Dynamic Circular Work-Stealing Deque

David Chase
Sun Microsystems Laboratories
Mailstop UBUR02-311
1 Network Drive
Burlington, MA 01803, USA
dr2chase@sun.com

Yossi Lev
Brown University & Sun Microsystems Laboratories
Mailstop UBUR02-311
1 Network Drive
Burlington, MA 01803, USA
yosef.lev@sun.com

ABSTRACT

The non-blocking work-stealing algorithm of Arora, Blumofe, and Plaxton (henceforth *ABP work-stealing*) is on its way to becoming the multiprocessor load balancing technology of choice in both industry and academia. This highly efficient scheme is based on a collection of array-based double-ended queues (deques) with low cost synchronization among local and stealing processes. Unfortunately, the algorithm's synchronization protocol is strongly based on the use of fixed size arrays, which are prone to overflows, especially in the multiprogrammed environments for which they are designed. We present a work-stealing deque that does not have the overflow problem.

The only ABP-style work-stealing algorithm that eliminates the overflow problem is the list-based one presented by Hendler, Lev and Shavit. Their algorithm indeed deals with the overflow problem, but it is complicated, and introduces a trade-off between the space and time complexity, due to the extra work required to maintain the list.

Our new algorithm presents a simple lock-free work-stealing deque, which stores the elements in a cyclic array that can grow when it overflows. The algorithm has no limit other than integer overflow (and the system's memory size) on the number of elements that may be on the deque, and the total memory required is linear in the number of elements in the deque.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*load balancing, lock-free*; E.1 [**Data**]: Data Structures

General Terms

Algorithms

Keywords

work stealing, load balancing, lock-free, deque

SPAA'05, July 18–20, 2005, Las Vegas, Nevada, USA.

ACM 1-58113-986-1/05/0007

1. INTRODUCTION

The ABP work-stealing algorithm of Arora, Blumofe, and Plaxton [2] has been gaining popularity as the multiprocessor load-balancing technology of choice in both industry and academia [2, 1, 4, 9]. The scheme implements a provably efficient work-stealing paradigm due to Blumofe and Leiserson [3] that allows each process to maintain a local work deque,[1] and steal an item from others if its deque becomes empty. The deque's owner process pushes and pops local work to and from the deque's bottom end. To minimize synchronization overhead for the deque's owner, stolen elements are taken from the top end of the deque. No elements are added to the top end of the deque. An ABP deque thus presents three methods in its interface:

- `pushBottom(Object o)`:
  Pushes *o* onto the bottom of the deque.
- `Object popBottom()`:
  Pops an object from the bottom of the deque if the deque is not empty, otherwise returns *Empty*.
- `Object steal()`:
  If the deque is empty, returns *Empty*. Otherwise, returns the element successfully stolen from the top of the deque, or returns *Abort* if this process loses a race with another process to steal the topmost element[2].

Note that `pushBottom` and `popBottom` operations are invoked only by the deque's owner.

Unfortunately, the use of fixed size arrays introduces an inefficient memory-size/robustness tradeoff: for n processes and total allocated memory size m, one can tolerate at most $\frac{m}{n}$ items in a deque. Using cyclic arrays, or the reset-on-empty heuristic presented in the original ABP algorithm,[3] reduces the chance of overflow but does not eliminate it. The list-based work-stealing deque algorithm presented by

[1]Actually, the work-stealing algorithm uses a *work-stealing deque*, which is like a deque [8] except that only one process can access one end of the queue (the "bottom"), and only Pop operations can be invoked on the other end (the "top"). For brevity, we refer to the data structure as a deque in the remainder of the paper.

[2]In our implementation, as we describe, *Abort* is also returned if a `steal` operation lost a race with an array memory reclamation caused by a concurrent `popBottom` operation.

[3]The reset-on-empty heuristic resets `top` and `bottom` to point to the beginning of the array whenever the deque becomes empty. It was used by the original ABP algorithm to make overflow scenarios less frequent.

Hendler, Lev and Shavit[5] uses a list of small arrays to eliminate the overflow problem. However, it is relatively complicated, does not use cyclic arrays (and therefore wastes some memory), and introduces a trade-off between its time and space complexity due to the extra work required for the list's maintenance.

This paper presents the first work-stealing deque that uses a *dynamic-cyclic-array*. The algorithm is remarkably simple, efficient, and completely eliminates the overflow problem. Also, unlike all previous algorithms, the `top` and `bottom` fields are used merely to indicate the two ends of the deque—no tag field is required to eliminate the ABA problem. Therefore, the only restriction on the deque size is due to integer overflow. A 64-bit integer is large enough to accommodate 64 years of pushes, pops, and steals executing at a rate of 4 billion operations per second, so it appears that this will not be a problem in practice.

The rest of the paper is organized as follows: Section 2 presents the simplest version of the algorithm, which requires a garbage collector to reclaim unused buffers. Section 3 adds to the algorithm the ability to shrink the array if the deque retracts from its maximum size, and by that improves its space complexity. Section 4 presents the final version of the algorithm, that works with a shared pool of buffers and does not depend upon a garbage collector for buffer reclamation. Section 5 presents some preliminary experiments results, and we summarize in Section 6.

2. THE BASIC ALGORITHM

2.1 Overview

The deque is implemented using a cyclic array, together with two indexes: `top` and `bottom`, indicating the two ends of the deque. Specifically, the `bottom` index indicates the next available slot in the array where the next new element is pushed, and is incremented on every `pushBottom` operation. The `top` index indicates the topmost element in the deque (if there is any), and is incremented on every `steal` operation. If `bottom` is less than or equal to `top`, the deque is empty.

Since the algorithm uses a cyclic array it makes more efficient use of the array, and does not need the reset-on-empty heuristic used in the original ABP algorithm. If a `pushBottom` operation discovers that the current circular array is full, it enlarges it by copying the deque's elements into a bigger array, and pushes the new element into the new enlarged array. The deque elements stored in the circular array are indexed modulo its size, and therefore when moving the elements into a bigger array, there is no need to update `top` or `bottom` although the actual array indexes where the elements are stored might change. Since the only operation that modifies `top` is `steal`, `top` is never decremented, and there is no need for a tag field as in all previous work-stealing algorithms.[4]

2.2 The deque operations

The pseudocode for the `pushBottom`, `steal` and `popBottom` operations appears in Figures 1, 2 and 3 respectively. The `steal` and `popBottom` methods are similar to their counterparts in the original ABP algorithm. The main difference is that the new algorithm does not have a tag field in the `top` variable—instead, it maintains the property that `top` is never decremented. The deque object also has a private `casTop` method (depicted in Figure 1), which receives two values for `top`, *old* and *new*, and atomically modifies `top` to the new value if and only if it has the old value. In practice this method is implemented using the Compare-And-Swap (CAS) instruction, which is widely available on current machines.

```
public class CircularWSDeque {
    public final static Object Empty = new Object();
    public final static Object Abort = new Object();

    private final static int LogInitialSize = /* Log of
                              the initial array size */;

    private volatile long bottom = 0;
    private volatile long top = 0;
    private volatile CircularArray activeArray =
                    new CircularArray(LogInitialSize);

    private boolean casTop(long oldVal, long newVal) {
        boolean preCond;
        atomically {
            preCond = (top==oldVal);
            if (preCond)
                top=newVal;
        }
        return preCond;
    }

    public void pushBottom(Object o) {
1       long b = this.bottom;
2       long t = this.top;
3       CircularArray a = this.activeArray;
4       long size = b - t;
5       if (size >= a.size()-1) {
6           a = a.grow(b, t);
7           this.activeArray = a;
        }
8       a.put(b, o);
9       bottom = b+1;
10  }
```

Figure 1: The `pushBottom` operation

```
    public Object steal() {
11      long t = this.top;
12      long b = this.bottom;
13      CircularArray a = this.activeArray;
14      long size = b - t;
15      if (size <= 0) return Empty;
16      Object o = a.get(t);
17      if (! casTop(t, t+1))
18          return Abort;
19      return o;
    }
```

Figure 2: The `steal` operation

The `pushBottom` operation: As in the original algorithm, the `pushBottom` operation inserts the pushed entry to the deque simply by writing it in the location specified by `bottom`, and then incrementing `bottom` by 1. In our algorithm, however, the `pushBottom` operation is also responsible for enlarging the array if an element is pushed into an already-full

[4]We assume that `top` never overflows. As explained in the previous section, with a 64-bit integer implementation this is a very reasonable assumption.

```
    public Object popBottom() {
20      long b = this.bottom;
21      CircularArray a = this.activeArray;
22      b = b - 1;
23      this.bottom = b;
24      long t = this.top;
25      long size = b - t;
26      if (size < 0) {
27          bottom = t;
28          return Empty;
        }
29      Object o = a.get(b);
30      if (size > 0)
31          return o;
32      if (! casTop(t, t+1))
33          o = Empty;
34      this.bottom = t+1;
35      return o;
    }
```

Figure 3: The popBottom operation

array. Whether the array is enlarged or not, the abstract `pushBottom` operation takes place when `bottom` is updated at Line 9.

To check whether the current array is full, the operation subtracts the value of `top` from `bottom` (which gives the number of elements that were in the deque when `top` was read), and compares it to the size of the array. For memory reclamation reasons described in Section 4.1, the `pushBottom` operation always leaves one array cell unused. If necessary, the `pushBottom` operation uses the `grow` method (of the `CircularArray` class) to enlarge the current array.

The growable circular array: The simplest implementation of a growable circular array is a power-of-two-sized array that grows by doubling its size. When the array is full, a new doubled size array is allocated, and the elements are copied from the old array to the new one. The pseudocode for the `CircularArray` class appears in Figure 4. Note that since the elements are indexed modulo the array size, the actual array indexes where the elements are stored might change when copying from one array to another, but the value of `top` and `bottom` remains the same.

The `steal` operation: As in the original algorithm, the `steal` method begins by reading `top` and `bottom`, and checking whether the deque is empty by comparing these values. If the deque is not empty, it reads the element stored in the `top` position of the cyclic array, and tries to increment `top` using a CAS operation. If the CAS fails, it implies that a concurrent `steal` operation successfully removed an element from the deque, so the operation returns the Abort value; otherwise it returns the element read right before the successful CAS operation. Since the algorithm uses a cyclic array, it is important to read the element from the array before we do the CAS, because after the CAS completes, this location may be refilled with a new value by a concurrent `pushBottom` operation.

A successful CAS is the point at which the abstract `steal` operation takes place. Note that because `top` is read before `bottom`, it is guaranteed that the values read represent a consistent view of the memory. Specifically, it implies that `bottom` and `top` indeed had their observed values when `bottom` was read at Line 12. A subtle case may arise, however, if the deque is emptied by a concurrent `popBottom` operation after `bottom` is read, but before the CAS is executed. For that reason, as we describe later, any `popBottom` operation that empties the deque tries to modify `top` (using a CAS operation), to guarantee that no concurrent `steal` operation will also returns the deque's last entry.

The `popBottom` operation: As in the original algorithm, the owner can pop inexpensively (without using a CAS operation) provided that doing so does not cause the deque to become empty. If the deque was already empty, then it simply resets it to a canonical empty state (`bottom` = `top`) and returns the Empty value (Lines 26-28). If the deque becomes empty, then the owner must perform a CAS on `top` to see if it won or lost any race (with a concurrent `steal` operation) to pop the last item. Unlike the original ABP algorithm, the new algorithm performs the CAS on the value of the `top` index and not on a tag value (note that incrementing `top` when the deque is empty leaves the deque in an empty state). Right after the CAS operation, whether it succeeds or not, the value of `top` is $t + 1$ (note that if the CAS fails, then some concurrent `steal` operation updated `top` to that value). Therefore the deque is empty, and the operation completes by storing $t + 1$ in `bottom` (by that, resetting the deque to a canonical empty state). In any case that the `popBottom` operation does not return the empty value, the abstract `popBottom` operation takes place when `bottom` is updated at Line 23.

```
class CircularArray {
  private int log_size;
  private Object[] segment;

  CircularArray(int log_size) {
    this.log_size = log_size;
    this.segment = new Object[1<<this.log_size];
  }
  long size() {
      return 1<<this.log_size;
  }
  Object get(long i) {
      return this.segment[i % size()];
  }
  void put(long i, Object o) {
      this.segment[i % size()] = o;
  }
  CircularArray grow(long b, long t) {
      CircularArray a =
             new CircularArray(this.log_size+1);
      for (long i=t; i<b; i++) {
          a.put(i, this.get(i));
      }
      return a;
  }
}
```

Figure 4: Growable Circular Array

2.3 Avoid top accesses in pushBottom

Unlike the original ABP algorithm, the new algorithm requires reading `top` on every execution of the `pushBottom` operation. This may result in more data-cache misses compared to the original algorithm (recall that unlike `bottom`, `top` is modified by *all* processes).

The frequency of accesses to the `top` variable can be significantly reduced, by keeping a local upper bound on the size of the deque, and only read `top` when the upper bound

indicates that an array expansion may be necessary. Such a local upper bound can be easily achieved by saving the last value of `top` read in a local variable, and using this variable to compute the size of the deque (instead of the real value of `top`). Because `top` is never decremented, the real size of the deque can only be smaller than the one calculated using this local variable.

3. SHRINKING AFTER GROWTH

One disadvantage of the algorithm as presented is that it does not shrink the array as the deque retreats from its maximum. That means that the memory used by the deque is a constant factor times its *maximum size*, which might result in a big waste of memory.

Shrinking the array is no harder than growing the array; it only requires that the algorithm check against a minimum use fraction of the current array when performing a `popBottom` operation[5]. The code for the `popBottom` operation with the possible shrinking operation appears in Figures 5 and 6. As illustrated by the code, Line 31 was modified to call the `perhapsShrink` method just before returning the popped value. The `perhapsShrink` method shrinks the array if the number of elements in the deque is less than some fraction $1/K$ of the array size, where $K \geq 3$. We omit the code for the `CircularArray`'s shrink method since it is almost identical to the code of this class's `grow` method.

Finally, note that the `perhapsShrink` method is independent of the `popBottom` operation, and therefore can be invoked by the deque's owner on other occasions (for example after a `pushBottom` operation).

```
    public Object popBottom() {
20      long b = this.bottom;
21      CircularArray a = this.activeArray;
22      b = b - 1;
23      this.bottom = b;
24      long t = this.top;
25      long size = b - t;
26      if (size < 0) {
27          bottom = t;
28          return Empty;
        }
29      Object o = a.get(b);
30      if (size > 0) {
31.1        perhapsShrink(b, t);
31.2        return o;
        }
32      if (!casTop(t, t+1))
33          o = Empty;
34      this.bottom = t+1;
35      return o;
}
```

Figure 5: The `popBottom` operation with shrinking

3.1 Shrinking without copying

The simplest way to shrink back to a smaller array is similar to the way we grow it: Allocate a new smaller array, and copy the data from the big array to the smaller one.

We can save the allocation time, however, if whenever we extend an array, we retain a reference from the bigger array to the smaller one. If each array has a reference to the smaller array from which it was extended, then the garbage collector cannot deallocate all the arrays that precede the current active one, and the algorithm can reuse these arrays when shrinking.

[5]With the current implementation, this fraction must be strictly less than $\frac{1}{2}$, to guarantee that the deque elements could fit into the smaller array while leaving one array cell unused.

```
    void perhapsShrink(long b, long t) {
36      CircularArray a = this.activeArray;
37      if (b-t < a.size()/K) {  // K constant >= 3
38          CircularArray aa = a.shrink(b, t);
39          this.activeArray = aa;
        }
    }
```

Figure 6: Simple conditional shrinking

Keeping the references to the smaller arrays not only saves the allocation time, it can also save some of the copying work: when the algorithm shrinks back from the big array to its previous smaller array, only the elements that were modified while the bigger array was active need to be copied (because the smaller array was not deallocated and therefore was not modified while the bigger array was active). This can be accomplished by maintaining a low-water-mark with each array: an integer that indicates the lowest value of `bottom` in which an element was stored while the array was active. When a deque shrinks its array, only the elements stored in indexes greater than or equal to the low water mark of the bigger array are copied. Also, the smaller array's low water mark is updated to the minimum of the larger and smaller array's low water mark values.

Note that the space overhead for referencing all the smaller arrays when growing is relatively low: if we double the array size every time we grow the array, the total overhead is less than the size of the current array.

3.2 Combining multiple shrinks

Sometimes it is useful to combine multiple shrink operations, that is to shrink back not to the previous smaller array, but to one (or more) preceding it. For example, suppose that we had 5 growing operations: $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5$ (here a_i represents an array, and a_{i+1} is a bigger array than a_i), and that on the next `popBottom` operation we find out that almost all the deque elements were stolen, and that the number of elements left is less than some fraction of the size of a_1. In such a case it makes more sense to shrink from a_5 directly to a_1, without going through all the intermediate arrays. Extra caution should be taken, however, when choosing which entries to copy using the low water mark: when copying from a_5 to a_1, the low water mark is the minimum of the low water marks of a_5 and all the intermediate arrays (that is a_4, a_3 and a_2).

4. WORKING WITH A SHARED POOL OF BUFFERS

The deque algorithm presented depends upon a garbage collector to reclaim the unused buffers. For work-stealing algorithms, where each process has its own deque and the maximum amount of memory needed by all deques together can usually be bounded, it is often more suitable to use the shared pool model.

With the shared pool model, the extra available buffers for all the deques are kept in a shared pool. Whenever the deque's owner needs a bigger array, it allocates one (of the

appropriate size) from the pool, and whenever it shrinks to a smaller array and does not need the bigger array anymore, it can return it to the pool. There are two main advantages for the shared pool model: First, it is much less expensive to reclaim and allocate buffers from the shared pool than to allocate them from the heap and use the garbage collector for reclamation. Second, as described in this section, by assuming that the reclaimed buffers are not returned to a global use by the operating system, the deque's owner can reclaim a buffer while there may be still some thieves referencing it (something that the garage collector will not do), which leads to a better use of the allocated space.

4.1 Allocating and reclaiming buffers

Whenever the deque's owner needs a new array it allocates the array's buffer from the shared pool. Reclaiming buffers is trickier: Consider the case in which a `steal` operation executes Line 13, and then a concurrent `popBottom` operation shrinks the array and returns the bigger array's buffer to the shared pool. When the `steal` operation resumes and executes Line 16, it reads and returns an element from the old bigger array which was already reclaimed, and therefore may be used by another deque. This scenario, of course, could not be allowed. Therefore, to return an array's buffer to the shared pool when shrinking back to a smaller one,[6] the deque's owner must have some guarantee that any thieves concurrently referencing that array will be forced to abort and will not return data read from it after it was reclaimed.

One way to do it is by self-stealing: the deque's owner steals an item from its own deque and returns that item instead of an item popped from the bottom of the deque. This solution, however, is problematic because it breaks the LIFO behavior that is normally expected for the deque's owner.

Instead, we can use the property that the algorithm references the array modulo its size, and increment `top` and `bottom` in a way that aborts concurrent `steal` operations, but does not change the content of the deque. For a given array size N, x and $x+N$ both address the same element in the array. Therefore, incrementing `top` and `bottom` by the size of the new array when an array is shrunk or grown preserves the deque's content, but aborts any concurrent thefts. In Section 4.4 we show that modifying `top` and `bottom` as described still allows us to assume that they will not overflow.

The modified `perhapsShrink` and `steal` methods appear in Figures 7 and 8. The new `perhapsShrink` method contains the `bottom` and `top` modifications (Lines 39.2-39.6), right after making the new array active (Line 39.1), and before returning the old array's buffer to the shared pool (Line 39.7). The order in which `bottom` and `top` are modified is important, because while the deque's owner is between these modifications, thieves must not spuriously return a value from an empty deque or report empty when elements are actually in the deque. Because the new `perhapsShrink` method modifies `bottom` before `top`, a new empty deque scenario is added, which is when the difference between `bottom` and `top` is exactly the current array size. Because the `pushBottom` method never fills the array to its maximum capacity, this difference uniquely identifies the new empty scenario, and can be detected by the `steal` method.

The CAS operation at Line 39.5 tries modifying `top`. If it fails, then there must have been a concurrent `steal` operation that modified `top` after it was read by `perhapsShrink` at Line 39.4, at which point the smaller array is already active. This implies that any subsequent CAS by another concurrent `steal` operation that read an element from the old array will fail. Therefore if the CAS at Line 39.5 fails, the `perhapsShrink` method simply returns `bottom` to its original value (Line 39.6), and reclaims the old array buffer.

```
    void perhapsShrink(long b, long t) {
36      CircularArray a = this.activeArray;
37      if (b-t < a.size()/K) { // K constant >= 3
38          CircularArray aa = a.shrink(b, t);
39.1        this.activeArray = aa;
39.2        long ss = aa.size();
39.3        this.bottom = b + ss;
39.4        t = this.top;
39.5        if (! casTop(t, t+ss))
39.6            this.bottom = b;
39.7        a.free();
        }
    }
```

Figure 7: Conditional shrinking in the shared pool model

```
    public Object steal() {
11      long t = this.top;
11.1    CircularArray oldArr = this.activeArray;
12      long b = this.bottom;
13      CircularArray a = this.activeArray;
14      long size = b - t;
15      if (size <= 0) return Empty;
15.1    if ((size % a.size())==0) {
15.2        if (a == oldArr && t == this.top)
15.3            return Empty;
15.4        else return Abort;
        }
16      Object o = a.get(t);
17      if (! casTop(t, t+1))
18          return Abort;
19      return o;
    }
```

Figure 8: Steal operation when shrinking in the shared pool model

The `steal` method is modified in two places: An additional read of the active array is added (Line 11.1), and the emptiness check is extended (Lines 15.1-15.4). The emptiness check must be extended because the `steal` operation may read `bottom` and `top` values that correspond to a deque in the "intermediate state" between the `bottom` and `top` updates of a shrink operation. These values may indicate an empty deque although the difference between them is possitive. Therefore, instead of only checking whether the difference between `bottom` and `top` is non-positive, the modified `steal` method also checks whether this difference *modulo the array size* is 0 (Line 15.1), for the array reference read at Line 13.

If the test at Line 15.1 succeeds, and the values read for `top`, `bottom`, and `activeArray` at Lines 11,12 and 13, correspond to some state of the memory, then this state indicates

[6] We assume that we do not reclaim the buffers of the smaller arrays when growing to bigger ones (for a faster shrink operation as described in Section 3.1). It is straightforward, however, to use the same solution for reclaiming buffers also when growing.

an empty deque. To check whether these values indeed correspond to an actual state of the memory, we use the additional `activeArray` reference read at Line 11.1. It can be shown that if the values of both `top` and `activeArray` were not modified from before reading `bottom` at Line 12 until right after reading `activeArray` at Line 13, then the values read correspond to the memory state during the read of `bottom` at Line 12. Therefore, if the test at Line 15.2 succeeds, the method returns Empty;[7] otherwise, some concurrent operation successfully popped an element from the deque,[8] and the method returns Abort.

Note that the deque's reference to the array is the first to be updated by the `perhapsShrink` method (Line 39.1), but the last to be read by the `steal` operation (Line 13), thereby guaranteeing that if the `steal` operation sees an intermediate state, it sees the new (smaller) array. In the full version of the paper, we prove that the shared pool version of the algorithm implements a linearizable ABP-style deque that cannot overflow. Here we only informally show that using the above `perhapsShrink` and `steal` methods, prevents a `steal` operation from returning an entry read from an array after it was returned to the shared pool.

DEFINITION 1. Array States:

- *A* live *array is an array whose buffer does not reside in the shared pool.*
- *The* active array *of a deque is the array referenced by the `activeArray` data member of the deque object.*

LEMMA 1. *The `steal` method never returns an entry that was read from an array which is not live.*

PROOF. The only statement that returns buffers to the shared pool is Line 39.7 in the `perhapsShrink` method, which is executed only by the deque's owner. Also, the deque's owner is the only process that may activate or de-activate an array (that is, replace the active array of the deque). Since the `perhapsShrink` method set the new smaller array to be the active array before executing Line 39.7, then an active array is never deallocated.

The `steal` operation returns an entry only if the CAS statement at Line 17 successfully writes the $t + 1$ value to `top`, in which case it returns the entry read at Line 16. The `steal` operation reads the deque's active array at Line 13, and therefore the array is live at that point. Thus, if the `steal` operation reads an entry from a non-live array, it must be the case that a concurrent `perhapsShrink` operation:

1. Executes Line 39.7 before the `steal` operation executes Line 16, and
2. Executes Line 39.1 after the `steal` operation executed Line 13.

This implies that Line 39.5 is executed after the `steal` operation reads `top` at Line 11, but before it executes Line 17. If the CAS at Line 39.5 succeeds, then the CAS at Line 17 fails. Otherwise, there must have been a concurrent `steal` operation that modified `top` after the `perhapsShrink` executed Line 39.4, which also guarantees that the CAS at Line 17 fails. We can conclude then that if a `steal` operation read an entry from a non-live array, then the CAS operation at Line 17 fails, which implies that the entry is not returned by that operation. □

[7] In the correctness proof we show that we do not need to worry about the ABA problem on the `activeArray` value.
[8] Assuming the `perhapsShrink` method is called only for a successful `popBottom` operation.

4.2 Lock freedom

The algorithm we presented is clearly wait-free: the only loop in the algorithm is the one that copies the data from one array to another, and it is guaranteed to be finite (linear in the length of the array from which we copy).

A more interesting property, which we formally prove in the full version of the paper, is that even if the `steal` operation retries every time it returns Abort, the algorithm is lock-free. Informally, this is correct because a `steal` operation only returns Abort when either `top` is modified by a concurrent `pop` operation (either a `popBottom` or a `steal` operation), or that `top` or `activeArray` are modified by a `perhapsShrink` operation. Note that neither a `pop` nor a `perhapsShrink` operation can cause more than 2 `steal` operations by the same process to return Abort. Therefore, if a `steal` operation by a process returns Abort infinitely often, then there must be other concurrent operations that complete successfully.

4.3 Deallocating memory from the shared pool

In case the shared pool runs out of buffers, it is possible to allocate more memory to it. Freeing memory from the shared pool is a more sensitive issue, especially in a strongly typed language, because reuse of the freed buffers can cause differently-typed data to be stored into these buffers. This may cause what is locally an actual typing error (between lines 16 and 19), because there might be some thieves that still reference the array that was stored in this buffer (even in non-strongly typed languages, releasing memory to the operating system and then referencing it may cause a run-time error). Therefore in order to safely free an array's buffer from the shared pool, we must first ensure that no thief is referencing that array. This is done automatically by a garbage collector, if one exists. Otherwise, other mechanisms should be used, like the Pass the Buck algorithm by Herlihy, Luchangco and Moir [7]. It is important to note, however, that freeing memory from the shared pool is a rare operation (usually it is not needed at all), and therefore even using a relatively expansive algorithm (like a garbage collector) should not significantly hurt the performance of the algorithm.

4.4 Analysis

Shrinking in the shared pool model consumes deque indices more rapidly than the earlier form of the algorithm, because every time the array is shrunk, `top` and `bottom` are incremented by the size of the smaller array. However, because of the policy of shrinking only at a particular utilization, the consumption of indices is still linear in the number of operations. Indices are consumed fastest when the deque oscillates between containing $N/2$ and N/K elements, where N is the capacity of the larger of two arrays and $1/K$ is the utilization fraction under which we shrink the array. Each time the deque grows to $N/2$ elements, it expands out of the small array; each time it reduces to N/K elements, it shrinks, and the `top` and `bottom` indices are bumped forward by $N/2$. The number of operations in a grow-shrink

cycle is $2N \times (1/2 - 1/K)$. Growth of `top` is maximized if the operations that remove the elements from the deque are `steal`s. The growth in one worst-case grow-shrink cycle is `top` is $N \times (1/2 - 1/K) + N/2$. Dividing, cancelling, and simplifying produces a worst-case growth of $(K-1)/(K-2)$ indices per operation, which is small given that $K \geq 3$.

If K is 3, then the amortized index consumption of each operation is no larger than 2. A 64-bit index will not overflow until at least 2^{63} operations are performed. If operations occur at a rate of 2^{32} per second, this provides at least 2^{31} seconds of run-time. One year is roughly equal to 2^{25} seconds, so a 64 bit index permits at least 64 years of continuous operation without overflow.

Because the active array in the deque is at least $1/K$ filled, and because the total size of the smaller blocks is at most the size of the active array, the storage overhead of this algorithm is at most $2K$ times the *current deque size*.[9]

5. PERFORMANCE

We evaluated the performance of the new dynamic circular work-stealing algorithm in comparison to the original fixed-array ABP work-stealing algorithm. We implemented both algorithms in `C++`, and used a simple shared pool algorithm that allocates and frees a buffer with a single CAS instruction.

The benchmark we ran simulates load balancing of a general computation by building the DAG corresponding to the computation [3], as follows: Initially a single deque contains a single node representing the first work item of the computation. Processes pop nodes from their own deques, and steal nodes from other deques if their own deque is empty. Each time a process pops a node from a deque, it generates up to `B` child nodes, and pushes them into its deque (`B` represents the maximum *branch* of the DAG, and it is a configurable parameter). The number of child nodes generated for a node is randomly chosen with probability that is inversely proportional to the depth of that node in the DAG. The expected number of child nodes for a node of depth d in a DAG of maximum depth `D` is: $B \cdot (1 - \frac{d}{D})$. To get the most accurate measure of the performance difference between the two algorithms, we did not perform any work on a node other than pushing its child nodes to the process's deque.

We ran the benchmark on a 16 node Sun Enterprise™ 6500, an SMP machine formed from 8 boards of two 400MHz UltraSparc® processors, connected by a crossbar UPA switch, and running a Solaris™ 9 operating system. We chose the maximum branch of the DAG to be 13, and the maximum depth to be 10. We used 72-element arrays for the original ABP deques, and our algorithm allocated the deques with an initial size of 64-elements, plus a few 128-element arrays in the shared pool.

Figure 9 presents the throughput of both algorithms, running stand-alone, as a function of the number of processes. As can be seen, both algorithms scale well, and the performance difference is relatively small. Recall that our benchmark does not perform any real computation - it only measures the load-balancing algorithm overhead. In real applications the time spent on the load-balancing algorithm is usually relatively small, and therefore the above performance difference would probably not be noticed.

[9]This upper bound assumes that the `perhapsShrink` method is called by the deque's owner often enough to shrink the deque's array when necessary.

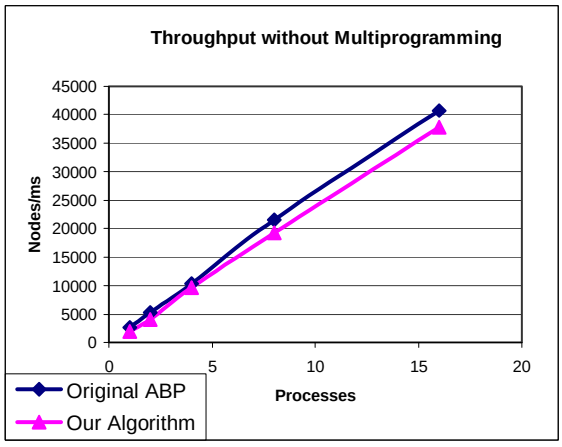

Figure 9: The performance of the two algorithms with no multiprogramming

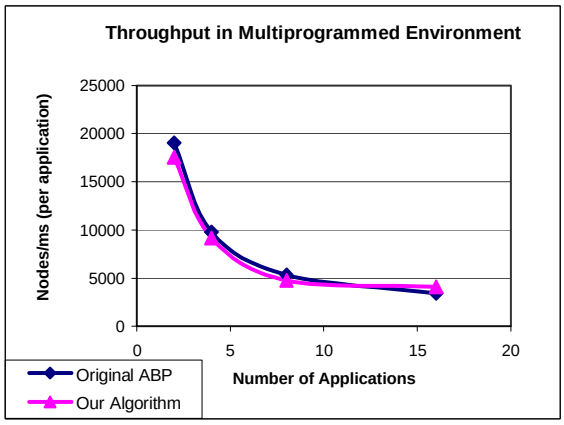

Figure 10: The performance of the two algorithms in a multiprogrammed environment

Next we ran our benchmark in a multiprogrammed fashion by running multiple instances of it in parallel, where each instance is running with 16 processes (as the number of processors on the machine). Figure 10 presents the throughput of an instance as a function the multiprogramming level. As can be seen, there is no significant difference in the performance of the two algorithms.

To compare the stability of the two algorithms, we measured how many of our 64-element arrays overflowed and needed a 128-element array from the shared pool, and noticed that at most one 128-element array is ever needed. On the other hand, the 72-element array allocated for each of the deques in the original ABP algorithm was not always sufficient, and in some cases the algorithm failed to complete due to an overflow of an array. These failures became more frequent as the level of multiprogramming increased. Therefore, for the same amount of array space (notice that $16 \cdot 72 = 128 + (16 \cdot 64)$), we get more robustness with our new algorithm than with the original ABP algorithm, without a noticeable cost in performance.

6. SUMMARY

We present the first ABP-style circular work stealing deque that cannot overflow. Our algorithm is simple, its space complexity is linear in the number of elements in the deque, and it does not require a garbage collector for its memory management. It may be interesting to see how our techniques are applied to other schemes that improve on ABP-work stealing such as the locality-guided work-stealing of Acar, Blelloch and Blumofe [1] or the steal-half algorithm of Hendler and Shavit [6].

7. REFERENCES

[1] Acar, U. A., Blelloch, G. E., and Blumofe, R. D. The data locality of work stealing. In *ACM Symposium on Parallel Algorithms and Architectures* (2000), pp. 1–12.

[2] Arora, N. S., Blumofe, R. D., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems 34*, 2 (2001), 115–144.

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

[4] Flood, C., Detlefs, D., Shavit, N., and Zhang, C. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)* (Monterey, CA, Apr. 2001).

[5] Hendler, D., Lev, Y., and Shavit, N. A dynamic-sized nonblocking work stealing deque. In *Proceedings of the 18th International Symposium on DIStributed Computing* (October 2004), vol. 3274, Springer-Verlag Heidelberg, pp. 188–200. An improved version of this paper is in preparation for journal submission; please contact levyossi@cs.brown.edu.

[6] Hendler, D., and Shavit, N. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing* (2002).

[7] Herlihy, M., Luchangco, V., and Moir, M. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on DIStributed Computing* (January 2002), vol. 2508, Springer-Verlag Heidelberg, pp. 339–353. An improved version of this paper is in preparation for journal submission; please contact authors.

[8] Knuth, D. *The Art of Computer Programming: Fundamental Algorithms*, 2nd ed. Addison-Wesley, 1968.

[9] Leiserson, and Plaat. Programming parallel applications in cilk. *SINEWS: SIAM News 31* (1998).