently executing a set of applications on multi-core systems. To dynamically estimate the number of cores desired by each application, they proposed a stable feedback algorithm, called ADeque, using the length of active deques, which more precisely captures the parallelism variation of the applications. Adnan and Sato [69] described extended work-stealing strategies for StackThreads/MP, in which thieves steal from the bottom of a victim's logical stack not just the bottommost task but multiple chained tasks. Acar et al. [70] proposed two work-stealing algorithms with private deques and proved that the algorithms guarantee similar theoretical bounds as work stealing with concurrent deques. Herlihy and Liu [71] focused primarily on structured single-touch computations, in which futures are used in a restricted way. They claimed that for such computations, a parallel execution by a work-stealing scheduler that runs future threads first can incur at most $O(CPT_\infty^2)$ cache misses more than the corresponding sequential execution, a substantially better cache locality than the $\Omega(CPT_\infty + CtT_\infty)$ worst-case additional cache misses possible with unstructured use of futures. However, they cannot prove this claim formally.

4 Processor Architecture Oriented Optimization

Current research on task scheduling, targeting processor architectures, involves the key techniques of work-stealing on manycore and cluster systems. See Table 4.

Table 4 Examples of processor architecture oriented optimization

Context	Section	References
Cluster systems	4.1	Cong et al. [38], Dinan et al. [57], Wang et al. [61], Guo et al. [66], Fohry et al. [72], Tardieu et al. [73,74], Agarwal et al. [75], Pezzi et al. [76], Saraswat et al. [77], Ravichandran et al. [78]
Manycore architecture	4.2	Cao et al. [79], Li et al. [80], Long et al. [81]

4.1 Hierarchical Task Scheduling on Cluster Systems

The X10 is based on the Asynchronous Partitioned Global Address Space (APGAS) programming model, supporting the same fine-grained concurrency mechanisms within and across shared-memory nodes [72–74]. It needs to simultaneously consider both intra- and inter-place task scheduling. There are some differences between intra-place task scheduling and traditional task scheduling such as Cilk: X10 is more general than Cilk in that it permits a parent activity to terminate while its child/descendant activities are still executing, thereby enabling an outer-level finish to serve as the root for exception handling and global termination [66,75]; considering the problem without indicating the parallel features of a function, the runtime system needs to maintain a function call relationship queue and a task queue for each thread. Function call relationship queue contains serial functions and parallel tasks, and the task queue only stores parallel tasks. This processing resembles putting each function (not distinguishing serial functions and parallel tasks) into the task queue, with high implementation cost of system. Cong et al. [38] presented XWS, the X10 Work Stealing framework, an open-source runtime for the parallel programming language X10 and a library to be used directly by application writers. Aiming at the stack-overflow problem caused by the spanning tree of a graph, Guo et al. [66] proposed help-first scheduling policy. Under this policy, the worker will create and push the task onto the deque and proceed to execute parent task.

Inter-place load balancing needs to extend the work stealing to distributed memory system, but prior work on work stealing largely focused on shared memory machines, its performance on clusters is not well understood. Dinan et al. [57] implemented work-stealing scheduling algorithm on large-scale clusters, and proposed following optimization methods for achieving performance improvement: (1) Split task queues. Each single task queue is split into local access and shared access portions: the local portion is located between the head and split pointers and is used for local threads to access tasks; the shared portion is located between the tail and split pointers and is used for other threads to steal tasks. The local portion of the queue can be accessed by the local process without locking and the shared portion can be accessed by any process and accesses are synchronized via a lock. Thread needs to periodically adjust the split-pointer's location between the head and tail of the queue for balancing the distribution of work between the public and private portions of the queue. (2) When a processor's task queue is empty, it generally attempts to steal a task from the head of victim processor's work queue on shared memory systems. However, the cost of stealing work on distributed memory systems is much higher than that on shared memory systems, thus stealing multiple tasks at once during a single steal attempt.

For MPI platforms, Pezzi et al. [76] proposed a hierarchical work stealing (HWS) algorithm. HWS employs a hierarchical structure of managers (i.e. master) and workers (i.e. slaves), which are arranged in a binary tree structure. The inner-nodes of the tree operate as managers and the leaf nodes operate as the worker nodes. Since in MPI platforms, processes can only communicate if they share an inter communicator, enabling one-to-one communication between large number of workers requires high implementation cost. HWS aims to reduce this cost by using managers which facilitate the task of communication between workers by mediating. However, their technique has the limitation of requiring several extra master (manager) nodes. Further, since each work stealing request must go through the manager nodes, compared to one-to-one communication, each work stealing request is slowed-down.

Saraswat et al. [77] pointed out that there are two main difficulties in extending Cilk-style work-stealing [3] to distributed memory: (1) In shared memory approach, thieves constantly attempt to asynchronously steal work from randomly chosen victims until they find work. In distributed memory, thieves cannot autonomously steal work from a victim without disrupting its execution. (2) It is non-trivial to detect active distributed termination. They proposed the idea of lifeline graphs. Each place adds an id of stolen thread. But if that place has no work then the id of the thief is recorded at that place as an “incoming” lifeline, so the id of stolen thread and the place form a directed graph. Such a graph is called lifeline graph and the id of stolen thread is called incoming edge of the place. Once a place finds some work, it then allocates them to incoming edge thread and clears the incoming edge. When a thread is unable to find work after w unsuccessful steals, it becomes a quiescent node and no longer steals tasks. A quiesced node is reactivated when work arrives from a lifeline. Termination occurs precisely when computation at all nodes has quiesced.

Considering that current work stealing approaches are not suitable for multi-core clusters due to the dichotomy of the underlying architecture. Ravichandran et al. [78] proposed a hierarchical work-stealing, which combines the best aspects of both the current approaches for shared memory architecture and distributed memory architecture into a new algorithm. The presented algorithm allows for more efficient execution of large-scale HPC applications, such as UTS, on clusters which have large multi-cores. Considering that high inter-node communication costs hinder work-stealing from being directly performed on multi-core clusters, Wang et al. [61] introduced an adaptive and hierarchical task scheduling scheme (AHS) for multi-core clusters. AHS can address this issue with initial partitioning for reducing the inter-node task migrations, with hierarchical scheduling scheme for performing work-stealing inside a node before going across the node boundary and adopting work-sharing to overlap computation and communication at the inter-node level, and with hierarchical and centralized control for inter-node task migration to improve the efficiency of victim selection and termination detection.

4.2 Task Scheduling on Manycore Architecture

Aiming to address the increasing difficulty of efficiently using large number of cores in manycore processors, Cao et al. [79] proposed a core-partitioned adaptive scheduling algorithm, named CASM (core-partitioned adaptive scheduling for manycore systems). CASM dynamically aggregates cores into different partitions by splitting or merging task-clusters, which ensures the efficiency of isolated accessing in these core partitions. To improve the scheduling efficiency of CASM, equi-partitioning scheduling algorithm is adopted to reallocate the cores among task-clusters, and the feedback-driven adaptive scheduling algorithm is implemented within the task-clusters.

The runtime system implemented in software on manycore systems runs with poor performance. Therefore, Li et al. [80] and Long et al. [81] have done some work on task scheduling implemented in hardware. In order to achieve a good balance between programmability and scalable hardware implementation, Long et al. [81] studied scalable hardware mechanisms to support programming model and proposed an architectural support for DAG consistency. Experimental results on a set of scientific benchmark programs show good performance speedup for a small number of cores. Experimental results also reveal two fundamental reasons which limit the performance scalability of computations on manycore architectures: the unbalanced on chip network bandwidth usage and limited memory band-width. Li et al. [80] presented a hardware support for conditional division-based approaches to parallel programming.

5 Conclusions and Recommendations

Work stealing is a scheduling strategy for multithreaded computer programs and has been gaining popularity as the multiprocessor load-balancing technology of choice in both Industry and Academia. As can be seen, there are several intensive researches in the field of work stealing scheduling other than just theoretical result and hence a huge body of literature. Instead of focusing on an extensive literature review in all aspects of work stealing scheduling, we have taken a holistic view to summarize theoretical results, optimization of algorithm implementation, and processor architecture oriented optimization. Though considerable efforts have been devoted to investigating work stealing scheduling, it is far from sufficient.

1. Current theoretical work-stealing scheduling analysis is mainly based on queuing theory. Unfortunately, so far there is no corresponding available computing and profiling tool and it is usually assumed that task load distribution is approximately satisfied the Markov chain (exponential distribution). However, many task distributions are not the case in many real world situations, and existing theoretical work stealing models may not so perfect that means it can be further improved and even reconstructed. When the synchronization overhead introduced by improving parallelism is more than the gain and the problems to be solved have coarse-grain parallelism or a high degree of dependence between tasks, it may not be worth the effort to use more multi-cores to solve these problems.

2. Traditional work stealing strategy mainly verifies its performance by simulation data or individual applications and concludes corresponding results. Parallel computing becomes popular in multi-core era, and it is not persuasive to verify scheduling performance just by selecting experimental data for individual applications, but needs to have a lot of applications to test its performance. To take a fresh approach to work stealing scheduling problem, instead of traditional benchmarks, the research agenda may be driven by compelling applications and motifs that are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications.
3. Runtime systems use work stealing scheduling to balance load. At present, the work stealing scheduling is based on shared memory multicore chips and the used approach is mobile computing without moving data. In recent years, both hardware and software have changed. On the hardware side, multi-core processor based on NUMA (non-uniform memory access) architecture is the mainstream in the future. For example, for the Nehalem's architecture of Intel and and Niagara architecture of Oracle, the processors include a plurality of multi-core chips that each multi-core chip has its own memory controller, so the memory access is no longer a unified access mode, but includes local memory and remote memory. The local memory can directly access through the IMC (integrated memory controller) and is fast, and the remote memory can access through the QPI (quickpath interconnect) and is slow. In addition, the development trend in the future is the heterogeneous platform that composed of general multi-core processor and acceleration components such as GPU etc. How to scheduling this kind of platform is a hard work.

The research on work stealing scheduling will be driven by new problems in theory, emerging applications and architectures, so at least the following existing problems and promising efforts need particular attention.

1. Theory-driven research. In order to reveal and profile the statistical laws of parallel and distributed computing, new theoretical models for work stealing scheduling will be developed and the scheduling algorithm needs to be redesigned to improve the performance of parallel and distributed computing. The work stealing scheduling strategy should retarget achieving required performance by using the least resources, instead of generating hundreds of threads to have all the processor cores running busily. In this regard, further research is still needed.
2. Application- and motif-driven research. With the rise of cloud computing and social networks, a series of emerging applications such as social networks, network security, data mining and bioinformatics are emerging. Most of the core algorithms involved in these emerging computing fields belong to the category of non-numerical computation, and contain irregular parallelism and pose data-intensive characteristics. These applications can generally be modelled as a complex networks that can be analyzed by using graph related algorithms. Therefore, how to support these irregular applications is another research direction for work stealing scheduling. For these emerging irregular applications, work stealing scheduling strategy needs to provide efficient concurrent queues and concurrent priority queues to improve the applicability and performance. For example, when performing parallel graph calculation, active nodes are handled in parallel and common

queue and priority queue are used to manage the active nodes for graph calculation. Therefore, if we can provide efficient concurrent queues and concurrent priority queue, then we can improve the scheduling efficiency and performance of this kind of application. Instead of traditional benchmarks, we can use 13 "motifs" [82] to design and evaluate work stealing scheduling strategy. Each motif captures a pattern of computation and communication common to a class of important applications and motifs are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications. The work-stealing scheduling strategies based on these motifs affecting future parallel computing deserve further study.

3. Architecture-driven research. For multi-socket multi-core processors with NUMA architecture, work stealing scheduling strategy needs to consider the appropriate data distribution. Task parallelism model considers the speed that each core accesses the shared memory is consistent, having no problem with data distribution. But the memory access speed is not consistent due to the NUMA structure, and if the data and computation are on the same multi-core chip then the computing speed will be greatly improved. Therefore, work stealing scheduling strategy needs to consider the appropriate data distribution. For the heterogeneous platforms, the idea of task parallel mechanism is to divide the application into a large number of fine grained tasks for parallel execution, and thus it can generate large amounts of a small amount of data communication between the general processor and GPU on heterogeneous platforms. However, the data transmission between the general processor and GPU is very slow, and each time the transmission time of small data and big data is the same. Therefore, work stealing scheduling strategy should consider aggregating a large amount of fine grained tasks into a large task and then passing it to the GPU for execution. This can reduce the number of communications, thereby improving the scheduling performance.

Acknowledgements This work was supported by Chongqing Natural Science Foundation (No. KJ1400316), partially supported by National Natural Science Foundation of China (No. 11401061).

References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.L.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
2. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98), pp. 212–223. ACM, New York, NY, USA, June 16–19 (1998)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
4. Burton, F.W., Sleep, M.R.: Executing functional programs on a virtual tree of processors. In: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (FPCA'81), pp. 187–194. ACM, New York, NY, USA, October 18–22 (1981)
5. Halstead, R.H.: Implementation of multilisp: Lisp on a multiprocessor. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP'84), pp. 9–17. ACM, New York, NY, USA, August 6–8 (1984)
6. Mohr, E., Kranz, D.A., Halstead, J.R.H.: Lazy task creation: a technique for increasing the granularity of parallel programs. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP'90), pp. 185–197. ACM, New York, NY, USA, June 27–29 (1990)

7. Vrba, Ž., Espeland, H., Halvorsen, P., Griwodz, C.: Limits of work-stealing scheduling. In: Proceedings of the 14th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'09), pp. 280–299. Springer, Berlin, Germany, May 29 (2009)
8. Squillante, M.S., Nelson, R.D.: Analysis of task migration in shared-memory multiprocessor scheduling. In: Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'91), New York, NY, USA, May 21–24 (1991)
9. Mitzenmacher, M.: Analyses of load stealing models based on differential equations. In: Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'98), pp. 212–221. ACM, New York, NY, USA, June 28–July 2 (1998)
10. Rudolph, L., Slivkin-Allalouf, M., Upfal, E.: A simple load balancing scheme for task allocation in parallel machines. In: Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'91), pp. 237–245. New York, NY, USA, July 21–24 (1991)
11. Eager, D.L., Lazowska, E.D., Zahorjan, J.: A comparison of receiver-initiated and sender-initiated adaptive load sharing. Perform. Eval. **6**(1), 53–68 (1986)
12. Mirchandaney, R., Towsley, D., Stankovic, J.A.: Analysis of the effects of delays on load sharing. IEEE Trans. Comput. **38**(11), 1513–1525 (1989)
13. Squillante, M.S., Lazowska, E.D.: Using processor-cache affinity information in shared-memory multiprocessor scheduling. IEEE Trans. Parallel Distrib. Syst. **4**(2), 131–143 (1993)
14. Squillante, M.S., Xia, C.H., Yao, D.D., Zhang, L.: Threshold-based priority policies for parallel-server systems with affinity scheduling. In: Proceedings of the 2001 American Control Conference (ACC'01), pp. 2992–2999. IEEE, New York, NY, USA, June 25–27 (2001)
15. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. Theory Comput. Syst. **35**(3), 321–347 (2002)
16. Narang, A., Shyamasundar, R.K.: Performance driven distributed scheduling of parallel hybrid computations. Theor. Comput. Sci. **412**(32), 4212–4225 (2011)
17. Suksompong, W., Leiserson, C.E., Schardl, T.B.: On the efficiency of localized work stealing. Inf. Process. Lett. **116**(2), 100–106 (2016)
18. Mirchandaney, R., Towsley, D., Stankovic, J.A.: Adaptive load sharing in heterogeneous distributed systems. J. Parallel Distrib. Comput. **9**(4), 331–346 (1990)
19. Bender, M.A., Rabin, M.O.: Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. Theory Comput. Syst. **35**(3), 289 (2002)
20. Gast, N., Bruno, G.: A mean field model of work stealing in large-scale systems. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'10), pp. 13–24. ACM, New York, NY, USA, June 14–18 (2010)
21. Blelloch, G.E., Gibbons, P.B., Matias, Y.: Provably efficient scheduling for languages with fine-grained parallelism. J. ACM **46**(2), 281–321 (1999)
22. Fatourou, P., Spirakis, P.: Efficient scheduling of strict multithreaded computations. Theory Comput. Syst. **33**(3), 173–232 (2000)
23. Berenbrink, P., Friedetzky, T., Goldberg, L.A.: The natural work-stealing algorithm is stable. SIAM J. Comput. **32**(5), 1260–1279 (2003)
24. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory Comput. Syst. **34**(2), 115–144 (2001)
25. Tchiboukdjian, M., Gast, N., Trystram, D.: Decentralized list scheduling. Ann. Oper. Res. **207**(1), 237–259 (2013)
26. Tchiboukdjian, M., Gast, N., Trystram, D., Roch, J., Bernard, J.: A tighter analysis of work stealing. In: Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC'10), pp. 291–302. Springer, Berlin, Germany, December 15–17 (2010)
27. Agrawal, K., Leiserson, C.E., Sukha, J.: Helper locks for fork-join parallel programming. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10), pp. 245–256. ACM, New York, NY, USA, January 9–14 (2010)
28. Cole, R., Ramachandran, V.: Analysis of randomized work stealing with false sharing. In: Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13), pp. 985–998. IEEE, Los Alamitos, CA, USA, May 20–24 (2013)
29. Agrawal, K., Fineman, J.T., Sheridan, B., Sukha, J., Utterback, R.: Provably good scheduling for parallel programs that use data structures through implicit batching. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architecture (SPAA'14), pp. 84–95. ACM, New York, NY, USA, June 23–25 (2014)

30. Sanchez, D., Yoo, R.M., Kozyrakis, C.: Flexible architectural support for fine-grain scheduling. In: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10), pp. 311–322. ACM, New York, NY, USA, March 13–17 (2010)
31. Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Chew, L.P.: Scheduling strategies for optimistic parallel execution of irregular programs. In: Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08), pp. 217–228. ACM, New York, NY, USA, June 14–16 (2008)
32. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**(7), 33–38 (2008)
33. Chen, S., Gibbons, P.B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G.E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T.C., Wilkerson, C.: Scheduling threads for constructive cache sharing on CMPs. In: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07), pp. 105–115. ACM, New York, NY, USA, June 9–11 (2007)
34. Faxén, K.F.: Efficient work stealing for fine grained parallelism. In: Proceedings of the 39th International Conference on Parallel Processing (ICPP'10), pp. 313–322. IEEE, Los Alamitos, CA, USA, September 13–16 (2010)
35. Mohr, E., Kranz, D.A., Halstead Jr., R.H.: Lazy task creation: a technique for increasing the granularity of parallel programs. IEEE Trans. Parallel Distrib. Syst. **2**(3), 264–280 (1991)
36. Loidl, H.W., Hammond, K.: On the granularity of divide-and-conquer parallelism. In: Proceedings of the 1995 Glasgow Workshop on Functional Programming (FP'95), pp. 8–10. Springer, Berlin, Germany, July 10–12 (1995)
37. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP task scheduling strategies. In: Proceedings of the 4th International Workshop on OpenMP in New Era of Parallelism (IWOMP'08), pp. 100–110. Springer, Berlin, Germany, May 12–14 (2008)
38. Cong, G., Kodali, S., Krishnamoorthy, S., Lea, D., Saraswat, V., Wen, T.: Solving large, irregular graph problems using adaptive work-stealing. In: Proceedings of the 37th International Conference on Parallel Processing (ICPP'08), pp. 536–545. IEEE, Washington, DC, USA, September 9–12 (2008)
39. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08), pp. 339–349. IEEE, Piscataway, NJ, USA, November 15–21 (2008)
40. Acar, U.A., Charguéraud, A., Rainey, M.: Oracle scheduling: controlling granularity in implicitly parallel languages. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Ap-plications (OOPSLA'11), pp. 499–518. ACM, New York, NY, USA, October 22–27 (2011)
41. Wang, L., Cui, H., Duan, Y., Lu, F., Feng, X., Yew, P.: An adaptive task creation strategy for work-stealing scheduling. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10). ACM, New York, NY, USA, April 24–28 (2010)
42. Tzannes, A., Caragea, G.C., Barua, R., Vishkin, U.: Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10), pp. 179–190. ACM, New York, NY, USA, January 9–14 (2010)
43. Cao, Q., Hu, C.J., Li, S.G., He, H.H.: Adaptive task granularity strategy for OpenMP3.0 task model on cell architecture. In: Proceedings of International Conferences on High Performance Networking, Computing, Communication Systems, and Mathematical Foundations (ICHCC 2011), pp. 393–400. Springer, Berlin, Germany, May 5–6 (2011)
44. Hoffmann, R., Rauber, T.: Fine-grained task scheduling using adaptive data structures. In: Proceedings of the 14th International European Conference on Parallel Processing (Euro-Par'08), pp. 253–262. Springer, Berlin, Germany, August 26–29 (2008)
45. Lee, I.A., Boyd-Wickizer, S., Huang, Z., Leiserson, C.E.: Using memory mapping to support cactus stacks in work-stealing runtime systems. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10), pp. 411–420. ACM, New York, NY, USA, September 11–15 (2010)
46. Zhao, J.S., Shirako, J., Nandivada, V.K., Sarkar, V.: Reducing task creation and termination overhead in explicitly parallel programs. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10), pp. 169–180. ACM, New York, NY, USA, September 11–15 (2010)
47. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08), pp. 1–8. IEEE, New York, NY, USA, April 14–18 (2008)

48. Chen, Q., Huang, Z., Guo, M., Zhou, J.: CAB: cache aware bi-tier task-stealing in multi-socket multi-core architecture. In: Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11), pp. 722–732. IEEE, New York, NY, USA, September 13–16 (2011)
49. Chen, Q., Guo, M.Y., Huang, Z.Y.: Adaptive cache aware bitier work-stealing in multisocket multicore architectures. IEEE Trans. Parallel Distrib. Syst. **24**(12), 2334–2343 (2013)
50. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'11), pp. 49–56. ACM, New York, NY, USA, May 31 (2011)
51. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. Int. J. High Perform. Comput. Appl. **26**(2), 110–124 (2012)
52. Chen, Q., Guo, M.: Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures. ACM Trans. Arch. Code Optim. **12**(222), 1–24 (2015)
53. Tzannes, A., Caragea, G.C., Barua, R., Vishkin, U.: Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10), pp. 179–189. ACM, New York, NY, USA, January 9–14 (2010)
54. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'05), pp. 21–28. ACM, New York, NY, USA, July 18–20 (2005)
55. Lê, N.M., Pop, A., Cohen, A., Nardelli, F.Z.: Correct and efficient work-stealing for weak memory models. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13), pp. 69–79. ACM, New York, NY, USA, February 23–27 (2013)
56. Traoré, D., Roch, J., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel STL algorithms. In: Proceedings of the 14th International European Conference on Parallel Processing (Euro-Par'08), pp. 887–897. Springer, Berlin, Germany, August 26–29 (2008)
57. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09), pp. 1–11. ACM, New York, NY, USA, November 14–20 (2009)
58. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent work stealing. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09), pp. 45–53. ACM, New York, NY, USA, February 14–18 (2009)
59. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: architectural support for fine-grained parallelism on Chip multiprocessors. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, Conference (ISCA'07), pp. 162–173. ACM, New York, NY, USA, June 9–13 (2007)
60. Quintin, J., Wagner, F.: Hierarchical work-stealing. In: Proceedings of the 16th International European Conference on Parallel Processing (Euro-Par'10). Springer, Berlin, Germany, August 31–September 3 (2010)
61. Wang, Y., Zhang, Y., Su, Y., Wang, X., Chen, X., Ji, W., Shi, F.: An adaptive and hierarchical task scheduling scheme for multi-core clusters. Parallel Comput. **40**(10), 611–627 (2014)
62. Tsai, Y.C., Huang, Y.C.: A generalized framework for parallelizing traffic sign inventory of video log images using multicore processors. Comput. Aided Civ. Infrastruct. Eng. **27**(7SI), 476–493 (2012)
63. van Dijk, T., van de Pol, J.C.: Lace: non-blocking split deque for work-stealing. In: Proceedings of the 20th International European Conference on Parallel Processing (Euro-Par'14), pp. 206–217. Springer, Berlin, Germany, August 25–26 (2014)
64. Hendler, D., Lev, Y., Shavit, N.: Dynamic memory ABP work-stealing. In: Proceedings of the 18th International Conference on Distributed Computing (DISC 2004), pp. 188–200. Springer, Berlin, Germany, October 4–8 (2004)
65. Guo, Y., Zhao, J.Z., Cave, V., Sarkar, V.: SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10), pp. 341–342. ACM, New York, NY, USA, January 09–14 (2010)
66. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-first and help-first scheduling policies for async-finish task parallelism. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09), pp. 1–12. IEEE, New York, NY, USA, May 23–29 (2009)
67. Paudel, J., Tardieu, O., Amaral, J.N.: On the merits of distributed work-stealing on selective locality-aware tasks. In: Proceedings of the 2013 42nd International Conference on Parallel Processing (ICPP'13), pp. 100–109. IEEE, New York, NY, USA, October 1–4 (2013)

68. Cao, Y., Sun, H.Y., Qian, D.P., Wu, W.G.: Stable adaptive work-stealing for concurrent multicore runtime systems. In: Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC'11). IEEE, New York, NY, USA, September 2–4 (2011)
69. Adnan, Sato, M.: Efficient work-stealing strategies for fine-grain task parallelism. In: Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11). IEEE, Los Alamitos, CA, USA, May 16–22 (2011)
70. Acar, U.A., Charguéraud, A., Rainey, M.: Scheduling parallel programs by work stealing with private deques. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13), pp. 219–228. ACM, New York, NY, USA, February 23–27 (2013)
71. Herlihy, M., Liu, Z.: Well-structured futures and cache locality. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), pp. 155–166. ACM, New York, NY, USA, February 15–19 (2014)
72. Fohry, C., Bungart, M., Posner, J.: Fault tolerance schemes for global load balancing in X10. Scalable Comput. Pract. Exp. **16**(2SI), 169–185 (2015)
73. Tardieu, O., Wang, H., Lin, H.: A work-stealing scheduler for X10's task parallelism with suspension. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12), pp. 267–276. ACM, New York, NY, USA, February 25–29 (2012)
74. Tardieu, O., Herta, B., Cunningham, D., Grove, D., Kambadur, P., Saraswat, V., Shinnar, A., Takeuchi, M., Vaziri, M., Zhang, W.: X10 and APGAS at Petascale. ACM Trans. Parallel Comput. **2**(4), 1–32 (2015)
75. Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shymasundar, R.K., Yelick, K.: Deadlock-free scheduling of X10 computations with bounded resources. In: Proceedings of the 19th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'07), pp. 229–240. ACM, New York, NY, USA, June 9–11 (2007)
76. Pezzi, G.P., Cera, M.C., Mathias, E., Maillard, N., Navaux, P.O.A.: Online scheduling of MPI-2 programs with hierarchical work stealing. In: Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07), pp. 247–254. IEEE, Los Alamitos, CA, USA, October 24–27 (2007)
77. Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S.: Lifeline-based global load balancing. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11). ACM, New York, NY, USA, February 12–16 (2011)
78. Ravichandran, K., Sangho, L., Pande, S.: Work stealing for multi-core HPC clusters. In: Proceedings of the 17th International European Conference on Parallel Processing (Euro-Par'11), pp. 205–217. Springer, Berlin, Germany, August 29–September 2 (2011)
79. Cao, Y.J., Qian, D.P., Wu, W.G., Dong, X.S.: Adaptive scheduling algorithm based on dynamic core-resource partitions for manycore processor systems. Ruanjian Xuebao J. Softw. **23**(02), 240–252 (2012)
80. Li, Z., Certner, O., Duato, J., Temam, O.: Scalable hardware support for conditional parallelization. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10), pp. 157–168. ACM, New York, NY, USA, September 11–15 (2010)
81. Long, G.P., Zhang, J.C., Fan, D.R.: Architectural support and evaluation of Cilk language on many-core architectures. Chin. J. Comput. **31**(11), 1975–1985 (2008)
82. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J.D., Lee, E.A., Morgan, N., Necula, G., Patterson, D.A., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.A.: The parallel computing laboratory at U.C. Berkeley: a research agenda based on the Berkeley view, Technical Report No. UCB/EECS-2008-23, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley (2008)