Lace: Non-blocking Split Deque for Work-Stealing

Tom van Dijk* and Jaco C. van de Pol

Formal Methods and Tools, Dept. of EEMCS, University of Twente
P.O.-box 217, 7500 AE Enschede, The Netherlands
{t.vandijk,vdpol}@cs.utwente.nl

Abstract. Work-stealing is an efficient method to implement load balancing in fine-grained task parallelism. Typically, concurrent deques are used for this purpose. A disadvantage of many concurrent deques is that they require expensive memory fences for local deque operations.

In this paper, we propose a new non-blocking work-stealing deque based on the split task queue. Our 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.

We present Lace, an implementation of work-stealing based on this deque, with an interface similar to the work-stealing library Wool, and an evaluation of Lace based on several common benchmarks. We also implement a recent approach using private deques in Lace. We show that the split deque and the private deque in Lace have similar low overhead and high scalability as Wool.

Keywords: work-stealing, task-based parallelism, dynamic load balancing, lock-free algorithm, non-blocking deque.

1 Introduction

1.1 Task-Based Parallelism

In recent years, the importance of using parallelism to improve the performance of software has become self-evident, especially given the availability of multicore shared-memory systems and the physical limits of processor speeds. Frameworks like Cilk [3,9] and Wool [7,8] allow writing parallel programs in a style similar to sequential programs [1].

In task-based parallelism, a computation is divided into small tasks. Each task only depends on the results of its own immediate subtasks for its execution. Multiple independent subtasks can be executed in parallel. Especially recursive algorithms are easily parallelized.

Cilk, Wool, and similar task-based parallel frameworks use keywords **spawn** and **sync** to expose parallelism. The **spawn** keyword creates a new task. The **sync** keyword matches with the last unmatched **spawn**, i.e., operating as if spawned tasks are stored on a stack. It waits until that task is completed and

* The first author is supported by the NWO project MaDriD, grant nr. 612.001.101.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 206–217, 2014.

```
def spawn(task):
  self.tasks.push(task)

def sync():
  status, t = self.tasks.pop()
  if status = STOLEN:
    while not t.done:
      steal_work(t.thief)
    self.tasks.pop_stolen()
    return t.result
  else: return t.execute()

def steal_work(victim):
  t = victim.tasks.steal()
  if t != None:
    t.thief = self
    t.result = t.execute()
    t.done = True

thread worker(id, roottask):
  if id = 0: roottask.execute()
  else: forever:
    steal_work(random_victim())
```

Fig. 1. Simplified algorithm of work-stealing using leapfrogging when waiting for a stolen task to finish, i.e., steal from the thief. Note that stolen tasks are not removed from the task pool until completed.

retrieves the result. Every **spawn** during the execution of the program must have a matching **sync**. In this paper, we follow the semantics of Wool. In the original work-stealing papers, **sync** waits for all locally spawned subtasks, rather than the last unmatched subtask.

1.2 Work-Stealing

Work-stealing is a technique that efficiently implements load-balancing for task-based parallelism. It has been proven to be optimal for a large class of problems and has tight memory and communication bounds [4]. In work-stealing, tasks are executed by a fixed number of workers. Each worker owns a task pool into which it inserts spawned tasks. Idle workers steal tasks from random victims.

See Figure 1 for a simplified work-stealing algorithm. Workers start executing in `worker`. One worker executes the first task. The other workers steal from random victims. The task pool `tasks` acts like a stack with methods `push` and `pop`, and provides `steal` for potential thieves. Tasks are typically stolen from the bottom of the stack, since these tasks often have more subtasks. This reduces the amount of total steals necessary and thus the overhead from stealing.

When synchronizing with a stolen task, the victim steals from the thief until the stolen task is completed. By stealing back from the thief, a worker executes subtasks of the stolen task. This technique is called leapfrogging [16]. When stealing from random workers instead, the size of the task pool of each worker could grow beyond the size needed for complete sequential execution [8]. Using leapfrogging rather than stealing from random workers thus limits the space requirements of the task pools to those of sequential execution.

1.3 Work-Stealing Deques

Task pools are commonly implemented using double-ended queues (deques) specialized for work-stealing. The first provably efficient work-stealing scheduler for fully strict computations was presented in 1994 [4] and its implementation in

Cilk in 1996 [3]. One improvement of the original Cilk algorithm is the THE protocol in Cilk-5 [9], which eliminates acquiring the lock in push and in most executions of pop, but every steal still requires locking.

The first non-blocking work-stealing deque is the ABP algorithm, which uses a fixed-size array that might overflow [2]. Two unbounded non-blocking deques were proposed, the deque by Hendler et al. based on linked lists of small arrays [10], and the Chase-Lev deque that uses dynamic circular arrays [5].

In weak memory models that allow reordering loads before stores, most deques that allow any spawned task to be stolen require a memory fence in every pop operation. Memory fences are expensive. For example, the THE protocol spends half of its execution time in the memory fence [9].

Several approaches alleviate this problem. The split task queue by Dinan et al. [6], designed for clusters of multiprocessor computers, allows lock-free local access to a private portion of the queue and can transfer work between the public and private portions of the queue without copying tasks. Thieves synchronize using a lock and the local process only needs to take the lock when transferring work from the public portion to the private portion of the queue. Michael et al. propose relaxed semantics for work-stealing: inserted tasks are executed *at least* once instead of *exactly* once, to avoid requiring memory fences and atomic instructions [12]. In the work scheduler Wool [7], originally only the first N tasks in the deque can be stolen, where N is determined by a parameter at startup. Only executing pop on stealable tasks requires a memory fence. In a later version, the number of stealable tasks is dynamically updated [8].

In some work-stealing algorithms, shared deques are replaced by private deques, and work is explicitly communicated using a message-passing approach. Recently, Acar et al. proposed two algorithms for work-stealing using private deques [1]. See further [1] for an overview of other work with private deques.

Tasks are often stored as pointers that are removed from the deque when the task is stolen [9,2,10,5]. To virtually eliminate the overhead of task creation for tasks that are never stolen, Faxén proposed a direct task stack, storing tasks instead of pointers in the work queue, implemented in Wool [7,8]. Rather than synchronizing with thieves on the metadata of the queue (e.g. variables top and bot in the ABP algorithm), Wool synchronizes on the individual task descriptors, using locks when synchronizing with potential thieves, similar to the THE protocol. Sundell and Tsigas presented a lock-free version of Wool [15,8], which still synchronizes on the individual task descriptors.

1.4 Contributions

Acar et al. write that concurrent deques suffer from two limitations: 1) local deque operations (mainly pop) require expensive memory fences in modern weak-memory architectures; 2) they can be very difficult to extend to support various optimizations, especially steal-multiple extensions [1]. They lift both limitations using private deques. Wool reduces the first limitation for concurrent deques by using a dynamic number of stealable tasks, but is difficult to extend for steal-multiple strategies, since tasks must be stolen individually.

We present a work-stealing algorithm that eliminates these limitations using concurrent deques, by combining a non-blocking variant of the split task queue [6] with direct task stealing from Wool [7,8]. This algorithm splits the deque into a shared portion and a private portion. The split point between these portions is modified in a non-blocking manner.

We present an implementation of this algorithm in a C library called Lace[1], which has the same interface as Wool. We evaluate the performance of Lace using several benchmarks, including standard Cilk benchmarks and the UTS benchmark [13]. We compare our algorithm with Wool and with an implementation of the receiver-initiated private deque algorithm [1] in the Lace framework. Our experiments show that our algorithm is competitive with both Wool and the private deque algorithm, while lifting both limitations described in [1]. Compared to the private deque algorithm, our algorithm allows stealing of all tasks in the shared deque without cooperation of the owner, while the private deque algorithm requires cooperation of the owner for every steal transaction.

2 Preliminaries

We assume a shared memory system with the x86 memory model. The x86 memory model is not sequentially consistent, but allows reordering loads before stores. Memory writes are buffered before reaching the memory, hence reads can occur before preceding memory writes are globally visible. Memory fences flush the write buffer before reading from memory. Apart from memory fences, we use the atomic memory operation compare_and_swap (`cas`) to ensure safety. The `cas` operation atomically compares a value in memory to an expected value and modifies it only if the values match. We use `cas` to ensure that exactly one worker performs a transition.

We assume that the system consists of one or more processor chips and one or more memory chips, connected using an interconnection network, for example in Non-Uniform Memory Access (NUMA) shared-memory systems. We also assume that data on this interconnection network is transferred in blocks called cachelines, which are typically 64 bytes long.

3 Algorithm

3.1 Design Considerations

To obtain a low execution time when performing work-stealing with all available workers, we aim at low overhead compared to purely sequential programs and good scalability with increasing worker count. Memory fences and `cas` operations increase the overhead compared to purely sequential programs. Some memory fences are unavoidable, since thieves may steal a task while the owner is retrieving it. By splitting the deque into a shared deque and a private deque (see Figure 2),

[1] Lace is available at `http://fmt.ewi.utwente.nl/tools/lace/`

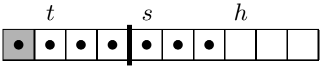

Fig. 2. The split deque, with tail t, split point s and head h. A task at position x is **stolen** if $x < t$. It is **shared** if $x < s$, and **private** otherwise. Of the 7 tasks in this example, 4 are **shared** and 1 is **stolen**.

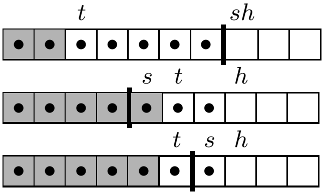

Fig. 3. The owner shrinks the shared portion of the deque, but thieves may have stolen tasks beyond the new split point. The owner detects this and updates the split point to its final position.

we only need a memory fence when shrinking the shared deque, to detect the scenario of Figure 3. Also, `cas` operations are only needed to coordinate stealing.

The deque is described using variables `tail`, `split` and `head`, which are indices in the task array. To steal work, thieves only require knowledge of `tail` and `split`, and only need to modify `tail`. The owner uses `head` and `o_split` (a private copy of `split`) to operate on the private deque. The owner only accesses `tail` and `split` when changing the split point.

Thieves are not allowed to change the split point, since this would force a memory fence on every execution of `pop`. Instead, thieves set a shared flag `splitreq` on a dedicated cacheline when there are no more unstolen shared tasks. Since `splitreq` is checked at every execution of `pop` and `push`, it should always be in the processor cache of the owner, and no traffic on the interconnect network is expected until the flag is set. There is no other communication between the owner and the thieves, except when tasks are stolen soon after their creation, or when the owner is waiting for an unfinished stolen task.

If the owner determines that all tasks have been stolen, it sets a flag `allstolen` (and a private copy `o_allstolen`). Thieves check `allstolen` first before attempting to steal tasks, which results in a small performance gain. When the owner already knows that all tasks are stolen, it does not need to shrink the shared deque until new tasks are added.

Similar to the direct task stack in Wool, the deque contains fixed-size task descriptors, rather than pointers to task descriptors stored elsewhere. Stolen tasks remain in the deque. The result of a stolen task is written to the task descriptor. This reduces the task-creation overhead of making work available for stealing, which is important since most tasks are never stolen. Another advantage is that the cachelines accessed by a thief are limited to those containing the task descriptor and the variables `tail`, `split` and (rarely) `splitreq`, while in designs that use pointers, there is at least one additional accessed cacheline. If task descriptors are properly aligned and fit into one cacheline, then thieves only access two cachelines per successful steal. Also, in a pointer-based design, there are many pointers per cacheline, which can increase contention on that cacheline.

```
def steal():
  if allstolen: return None
  (t,s) = (tail,split)
  if t < s:
    if cas((tail,split),
           (t,s), (t+1,s)):
      return Task(t)
    else: return None
  if ! splitreq: splitreq=1
  return None

def push(data):
  if head == size: return FULL
  write task data at head
  head = head + 1
  if o_allstolen:
    (tail,split) = (head-1,head)
    allstolen = 0
    if splitreq: splitreq=0
    o_split = head
    o_allstolen = 0
  elif splitreq: grow_shared()

def pop():
  if head = 0: return EMPTY,-
  if o_allstolen:
    return STOLEN, Task(head-1)
  if o_split = head:
    if shrink_shared():
      return STOLEN, Task(head-1)
  head = head-1
  if splitreq: grow_shared()
  return WORK, Task(head)

def pop_stolen():
  head = head-1
  if ! o_allstolen:
    allstolen = 1
    o_allstolen = 1

def grow_shared():
  new_s = (o_split+head+1)/2
  split = new_s
  o_split = new_s
  splitreq = 0

def shrink_shared():
  (t,s) = (tail,split)
  if t != s:
    new_s = (t+s)/2
    split = new_s
    o_split = new_s
    MFENCE
    t = tail # read again
    if t != s:
      if t > new_s:
        new_s = (t+s)/2
        split = new_s
        o_split = new_s
      return False
  allstolen = 1
  o_allstolen = 1
  return True
```

Fig. 4. Algorithm of the non-blocking split deque. Thieves have access to the cacheline with tail, split and allstolen and to the cacheline with splitreq. The owner also has access to the cacheline with head, o_split and o_allstolen.

3.2 Algorithms

See Figure 4 for the deque algorithms. Note that if allstolen is not set, then tail $\leq$ split $\leq$ head. If allstolen is set, then tail $\geq$ split and tail $\geq$ head.

The steal operation tries to steal a task by increasing tail, using cas on the (consecutive) variables tail and split. The cas operation fails when other thieves have changed tail, or when the owner has changed split. If there is no available work, then splitreq is set. It is important that splitreq is only written to if it must be changed, to avoid unnecessary communication.

Method push adds a new task to the deque and increases head. If this is the first new task (i.e., allstolen is set), then tail and split are set to reflect that the new task is shared and that it is the next task to be stolen. All tasks before the new task remain stolen tasks. Note that tail and split must be updated simultaneously. If thieves have set splitreq, then push calls grow_shared to move the split point.

Method pop determines whether the last task is stolen. This is the case when allstolen is set, or when all tasks are shared (i.e., o_split = head) and the method shrink_shared reports that all tasks are stolen. If the last task is stolen, then it remains on the deque. If the last task is not stolen, then head is decreased, and if splitreq is set, pop calls grow_shared.

If the last task is stolen, then pop_stolen is called after the stolen task is completed (see Figure 1). Leapfrogging may have changed the state of the deque, therefore allstolen is set again, since the remaining tasks are still stolen.

In grow_shared, the new value of the split point is the ceiling of the average of split and head. Since grow_shared is only called if not allstolen, i.e., split ≤ head, the shared deque will always grow and therefore atomic operations or memory fences are not necessary.

Method shrink_shared moves the split point to decrease the size of the shared deque. It then detects whether thieves have stolen tasks beyond the new split point, and if so, it moves the split point again. If all tasks were stolen, then shrink_shared sets allstolen and returns True. It returns False otherwise. Since shrink_shared is called by the owner only if split = head, line 43 really checks whether tail = head, i.e., whether all tasks are stolen. If not, then the split point is moved between tail and split. The memory fence ensures that the new split point is globally visible before reading tail. Once the new split point is globally visible, no tasks can be stolen beyond the new split point. Therefore, we only need to check once whether more tasks are stolen. If at that point all remaining tasks are stolen, then allstolen is set and shrink_shared returns True. If not, then if only some tasks are stolen beyond the new split point, the split point is moved again. Finally, shrink_shared returns False.

3.3 Extensions

There are several possible extensions to the work-stealing deque.

Resizing. Our work-stealing deque uses a fixed-size array. Given that virtual memory is several orders of magnitude larger than real memory and the ability of modern operating systems to allocate only used pages, we can avoid overflows by allocating an amount of virtual memory much higher than required. The deque could be extended for resizing, for example using linked lists of arrays, but we feel this is unnecessary in practice.

Steal-multiple strategies. One extension to work-stealing is the policy to steal more than one task at the same time, e.g., stealing half the tasks in the deque, which has been argued to be beneficial in the context of irregular graph applications [11,6]. This is easily implemented by modifying line 5 to steal multiple tasks, and executing the stolen tasks in reverse order (last one first). However, in experiments on a single NUMA machine, this did not improve performance.

Other memory models. The algorithm in Figure 4 is designed for the x86 TSO memory model, which only allows reordering loads before stores. Weaker memory models may for example allow reordering stores. Assuming that reordering only takes place on independent operations, we believe no additional memory fences are required in Figure 4 to ensure correctness. Memory fences are however required in Figure 1 to ensure that `result` is set before `done`.

4 Evaluation

We implemented Lace, a C library that provides a work-stealing framework similar to Wool and Cilk. The library creates one POSIX thread (*pthread*) for each available core. Our implementation is NUMA-aware, i.e., all pthreads are pinned to a NUMA domain and their program stack and the deque structures for each worker are allocated on the same NUMA domain as the worker.

We evaluate Lace using several benchmarks compared to the work-stealing framework Wool [8] using the classic leapfrogging strategy. This version of Wool has a dynamic split point and does not use locking. We compare the performance of Lace and Wool, for two reasons. Our implementation resembles the implementation of Wool, making a comparison easier. Also, [8] and [14] show that Wool is efficient compared to Cilk++, OpenMP and the Intel TBB framework, with a slight advantage for Wool. We also compare our algorithm to the receiver-initiated version of the private deque of Acar et al. [1], using the alternative `acquire` function, which we implemented in the Lace framework.

4.1 Benchmarks

For all benchmarks, we use the smallest possible granularity and do not use sequential cut-off points, since we are interested in measuring the overhead of the work-stealing algorithm. Using a larger granularity and sequential cut-off points may result in better scalability for some benchmarks.

Fibonacci. For a positive integer N, calculate the Fibonacci number by calculating the Fibonacci numbers $N-1$ and $N-2$ recursively and add the results. This benchmark generates a skewed task tree and is commonly used to benchmark work-stealing algorithms, since the actual work per task is minimal. Number of tasks: 20,365,011,073 (`fib 50`).

Queens. For a positive integer N, calculate the number of solutions for placing N queens on a $N \times N$ chessboard so that no two queens attack each other. Each task at depth i spawns up to N new tasks, one for every correct board after placing a queen on row i. Number of tasks: 171,129,071 (`queens 15`).

Unbalanced Tree Search. This benchmark is designed by Olivier et al. to evaluate the performance for parallel applications requiring dynamic load balancing. The algorithm uses the SHA-1 algorithm to generate geometric and binomial trees. The generated binomial trees (T3L) have unpredictable subtree sizes and depths and are optimal adversaries for load balancing strategies [13]. The geometric tree (T2L) appears to be easy to balance in practice. Number of tasks: 96,793,509 (`uts T2L`) and 111,345,630 (`uts T3L`).

Rectangular matrix multiplication. Given N, compute the product of two random rectangular $N \times N$ matrices A and B. We use the `matmul` algorithm from the Cilk benchmark set. Number of tasks: 3,595,117 (`matmul 4096`).

Benchmark	Lace		Speedup	
	T_1	T_{48}	T_S/T_{48}	T_1/T_{48}
fib `50`	144	4.13	34.5	34.9
uts `T2L`	86.0	1.81	46.1	47.4
uts `T3L`	44.2	2.23	18.7	19.9
queens `15`	602	12.63	42.2	47.7
matmul `4096`	781	16.46	47.0	47.5
	Private deque			
fib `50`	208	5.22	23.2	39.8
uts `T2L`	86.1	1.83	45.7	47.0
uts `T3L`	44.8	2.55	17.3	17.5
queens `15`	541	11.34	43.3	47.7
matmul `4096`	774	16.34	47.3	47.4

Benchmark	Wool		Speedup	
	T_1	T_{48}	T_S/T_{48}	T_1/T_{48}
fib `50`	185	4.38	34.1	42.2
uts `T2L`	85.1	2.00	42.5	42.5
uts `T3L`	44.3	2.12	19.4	20.9
queens `15`	539	11.23	47.5	48.0
matmul `4096`	780	16.40	47.2	47.5
	T_S	Sequential		
fib `50`	149.2	-	-	-
uts `T2L`	84.5	-	-	-
uts `T3L`	43.11	-	-	-
queens `15`	533	-	-	-
matmul `4096`	773	-	-	-

Fig. 5. Averages of running times (seconds) for all benchmarks. Speedups are calculated relative to both the time of the sequential version (T_S) and the parallel version with one worker (T_1). Each T_{48} data point is the average of 50 measurements. Each T_1/T_S data point is the average of 20 measurements.

4.2 Results

Our test machine has four twelve-core AMD Opteron 6168 processors. The system has 128 GB of RAM and runs Scientific Linux 6.0 with kernel version 2.6.32. We considered using less than 48 cores to reduce the effects of operating system interference, but we did not see significant effects in practice. We compiled the benchmarks using gcc 4.7.2 with flag `-O3`.

See Figure 5 for the results of the benchmark set. Each T_{48} data point is the average of 50 measurements. Each T_1 and T_S data point is the average of 20 measurements. This resulted in measurements with three significant digits. In general, Figure 5 shows similar performance for all three algorithms. The three benchmarks `uts T2L`, `queens` and `matmul` are trivial to parallelize and have no extra overhead with 48 workers, i.e., $T_1/T_{48} \approx 48$.

Comparing T_S and T_1 for all benchmarks, we see that the overhead of work-stealing is small for all three work-stealing algorithms, with the exception of the `fib` benchmark. For benchmark `fib` with our algorithm, $T_1 < T_S$, which appears to be related to compiler optimizations. During implementation, we observed that variation in T_1 is often related to code generation by the compiler. In some cases, removing unused variables and other minor changes even increased T_1 by up to 20%. It is therefore difficult to draw strong conclusions regarding the overhead of the algorithms, except that it is small compared to the sequential program.

We measured the runtimes of `fib` and `uts T3L` using 4, 8, 16, 24, 32 and 40 workers to obtain the speedup graph in Figure 6. This graph suggests that the `fib` benchmark scales well and that similar results may be obtained using a higher number of processors in the future. The scalability of the `uts T3L` benchmark appears to be limited after 16 workers. We discuss this benchmark below.

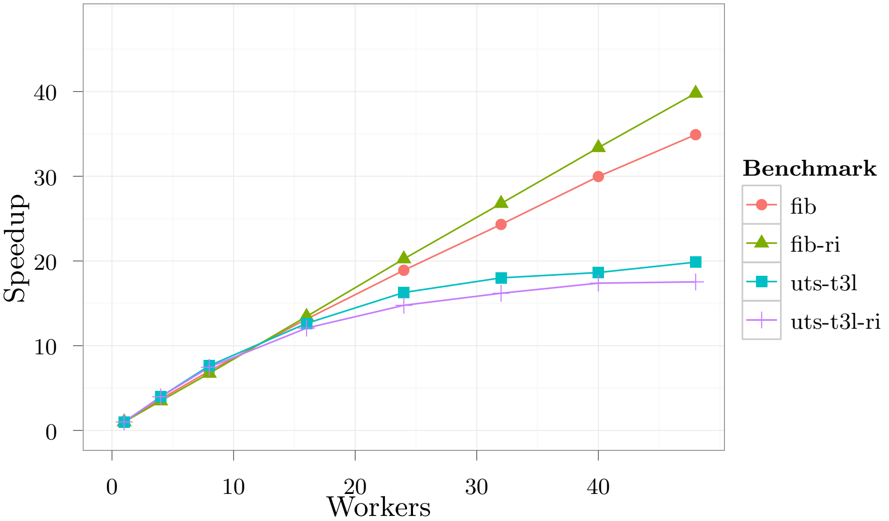

Fig. 6. Absolute speedup graph (T_1/T_N) of the `fib` and `uts T3L` benchmarks using Lace with our algorithm and Lace with the private deque receiver initiated (`-ri`) algorithm. Each data point is based on the average of 20 measurements.

Benchm.	#steals	#leaps	#grows	#shrinks
fib `50`	865	50,569	70,789	97,750
uts `T2L`	4,554	82,440	72,222	57,701
uts `T3L`	158,566	4,443,432	2,173,006	846,509
queens `15`	1,964	6,053	5,694	6,618
matmul `4096`	2,492	12,456	13,081	9,911

Fig. 7. The average total number of steals, leaps, grows and shrinks over 7 runs with 48 workers

Algo.	T_1	T_{48}	T_1/T_{48}
Lace	44.26	1.154	38.3
Private	44.83	1.240	36.2
Wool	44.27	1.172	37.8

Fig. 8. Averages of runtimes (seconds) of `uts T3L` with transitive leapfrogging (Wool) or random stealing (Lace/Private)

We also measured the average number of steals during a parallel run with 48 workers. See Figure 7. We make a distinction between normal stealing when a worker is idle, and leapfrogging when a worker is stalled because of unfinished stolen work. We also measured the amount of split point changes by `grow_shared` and `shrink_shared`. The number of 'grows' indicates how often thieves set `splitreq`. The number of 'shrinks' is equal to the number of memory fences. In the `uts T3L` benchmark, the high number of leaps and split point changes may indicate that the stolen subtrees were relatively small.

4.3 Extending Leapfrogging

Benchmark `uts T3L` appears to be a good adversary for all three algorithms. This is partially related to the leapfrogging strategy, which forces workers that wait for the result of stolen tasks to steal from the thief. This strategy can result in chains of thieves waiting for work to trickle down the chain. For example, when worker 2 steals from worker 1, worker 1 will only steal from worker 2. If worker 3 steals from worker 2 and worker 4 steals from worker 3, new tasks will be generated by worker 4 and stolen by worker 3 first. Worker 3 then generates new work which can be stolen by workers 2 and 4. Worker 1 only acquires new work if the subtree stolen by worker 4 is large enough. The updated version of Wool [8] implements an extension to leapfrogging, called transitive leapfrogging[2]. Transitive leapfrogging enables workers to steal from the thief of the thief, i.e., still steal subtasks of the original stolen task.

We extended Lace to steal from a random worker whenever the thief has no available work to steal. See Figure 8 for the results of this extension, compared to transitive leapfrogging in Wool. Compared to the results in Figure 5, all benchmarks now have reasonable speedups, improving from a speedup of 20x to a speedup of 36x with 48 workers.

Our extension has the disadvantage of not guaranteeing the upper bound on the stack size that leapfrogging and transitive leapfrogging does. It is, however, very simple to implement, while resulting in similar performance. We measured the peak stack depth with the `uts T3L` benchmark for all 48 workers. We observed an increase from a peak stack depth of 6500-12500 tasks with normal leapfrogging to 17000-21000 tasks with the random stealing extension. Since every task descriptor for `uts T3L` is 64 bytes large (including padding), this strategy required at most 1 extra megabyte per worker for `uts T3L`. We also observed that the number of 'grows' decreased by 50%.

5 Conclusion

In this paper, we presented a new non-blocking split deque for work-stealing. Our design has the advantage that it does not require memory fences for local deque operations, except when reclaiming tasks from the shared portion of the deque. Furthermore, we implemented this deque in a C library called Lace, which has an interface similar to Wool. This framework has the advantage of a small source code footprint. We also implemented the receiver-initiated version of the private deque algorithm described by Acar et al. in Lace.

Our experiments show that our algorithm is competitive with Wool and with the private deque algorithm. We gain near optimal speedup for several benchmarks, with very limited overhead compared to the sequential program. Extending leapfrogging with random stealing greatly improves scalability for the `uts T3L` benchmark.

Several open questions remain. When growing the shared deque, the new split point is the average of `split` and `head`, and when shrinking the shared deque, the new split point is the average of `tail` and `head`. It is unknown whether more optimal strategies exist. A limitation of our approach is that tasks can only be

[2] This feature is documented in the distribution of Wool version 0.1.5alpha, which is currently available at `http://www.sics.se/~kff/wool/`

stolen at the tail of the deque. This limits work-stealing strategies. Designs that allow stealing any task may be useful for some applications. Our benchmarks all consist of uniformly small tasks. Benchmarking on larger or irregular sized tasks may be disadvantageous for the private deque algorithm, since it requires owner cooperation on every steal. Finally, we performed our experiments on a single NUMA machine. On such machines, communication costs are low compared to distributed systems. It may be interesting to compare the work-stealing algorithms on a cluster of computers using a shared-memory abstraction. Especially steal-multiple strategies may be more beneficial when communication is more expensive.

References

1. Acar, U.A., Charguéraud, A., Rainey, M.: Scheduling parallel programs by work stealing with private deques. In: PPOPP, pp. 219–228. ACM (2013)
2. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread Scheduling for Multiprogrammed Multiprocessors. Theory Comput. Syst. 34(2), 115–144 (2001)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. J. Parallel Distrib. Comput. 37(1), 55–69 (1996)
4. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. In: FOCS, pp. 356–368. IEEE Computer Society (1994)
5. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: SPAA, pp. 21–28. ACM (2005)
6. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: SC. ACM (2009)
7. Faxén, K.F.: Wool-A work stealing library. SIGARCH Computer Architecture News 36(5), 93–100 (2008)
8. Faxén, K.F.: Efficient Work Stealing for Fine Grained Parallelism. In: ICPP, pp. 313–322. IEEE Computer Society (2010)
9. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: PLDI, pp. 212–223. ACM (1998)
10. Hendler, D., Lev, Y., Moir, M., Shavit, N.: A dynamic-sized nonblocking work stealing deque. Distributed Computing 18(3), 189–207 (2006)
11. Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: PODC, pp. 280–289. ACM (2002)
12. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent work stealing. In: PPOPP, pp. 45–54. ACM (2009)
13. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An Unbalanced Tree Search Benchmark. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
14. Podobas, A., Brorsson, M.: A comparison of some recent task-based parallel programming models. In: Programmability Issues for Multi-Core Computers (MULTIPROG 2010), Pisa (January 2010)
15. Sundell, H., Tsigas, P.: Brushing the Locks out of the Fur: A Lock-Free Work Stealing Library Based on Wool. In: The Second Swedish Workshop on Multi-Core Computing MCC 2009. University of Borås. School of Business and Informatics (2009)
16. Wagner, D.B., Calder, B.: Leapfrogging: A portable technique for implementing efficient futures. In: PPOPP, pp. 208–217. ACM (1993)