Distrib. Comput. (2005)
DOI 10.1007/s00446-005-0144-5

SPECIAL ISSUE DISC 04

Danny Hendler · Yossi Lev · Mark Moir · Nir Shavit

A dynamic-sized nonblocking work stealing deque

Received: date / Accepted: date / Published online: 28 December 2005

Abstract The non-blocking work-stealing algorithm of Arora, Blumofe, and Plaxton (hencheforth *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. This is a significant drawback since, apart from memory inefficiency, it means that the size of the deque must be tailored to accommodate the effects of the hard-to-predict level of multiprogramming, and the implementation must include an expensive and application-specific overflow mechanism.

This paper presents the first *dynamic memory* work-stealing algorithm. It is based on a novel way of building non-blocking dynamic-sized work stealing deques by detecting synchronization conflicts based on "pointer-crossing" rather than "gaps between indexes" as in the original ABP algorithm. As we show, the new algorithm dramatically increases robustness and memory efficiency, while causing applications no observable performance penalty. We therefore believe it can replace array-based ABP work stealing deques, eliminating the need for application-specific overflow mechanisms.

This work was conducted while Yossi Lev was a student at Tel Aviv University, and is derived from his MS thesis [1].

D. Hendler
Tel-Aviv University

Y. Lev (✉)
Brown University & Sun Microsystems Laboratories
E-mail: levyossi@cs.brown.edu

M. Moir
Sun Microsystems Laboratories

N. Shavit
Sun Microsystems Laboratories & Tel-Aviv University

Keywords Concurrent programming · Load balancing · Work stealing · Lock-free · Data structures

1 Introduction

Scheduling multithreaded computations on multiprocessor machines is a well-studied problem. To execute multi-threaded computations, the operating system runs a collection of kernel-level *processes*, one per processor, and each of these processes controls the execution of multiple computational *threads* created dynamically by the executed program. The scheduling problem is that of dynamically deciding which thread is to be run by which process at a given time, so as to maximize the utilization of the available computational resources (processors).

Most of today's multiprocessor machines run programs in a *multiprogrammed mode*, where the number of processors used by a computation grows and shrinks over time. In such a mode, each program has its own set of processes, and the operating system chooses in each step which subset of these processes to run, according to the number of processors available for that program at the time. Therefore the scheduling algorithm must be dynamic (as opposed to static): at each step it must schedule threads onto processes, without knowing which of the processes are going to be run.

When a program is executed on a multiprocessor machine, the threads of computation are dynamically generated by the different processes, implying that the scheduling algorithm must have processes load balance the computational *work* in a distributed fashion. The challenge in designing such distributed work scheduling algorithms is that performing a re-balancing, even between a pair of processes, requires the use of costly synchronization operations. Rebalancing operations must therefore be minimized.

Distributed work scheduling algorithms can be classified according to one of two paradigms: *work-sharing* or *work-stealing*. In work-sharing (also known as *load-distribution*), the processes continuously re-distribute work so as to balance the amount of work assigned to each [2]. In work-stealing, on the other hand, each process tries to work on its newly created threads locally, and attempts to steal threads from other processes only when it has no local threads to execute. This way, the computational overhead of re-balancing is paid by the processes that would otherwise be idle.

The ABP work-stealing algorithm of Arora, Blumofe, and Plaxton [3] has been gaining popularity as the multiprocessor load-balancing technology of choice in both industry and academia [3–6]. The scheme implements a provably efficient work-stealing paradigm due to Blumofe and Leiserson [7] that allows each process to maintain a local work deque,[1] and steal an item from others if its deque becomes empty. It has been extended in various ways such as stealing multiple items [9] and stealing in a locality-guided way [4]. At the core of the ABP algorithm is an efficient scheme for stealing an item in a non-blocking manner from an array-based deque, minimizing the need for costly Compare-and-Swap (CAS)[2] synchronization operations when fetching items locally.

Unfortunately, the use of fixed size arrays[3] introduces an inefficient memory-size/robustness tradeoff: for n processes and total allocated memory size m, one can tolerate at most m/n items in a deque. Moreover, if overflow does occur, there is no simple way to malloc additional memory and continue. This has, for example, forced parallel garbage collectors using work-stealing to implement an application-specific blocking overflow management mechanism [5, 10]. In multiprogrammed systems, the main target of ABP work-stealing [3], even inefficient over-allocation based on an application's maximal execution-DAG depth [3, 7] may not always work. If a small subset of non-preempted processes end up queuing most of the work items, since the ABP algorithm sometimes starts pushing items from the middle of the array even when the deque is empty, this can lead to overflow.[4]

This state of affairs leaves open the question of designing a dynamic memory algorithm to overcome the above drawbacks, but to do so while maintaining the low-cost synchronization overhead of the ABP algorithm. This is not a straightforward task, since the the array-based ABP algorithm is unique: it is possibly the only real-world algorithm that allows one to transition in a lock-free manner from the common case of using loads and stores to using a costly CAS *only* when a potential conflict requires processes to synchronize. This transition rests on the ability to detect these boundary synchronization cases based on the relative gap among array indexes. There is no straightforward way of translating this algorithmic trick to the pointer-based world of dynamic data structures.

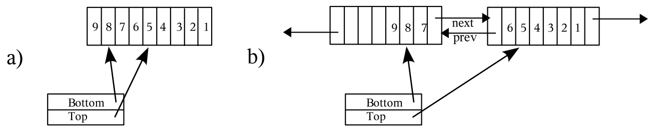

Fig. 1 The original ABP deque structure **a** vs. that of the new dynamic deque **b**. The structure is after 9 `PushBottom` operations, 4 successful `PopTop` operations, and 2 `PopBottom` operations. (In practice the original ABP deque uses cell indexes and not pointers as in our illustration.)

1.1 The new algorithm

This paper introduces the first lock-free[5] *dynamic-sized* version of the ABP work-stealing algorithm. It provides a near-optimal memory-size/robustness tradeoff: for n processes and total pre-allocated memory size m, it can potentially tolerate up to $O(m)$ items in a single deque. It also allows one to malloc additional memory beyond m when needed, and as our empirical data shows, it is far more robust than the array-based ABP algorithm in multiprogrammed environments.

An ABP-style work stealing algorithm consists of a collection of *deque* data structures with each process performing pushes and pops on the "bottom" end of its local deque and multiple thieves performing pops on the "top" end. The new algorithm implements each deque as a doubly linked list of nodes, each of which is a short array that is dynamically allocated from and freed to a shared pool; see Fig. 1. It can also use malloc to add nodes to the shared pool in case its node supply is exhausted.

The main technical difficulties in the design of the new algorithm arise from the need to provide performance comparable to that of ABP. This means the doubly linked list must be manipulated using only loads and stores in the common case, resorting to using a costly CAS *only* when a potential conflict requires it; it is challenging to make this transition correctly while maintaining lock-freedom.

The potential conflict that requires CAS-based synchronization occurs when a pop by a local process and a pop by a thief might both be trying to remove the same item from the deque. The original ABP algorithm detects this scenario by examining the gap between the `Top` and `Bottom` array indexes, and uses a CAS operation only when they are "too close." Moreover, in the original algorithm, the empty deque scenario is checked simply by checking whether `Bottom` $\leq$ `Top`.

A key algorithmic feature of our new algorithm is the creation of an equivalent mechanism to allow detection of these boundary situations in our linked list structures using the relations between the `Top` and `Bottom` pointers, even

[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] The *CAS* (*location, old-value, new-value*) operation atomically reads a value from *location*, and writes *new-value* in *location* if and only if the value read is *old-value*. The operation returns a boolean indicating whether it succeeded in updating the location.

[3] One may use cyclic array indexing but this does not help in preventing overflows.

[4] The ABP algorithm's built-in "reset on empty" mechanism helps in some, but not all, of these cases.

[5] Our abstract deque definition is such that the original ABP algorithm is also lock-free.

though these point to entries that may reside in different nodes. On a high level, our idea is to prove that one can restrict the number of possible ways the pointers interact, and therefore, given one pointer, it is possible to calculate the different possible positions for the other pointer that imply such a boundary scenario.

The other key feature of our algorithm is that the dynamic insertion and deletion operations of nodes into the doubly linked-list (when needed in a push or pop) are performed in such a way that the local thread uses only loads and stores. This contrasts with the more general linked-list deque implementations [11, 12] which require a double-compare-and-swap synchronization operation [13] to insert and delete nodes.

1.2 Performance analysis

We compared our new dynamic-memory work-stealing algorithm to the original ABP algorithm on a 16-node shared memory multiprocessor using the benchmarks of the style used by Blumofe and Papadopoulos [14]. We ran several standard *Splash2* [15] applications using the Hood scheduler [16] with the ABP and new work-stealing algorithms. Our results, presented in Sect. 3, show that the new algorithm performs as well as ABP, that is, the added dynamic-memory feature does not slow the applications down. Moreover, the new algorithm provides a better memory/robustness ratio: the same amount of memory provides far greater robustness in the new algorithm than the original array-based ABP work-stealing. For example, running Barnes-Hut using ABP work-stealing with an 8-fold level of multiprogramming causes a failure in 40% of the executions if one uses the deque size that works for stand-alone (non-multiprogrammed) runs. It causes *no failures* when using the new dynamic memory work-stealing algorithm.

2 The algorithm

2.1 Basic description

Figure 1b presents our new deque data-structure. The doubly-linked list's nodes are allocated from and freed to a shared pool, and the only case in which one may need to malloc additional storage is if the shared pool is exhausted. The deque supports the `PushBottom` and `PopBottom` operations for the local process, and the `PopTop` operation for the thieves.

The first technical difficulty we encountered is in detecting the conflict that may arise when the local `PopBottom` and a thief's `PopTop` operations concurrently try to remove the last item from the deque. Our solution is based on the observation that when the deque is empty, one can restrict the number of possible scenarios among the pointers. Given one pointer, we show that the "virtual" distance of the other, ignoring which array it resides in, cannot be more than 1 if the deque is empty. We can thus easily test for each of these scenarios. (Several such scenarios are depicted in parts (a) and (b) of Fig. 2).

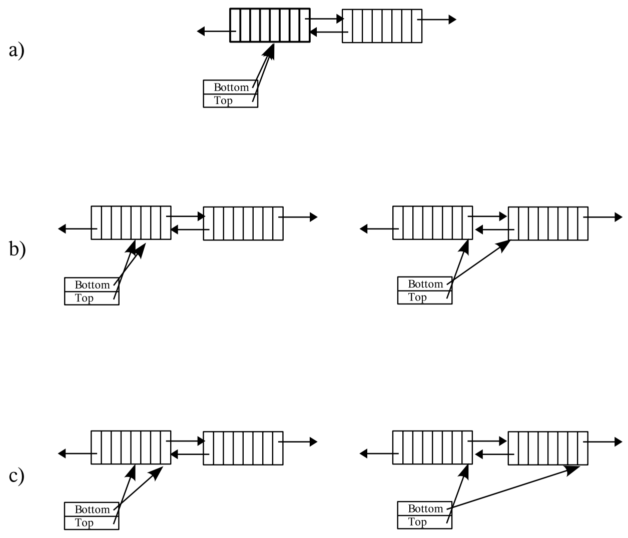

Fig. 2 The different types of empty deque scenarios. **a** Simple: `Bottom` and `Top` point to the same cell. **b** Simple Crossing: both the left and right scenarios are examples where `Bottom` passed over `Top` by one cell, but they still point to neighboring cells. **c** Non-Simple Crossing (with the reset-on-empty heuristic): both the left and right scenarios are examples of how pointers can cross given the reset-on-empty heuristic, between the reset of `Bottom` to the reset of `Top`

The next problem one faces is the maintenance of the deque's doubly-linked list structure. We wish to avoid using CAS operations when updating the `next` and `previous` pointers, since this would cause a significant performance penalty. Our solution is to allow only the local process to update these fields, thus preventing `PopTop` operations from doing so when moving from one node to another. We would like to keep the deque dynamic, which means freeing old nodes when they're not needed anymore. This restriction immediately implies that an active list node may point to an already freed node, or even to a node which was freed and reallocated again, essentially ruining the list structure. As we prove, the algorithm can overcome this problem by having a `PopTop` operation that moves to a new node free only the node *preceding* the old node and not the old node itself. This allows us to maintain the invariant that the doubly-linked list structure between the `Top` and `Bottom` pointers is preserved. This is true even in scenarios such as that depicted in parts b and c of Fig. 2 where the pointers cross over.

2.2 The implementation

`C++`-like pseudocode for our deque algorithm is given in Figs. 3–5. As depicted in Fig. 3, the deque object stores the `Bottom` and `Top` pointers information in the `Bottom` and `Top` data members. This information includes the pointer to a list's node and an offset into that node's array. For the `Top` variable, it also includes a tag value to prevent the ABA problem [17]. The deque methods uses the *EncodeBottom*, *DecodeBottom*, *EncodeTop* and *DecodeTop* macros to

```
struct BottomStruct {
     DequeNode* nodeP;
     int cellIndex;
};
struct TopStruct {
     DequeNode* nodeP;
     int cellIndex;
     int tag;
};

class DynamicDeque {
     void PushBottom(ThreadInfo theData);
     ThreadInfo PopTop();
     ThreadInfo PopBottom();

     BottomStruct Bottom;
     TopStruct Top;
};
```

```
struct DequeNode {
     enum{ArraySize=/*Size of array*/};
     ThreadInfo itsDataArr[ArraySize];
     DequeNode* next;
     DequeNode* prev;
};
```

Fig. 3 Data types and classes used by the dynamic deque algorithm

encode/decode this information to/from a value that fits in a CAS-able size word.[6] Underlined procedures in the pseudocode represent code blocks which are presented in the detailed algorithm presentation used for the correctness proof in Sect. 4. We now describe each of the methods.

2.2.1 PushBottom

The PushBottom method begins by reading Bottom and storing the pushed value in the cell it's pointing to (Lines 1–2). Then it calculates the next value of Bottom linking a new node to the list if necessary (Lines 3–14). Finally the method updates Bottom to its new value (Line 15). As in the original ABP algorithm, this method is executed only by the owner process, and therefore regular writes suffice (both for the value and Bottom updates). Note that the new node is linked to the list *before* Bottom is updated, so the list structure is preserved for the nodes between Bottom and Top.

2.2.2 PopTop

The PopTop method begins by reading the Top and Bottom values, in that order (Lines 16–18). Then it tests whether these values indicate an empty deque, and returns EMPTY if they do[7] (Line 19). Otherwise, it calculates the next position for Top (Lines 20–31). Before updating Top to its new value, the method must read the value which should be returned if the steal succeeds (Line 32) (this read cannot be done after the update of Top because by then the node may already be freed by some other concurrent PopTop execution). Finally the method tries to update Top to its new value using a CAS operation (Line 34), returning the popped value if it succeeds, or ABORT if it fails. (In the work stealing algorithm, if a thief process encounters contention with another, it may be preferable to try stealing from a different deque; returning ABORT in this case provides the opportunity for the system to decide between retrying on the same deque or doing something different.) If the CAS succeeds, the method also checks whether there is an old node that needs to be freed (Line 36). As explained earlier, a node is released only if Top moved to a new node, and the node released is not the old top node, but the preceding one.

2.2.3 PopBottom

The PopBottom method begins by reading Bottom and updating it to its new value (Lines 43–55) after reading the value to be popped (Line 54). Then it reads the value of Top (Line 56), to check for the special cases of popping the last entry of the deque, and popping from an empty deque. If the Top value read points to the old Bottom position (Lines 58–63), then the method rewrites Bottom to its old position, and returns EMPTY (since the deque was empty even without this PopBottom operation). Otherwise, if Top is pointing to the new Bottom position (Lines 64–78), then the popped entry was the last in the deque, and as in the original ABP algorithm, the method updates the Top tag value using a CAS, to prevent a concurrent PopTop operation from popping out the same entry. Otherwise there was at least one entry in the deque after the Bottom update (lines 79–83), in which case the popped entry is returned. Note that, as in the original ABP algorithm, most executions of the method will be short, and will not involve any CAS-based synchronization operations.

2.2.4 Memory management

We implement the shared node pool using a variation of Scott's shared pool [18]. It maintains a local group of g nodes per process, from which the process may allocate nodes without the need to synchronize. When the nodes in this local group are exhausted, it allocates a new group of g nodes from a shared LIFO pool using a CAS operation. When a process frees a node, it returns it to its local group, and if the size of the local group exceeds $2g$, it returns g nodes to the shared LIFO pool. In our benchmarks we used

[6] If the architecture does not support a 64-bit CAS operation, we may not have the space to save the whole node pointer. In this case, we might use the offset of the node from some base address given by the shared memory pool. For example, if the nodes are allocated continuously, the address of the first node can be such a base address.

[7] This test may also return ABORT if Top was modified, since then it is not guaranteed that the tested values represent a consistent view of the memory.

```
void DynamicDedeque::PushBottom(ThreadInfo theData)
{
<currNode, currIndex> = DecodeBottom(Bottom); // Read Bottom data
currNode->itsDataArr[currIndex] = theData;     // Write data in current bottom cell
if (currIndex!=0)
{
            newNode = currNode;
            newIndex = currIndex-1;
}
else
{   // Allocate and link a new node:
            newNode = AllocateNode();
            newNode->next = currNode;
            currNode->prev = newNode;
            newIndex = DequeNode::ArraySize-1;
}
Bottom = EncodeBottom(newNode,newIndex);    // Update Bottom
}



ThreadInfo DynamicDedeque::PopTop()
{
currTop = Top;     // Read Top
<currTopTag, currTopNode, currTopIndex> = DecodeTop(currTop);
currBottom = Bottom;  // Read Bottom
if (EmptinessTest(currBottom,currTop)) {
   if (currTop == Top) {return EMPTY;} else {return ABORT;}
}
if (currTopIndex!=0)     // if deque isn't empty, calculate next top pointer:
{   // stay at current node:
            newTopTag = currTopTag;
            newTopNode = currTopNode;
            newTopIndex = currTopIndex-1;
}
else
{   // move to next node and update tag:
            newTopTag = currTopTag+1;
            newTopNode = currTopNode->prev;
            newTopIndex = DequeNode::ArraySize-1;
}
retVal = currTopNode->itsDataArr[currTopIndex]; // Read value
newTopVal = Encode(newTopTag,newTopNode,newTopIndex);
if (CAS(&Top, currTop, newTopVal))   //Try to update Top using CAS
{
            FreeOldNodeIfNeeded();
            return retVal;
}
else
{
            return ABORT;
}
}
```

Fig. 4 Pseudocode for the PushBottom and PopTop operations

a group size of 1, which means that in case of a fluctuation between pushing and popping, the first node is always local and CAS is not necessary.

2.3 Enhancements

We briefly describe two enhancements to the above dynamic-memory deque algorithm.

2.3.1 Reset-on-Empty

In the original ABP algorithm, the `PopBottom` operation uses a mechanism that resets `Top` and `Bottom` to point back to the beginning of the array every time it detects an empty deque (including the case of popping the last entry by `PopBottom`). This reset operation is necessary in ABP since it is the only "anti-overflow" mechanism at its disposal.

```
ThreadInfo DynamicDedeque::PopBottom()
{
43 <oldBotNode,oldBotIndex > = DecodeBottom(Bottom); // Read Bottom Data
44 if (oldBotIndex != DequeNode::ArraySize-1)
45 {
46             newBotNode = oldBotNode;
47             newBotIndex = oldBotIndex+1;
48 }
49 else
50 {
51             newBotNode = oldBotNode->next;
52             newBotIndex = 0;
53 }
54 retVal = newBotNode->itsDataArr[newBotIndex];      // Read data to be popped
55 Bottom = EncodeBottom(newBotNode,newBotIndex);     // Update Bottom
56 currTop = Top;                                     // Read Top
57 <currTopTag,currTopNode,currTopIndex> = DecodeTop(currTop);

58 if (oldBotNode == currTopNode &&           // Case  1:  if Top has crossed Bottom
59     oldBotIndex == curTopIndex  )
60 {
     //Return bottom to its old possition:
61             Bottom = EncodeBottom(oldBotNode,oldBotIndex);
62             return EMPTY;
63 }
64 else if ( newBotNode == currTopNode &&     // Case 2:  When  popping the last entry
65           newBotIndex == currTopIndex )    //          in the deque  (i.e. deque is
66 {                                          //          empty after the update of bottom).

     //Try to update Top's tag so no concurrent PopTop operation will also pop the same entry:
67             newTopVal = Encode(currTopTag+1, currTopNode, currTopIndex);
68             if (CAS(&Top, currTop, newTopVal))
69             {
70                FreeOldNodeIfNeeded();
71                return retVal;
72             }
73             else  // if CAS failed (i.e. a concurrent PopTop operation already popped the last entry):
74             {
               //Return bottom to its old possition:
75                Bottom = EncodeBottom(oldBotNode,oldBotIndex);
76                return EMPTY;
77             }
78 }
79 else // Case 3:  Regular case (i.e. there was at least one entry in the deque after bottom's update):
80 {
81                FreeOldNodeIfNeeded();
82                return retVal;
83 }
}
```

Fig. 5 Pseudocode for the PopBottom operation

Our algorithm does not need this method to prevent overflows, since it works with the dynamic nodes. However, adding a version of this resetting feature gives the potential of improving our space complexity, especially when working with large nodes.

There are two issues to be noted when implementing the reset-on-empty mechanism in our dynamic deque. The first issue is that while performing the reset operation, we create another type of empty deque scenario, in which `Top` and `Bottom` do not point to the same cells nor to neighboring ones (see part *c* of Fig. 2). This scenario requires a more complicated check for the empty deque scenario by the `PopTop` method (Line 19). The second issue is that we must be careful when choosing the array node to which `Top` and `Bottom` point after the reset. In case the pointers point to the same node before the reset, we simply reset to the beginning of that node. Otherwise, we reset to the beginning of the node pointed to by Top. Note, however, that `Top` may point to the same node as `Bottom` and then be updated by a concurrent `PopTop` operation, which may result in changing on-the-fly the node to which we direct `Top` and `Bottom`.

2.3.2 Using a base array

In the implementation described, all the deque nodes are identical and allocated from the shared pool. This introduces a trade-off between the performance of the algorithm and its space complexity: small arrays save space but cost in allocation overhead, while large arrays cost space but reduce the allocation overhead.

One possible improvement is to use a large array for the initial `base` node, allocated for each of the deques, and to use the pool only when overflow space is needed. This base node is used only by the process/deque it was originally allocated to, and is never freed to the shared pool. Whenever a Pop operation frees this node, it raises a boolean flag, indicating that the base node is now free. When a `PushBottom` operation needs to allocate and link a new node, it first checks this flag, and if true, links the base node to the deque (instead of a regular node allocated from the shared pool).

3 Performance

We evaluated the performance of the new dynamic memory work-stealing algorithm in comparison to the original fixed-array based ABP work-stealing algorithm in an environment similar to that used by Blumofe and Papadopoulos [14] in their evaluation of the ABP algorithm. Our results include tests running several standard Splash2 [15] applications using the *Hood Library* [16] 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 the Solaris™ 9 operating system.

Our benchmarks used the work-stealing algorithms as the load balancing mechanism in Hood. The Hood package uses the original ABP deques for the scheduling of threads over processes. We compiled two versions of the Hood library, one using an ABP implementation, and the other using the new implementation. In order for the comparison to be fair, we implemented both algorithms in C++, using the same tagging method.

We present here our results running the *Barnes Hut* and *MergeSort* Splash2 [15] applications. Each application was compiled with the minimal ABP deque size needed for a stand-alone run with the biggest input tested. For our deque algorithm we chose a base-array size of about 75% of the ABP deque size, a node array size of 6 items, and a shared pool size such that the total memory used (by the deques and the shared pool together) is no more than the total memory used by all ABP deques. In all our benchmarks the number of processes equaled the number of processors on the machine.

Figure 6 shows the total execution time of both algorithms, running stand-alone, as we vary the input size. As can be seen, there is no real difference in performance between the two approaches. This is in spite of the fact that our tests show that the deque operations of the new algorithm take as much as 30% more time on average than those of ABP. The explanation is simple: work stealing accounts for only a small fraction of the execution time in these (and in fact in most) applications. In all cases both algorithms had a 100% completion rate in stand-alone mode, i.e. none of the deques overflowed.

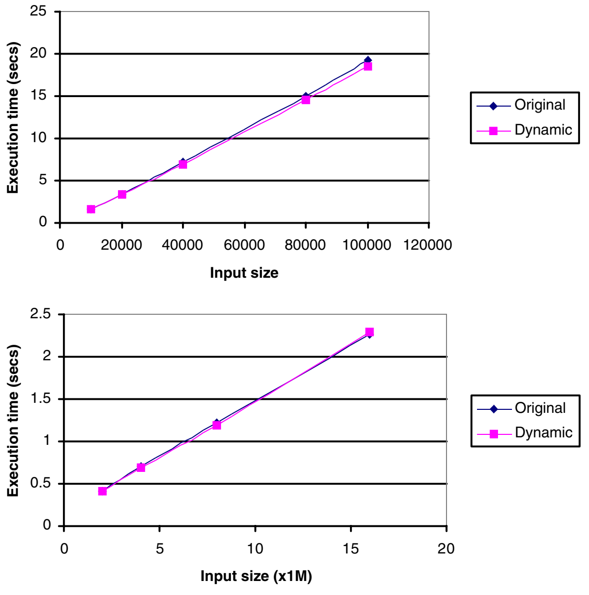

Fig. 6 *Barnes Hut* Benchmark on top and *MergeSort* on the bottom

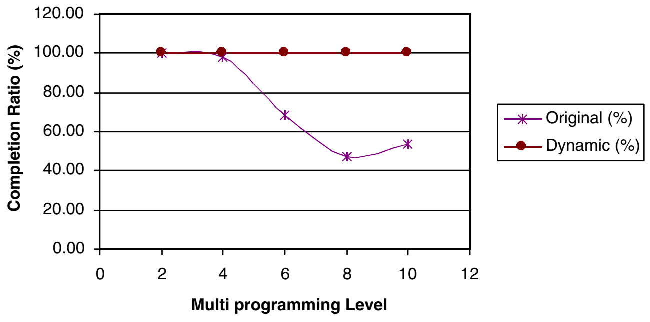

Fig. 7 *Barnes Hut* completion ratio vs. level of multiprogramming

Figure 7 shows the results of running the *Barnes Hut* [15] application (on the largest input) in a multiprogrammed fashion by running multiple instances of Hood in parallel. The graph shows the completion rate of both algorithms as a function of the multiprogramming level (i.e. the number of instances run in parallel). One can clearly see that while both versions perform perfectly at a multiprogramming level of 2, ABP work-stealing degrades rapidly as the level of multiprogramming increases, while the new algorithm maintains its 100% completion rate. By checking Hood's statistics regarding the amount of work done by each process, we noticed that some processes complete 0 work, which means much higher workloads on the others. This, we believe, caused the deque size which worked for a stand-alone run (in which the work was more evenly distributed between the processes), to overflow in the multiprogrammed run. We also note that as the workload on individual processes increases, the chances of a "reset-on-empty" decrease, and the likelihood of overflow increases. In the new dynamic version, because 25% of the memory is allocated in the common shared pool, there is much more flexibility in dealing with the work imbalance between the deques, and no overflow occurs.

Our preliminary benchmarks clearly show that for the same amount of memory, we get significantly more robustness with the new dynamic algorithm than with the original ABP algorithm, with a virtually unnoticeable effect on the application's overall performance. It also shows that the deque size depends on the maximal level of multiprogramming in the system, an unpredictable parameter which one may want to avoid reasoning about by simply using the new memory version of the ABP work stealing algorithm.

```
DynamicDeque:DynamicDeque()
nodeA = AllocateNode();
nodeB = AllocateNode();
nodeA→next = nodeB;
nodeB→prev = nodeA;
Bottom= EncodeBottom(nodeA,DequeNode::ArraySize-1);
Top= EncodeTop(0 ,nodeA, DequeNode::ArraySize-1);
Ordered= {nodeA, nodeB};
```

Fig. 8 Deque Constructor

```
  void DynamicDedeque::PushBottom(ThreadInfo theData)
1 <currNode, currIndex> = DecodeBottom(Bottom);
2 currNode→itsDataArr[currIndex] = theData;
  if (currIndex!=0) then
3     newNode = currNode;
      newIndex = currIndex-1;
  else
4     newNode = AllocateNode();
5     newNode→next = currNode;
6     currNode→prev = newNode;
      newIndex = DequeNode::ArraySize-1;
7 Bottom= EncodeBottom(newNode,newIndex);
  if (currNode != newNode) then Ordered.AddLeft(newNode);
```

Fig. 9 The PushBottom Method

4 Correctness proof

4.1 Overview

The full version of this paper [19] contains a detailed proof that the algorithm in Sect. 2 implements a lock-free linearizable deque.[8] While a work stealing system generally uses several deques, our proof concentrates on a single deque. For brevity, this section only contains the outline of the proof. All omitted proofs and claims can be found in the full version of the paper [19]. To make it easier for the reader, we kept the numbering of the claims the same as in the full version.

We first define notation and terminology and present a detailed version of the algorithm's pseudocode that we use throughout the proof. In Sects. 4.2–4.6 we present various properties of the algorithm, which are used later in the linearizability proof. Section 4.7 specifies the sequential semantics of the implemented deque and then shows that the deque is linearizable to this specification. Finally, Sect. 4.8 shows that the algorithm is lock-free.

4.1.1 Notation

Formally, we model the algorithm by a labelled state-transition system, where the labels are called *actions*. We write $s \xrightarrow{a} s'$ if the system has a transition from s to s' labelled a; s is called the *pre-state*, s' the *post-state*. We say that an action a is *enabled* in a state s if there exists another state s' such that $s \xrightarrow{a} s'$. An execution is a sequence of transitions such that the pre-state of the first transition is the initial state, and the post-state of any transition (except the last) is the pre-state of the next transition.

We use s and s' for states, a for actions, and p, p' and p'' for processes. We use $p@X$ to mean that process p is ready to execute statement number X. We use $p@\langle X_1, X_2, ..., X_n\rangle$ to denote $p@X_1 \vee p@X_2 \vee ... \vee p@X_n$. If process p is not executing any operation then $p@0$, holds. Thus, $p@0$ holds initially for all p, statements of process that return from any of the operations establish $p@0$, and if $p@0$ holds, then an action of process p is enabled that nondeterministically chooses a legal operation and parameters, and invokes the operation, thereby setting p's program counter to the first line of that operation, and establishing legal values for its parameters. We denote a private variable v of process p by $p.v$.

For any variable v, shared or local, $s.v$ denotes the value of v is state s. For any logical expression E, $s.E$ holds if and only if E holds in state s.

4.1.2 Pseudocode

Figures 8–12 present the detailed pseudocode of the deque implementation. The `EncodeTop`, `DecodeTop`, `EncodeBottom` and `DecodeBottom` macros are described in Sect. 4.2.2. The `Ordered` variable used in the pseudocode is an auxiliary variable; its use is described in Sect. 4.3. Auxiliary variables do not affect the behavior of the algorithm, and are used only for the proof: they are not included in a real implementation.

We consider execution of the algorithm beginning at one numbered statement and ending immediately before the next numbered statement to be one atomic action. This is justified by the fact that no such action accesses more than one shared variable (except auxiliary variables), and atomicity of the actions is consistent with available operations for memory access. Note that we can include accesses to auxiliary variables in an atomic action because they are not included in a real implementation, and therefore are not required to comport with the atomicity constraints of a target architecture.

As a concrete example, consider the action labelled 17 in Fig. 10, when executed by process p. This action atomically does the following. First, the action reads `Top` and compares

[8] As noted previously, the data structure we implement is not strictly speaking a deque. The precise semantics of the implemented data structure is specified in Sect. 4.7.1.

```
   ThreadInfo DynamicDedeque::PopTop()
 8 currTop = Top;
   <currTopTag, currTopNode, currTopIndex> = DecodeTop(currTop);
 9 currBottom = Bottom;
10 if (IndicateEmpty(currBottom, currTop)) then
11     if (currTop == Top) then return EMPTY;
       return ABORT;
12 if (currTopIndex!=0) then
13     nodeToFree = NULL;
       newTopTag = currTopTag;
       newTopNode = currTopNode;
       newTopIndex = currTopIndex-1;
       newTopVal = EncodeTop(newTopTag,newTopNode,newTopIndex);
   else
14     nodeToFree = currTopNode→next;
15     newTopTag = currTopTag+1;
       newTopNode = currTopNode→prev;
       newTopIndex = DequeNode::ArraySize-1;
       newTopVal = EncodeTop(newTopTag,newTopNode,newTopIndex);
16 retVal = currTopNode→itsDataArr[currTopIndex];
17 if (CAS(&Top, currTop, newTopVal)) then
       if (nodeToFree != NULL) then Ordered.RemoveRight();
18     if (nodeToFree != NULL) then FreeNode(nodeToFree);
19     return retVal;
   else
20     return ABORT;
```

Fig. 10 The PopTop Method

the value read to *p.currTop*. If Top $\neq$ *p.currTop*, then the action changes *p*'s program counter to 20. Otherwise, the action stores *p.newTopVal* to Top, removes the rightmost element from Ordered if *p.nodeToFree* $\neq$ *NULL*, and changes *p*'s program counter to 18.

4.1.3 Proof method

Most of the invariants in the proof are proved by induction on the length of an arbitrary execution of the algorithm. That is, we show that the invariant holds initially, and that for any transition $s \xrightarrow{a} s'$, if the invariant holds in the pre-state s, then it also holds in the post-state s'. For invariants of the form $A \Rightarrow B$, we often find it convenient to prove this by showing that the consequent (B) holds after any statement execution establishes the antecedent (A), and that no statement execution falsifies the consequent while the antecedent holds.

It is convenient to prove the conjunction of all of the properties, rather than proving them one by one. This way, we can assume that all properties hold in the pre-state when proving that a particular property holds in the post-state. It is also convenient to be able to use other properties in the post-state. However, this must be done with some care, in order to avoid circular reasoning. It is important that there is a single order in which we prove all properties hold in the post-state, given the inductive assumption that they all hold in the pre-state, without using any properties we have not yet proved. However, presenting the proofs in this order would disturb the flow of the proof from the reader's point of view. Therefore, we now present some rules that we adopted that imply that such an order exists.

The properties of the proof (Invariants, Claims, Lemmas and Corollaries) are indexed by the order in which they are proved. In some cases, we state an invariant without proving it immediately, and only provide its proof after presenting and proving some other properties. We call such invariants *Conjectures* to clearly distinguish them from regular properties, which are proved as soon as they are stated. To avoid circular reasoning, our proofs obey the following rules:

1. The proof of Property i can use any Property j in the pre-state.
2. If Property i is not a Conjecture, its proof can use the following properties in the post-state:
   (a) All Conjectures.
   (b) Property j if and only if $j < i$.
3. The proof of Conjecture i can use Conjecture j in the post-state if and only if $i < j$.

Informally, these rules simply state that the proof of a non-Conjecture property can use in the post-state any other

```
   ThreadInfo DynamicDedeque::PopBottom()
21 oldBotVal = Bottom;
   <oldBotNode,oldBotIndex> = DecodeBottom(oldBotVal);
22 if (oldBotIndex != DequeNode::ArraySize-1) then
23     newBotNode = oldBotNode;
       newBotIndex = oldBotIndex+1;
       newBotVal = EncodeBottom(newBotNode,newBotIndex);
   else
24     newBotNode = oldBotNode→next;
       newBotIndex = 0;
       newBotVal = EncodeBottom(newBotNode,newBotIndex);
25 Bottom= newBotVal;
26 currTop = Top;
   <currTopTag,currTopNode,currTopIndex> = DecodeTop(currTop);
27 retVal = newBotNode→itsDataArr[newBotIndex];
28 if (oldBotNode == currTopNode && oldBotIndex == curTopIndex) then
29     Bottom= EncodeBottom(oldBotNode,oldBotIndex);
30     return EMPTY;
31 else if (newBotNode == currTopNode && newBotIndex == currTopIndex)
   then
32     newTopVal = EncodeTop(currTopTag+1, currTopNode, currTopIndex);
33     if (CAS(&Top, currTop, newTopVal)) then
34         if (oldBotNode != newBotNode) then
               FreeNode(oldBotNode);
               Ordered.RemoveLeft();
35         return retVal;
       else
36         Bottom= EncodeBottom(oldBotNode,oldBotIndex);
37         return EMPTY;
   else
38     if (oldBotNode != newBotNode) then
           FreeNode(oldBotNode);
           Ordered.RemoveLeft();
39     return retVal;
```

Fig. 11 The PopBottom Method

```
bool IndicateEmpty(BottomStruct bottomVal, TopStruct topVal)
<botNode, botCellIndex> = DecodeBottom(bottomVal);
<topTag, topNode, topCellIndex> = DecodeTop(topVal);
if ((botNode==topNode) && (botCellIndex==topCellIndex ||
botCellIndex==(topCellIndex+1))) then
    return true;
else if ((botNode==topNode→next) && (botCellIndex==0) &&
(topCellIndex==(DequeNode::ArraySize-1))) then
    return true;
return false;
```

Fig. 12 The IndicateEmpty Macro

property that was already stated (because the only properties that were stated before it but with higher index are Conjectures), and that the proof of a Conjecture can use in the post-state any other Conjecture that was not already proven. In the full version of the paper [19] we show that our proof method is sound.

To make the proof more readable, we also avoid using in the pre-state properties that were not stated yet.

4.2 Basic notation and invariants

4.2.1 The deque data structure

Our deque is implemented using a doubly linked list. Each list node contains an array of deque entries. The structure has `Bottom` and `Top` variables that indicate cells at the two ends of the deque; these variables are discussed in more detail in Sect. 4.2.2. We use the following notation:

- We let N_j denote a (pointer to a) deque node, and C_i denotes a cell at index i in a node. $C_i \in N_j$ denotes that C_i is the ith cell in node N_j.
- We let $Node(C_i)$ denote the node of cell C_i. That is: $Node(C_i) = N_j \Leftrightarrow C_i \in N_j$.
- The *Bottom cell* of the deque, denoted by C_B, is the cell indicated by `Bottom`. The *Bottom node* is the node in which C_B resides, and is denoted by N_B.
- The *Top cell* of the deque, denoted by C_T, is the cell indicated by `Top`. The *Top node* is the node in which C_T resides, and is denoted by N_T.
- If N is a deque node, than $N \rightarrow next$ is the node pointed to by N's next pointer, and $N \rightarrow prev$ is the node pointed to by its previous pointer.

The following property models the assumption that only one process calls the PushBottom and PopBottom operations.

Invariant 1. *If* $p@\langle 1 \dots 7, 21 \dots 39\rangle$ *then:*

1. *p is the owner process of the deque.*
2. *There is no* $p' \neq p$ *such that* $p'@\langle 1 \dots 7, 21 \dots 39\rangle$.

Proof The invariant follows immediately from the requirement that only the owner process may call the PushBottom or PopBottom procedures. □

4.2.2 The `Top` *and* `Bottom` *variables*

The `Top` and `Bottom` shared variables store information about C_T and C_B, respectively, and they are both of a CASable size. The `Top` variable also contains an unbounded Tag value, to avoid the ABA problem as we describe in Sect. 4.2.3. The structure of `Top` and `Bottom` variables is detailed in Fig. 3.

In practice, in order to store all the information on a CASable word size even if only a 32-bit CAS operation is available, we represent the node's pointer by its *offset* from some base address given by the nodes' memory manager. In this case, if the size of the node is of a power of two, we can even save only the offsets to C_T and C_B, and calculate the offsets of N_T and N_B by simple bitwise operations. That way we save the space used by the `cellIndex` variable, and leave enough space for the tag value.

In the rest of the proof we use the *Cell* operator to denote the cell to which a variable of type `BottomStruct` or `TopStruct` points (for example, *Cell* (`Top`) $= C_T$ and *Cell* (`Bottom`) $= C_B$):

Definition 3 If `TorBVal` is a variable of type `TopStruct` or `BottomStruct` then: $Cell(TorBVal) = TorBVal.nodeP \rightarrow itsDataArr[TorBVal.cellIndex]$.

Our implementation uses the `EncodeTop` and `EncodeBottom` macros to construct values of type `TopStruct` and `BottomStruct`, respectively, and similarly uses the `DecodeBottom` and `DecodeTop` macros to extract the components from values of these types. For convenience, we use processes' private variables to hold the different fields of values read from `Top` and `Bottom`. For example, after executing the code segment:

```
oldBotVal = Bottom;
<oldBotNode,oldBotIndex> = DecodeBottom(oldBotVal);
```

using `oldBotNode` and `oldBotIndex`, as long as they are not modified, is equivalent to using `oldBotVal.nodeP` and `oldBotVal.cellIndex`, respectively.

Invariant 6.

1. $p@\langle 2 \dots 7\rangle \Rightarrow (N_B = p.currNode \wedge C_B = C_{p.currIndex} \in N_B)$.
2. $p@\langle 22 \dots 25, 30, 37\rangle \Rightarrow$ `Bottom` $= p.oldBotVal$.
3. $p@\langle 26 \dots 29, 31 \dots 36, 38 \dots 39\rangle \Rightarrow$ `Bottom` $= p.newBotVal$.

4.2.3 The ABA problem

Our implementation uses the CAS operation for all updates of the `Top` variable. The CAS synchronization primitive is susceptible to the *ABA problem*: Assume that the value A is read from some variable v, and later a CAS operation is done on that variable, with the value A supplied as the old-value parameter of the CAS. If, between the read and the CAS, the variable v has been changed to some other value B and then to A again, the CAS would still succeed.

In this section, we prove some properties concerning mechanisms used in the algorithm to avoid the ABA problem. We start by defining an order between different `Top` values:

Definition 7 Let $TopV_1$ and $TopV_2$ be two values of type `TopStruct`. $TopV_1 \ll TopV_2$ if and only if:

1. $TopV_1.tag \leq TopV_2.tag$, and
2. $(TopV_1.tag = TopV_2.tag) \Rightarrow (TopV_1.cellIndex > TopV_2.cellIndex)$.

Lemma 10 *Let $s \xrightarrow{a} s'$ be a step of the algorithm, and suppose a writes a value to* `Top`*. Then $s.Top \ll s'.Top$.*

Corollary 12 *Consider a transition $s \xrightarrow{a} s'$ where a writes a value to* `Top`*. Then: $\forall p\ s.p@\langle 9 \dots 20, 27 \dots 39\rangle \Rightarrow s'.p.currTop \neq s'.Top$*

4.2.4 Memory management

Our algorithm uses an external linearizable *shared pool* module, which stores the available list nodes. The shared pool module supports two operations: `AllocateNode` and `FreeNode`. The details of the shared pool implementation are not relevant to our algorithm, so we simply model a linearizable shared pool that supports atomic `AllocateNode` and `FreeNode` operations.

We model the shared pool using an auxiliary variable `Live`, which models the set of nodes that have been allocated from the pool and not yet freed:

1. Initially `Live` $= \emptyset$.
2. An `AllocateNode` operation atomically adds a node that is not in `Live` to `Live` and returns that node.
3. A `FreeNode`(N) operation with $N \in$ `Live`atomically removes N from `Live`.
4. While $N \in$ `Live`, the shared pool implementation does not modify any of N's fields.

The shared pool behaves according to the above rules provided our algorithm uses it properly. The following conjecture states the rules for proper use of the shared pool. We prove that the conjecture holds in Sect. 4.4.

Conjecture 30. Consider a transition $s \xrightarrow{a} s'$ of our algorithm.

- If $N \notin s.$`Live`, then a does not modify any of N's fields.
- If a is an execution of `FreeNode`(N), then $N \in s.$`Live`.

Definition 13 A node N is *live* if and only if $N \in$ `Live`.

4.3 Ordered nodes

We introduce an auxiliary variable `Ordered`, which consists of a sequence of nodes. We regard the order of the nodes in `Ordered` as going from *left* to *right*. Formally, the variable `Ordered` supports four operations: *AddLeft*, *AddRight*, *RemoveLeft* and *RemoveRight*. If $|$`Ordered`$| = l$, `Ordered` $= \{N_1, \dots, N_l\}$ then:

- N_1 is the leftmost node and N_l is the rightmost one.
- An `Ordered`.AddLeft(N) operation results in `Ordered` $= \{N, N_1, \dots, N_l\}$.
- An `Ordered`.AddRight(N) operation results in `Ordered` $= \{N_1, \dots, N_l, N\}$.
- A `Ordered`.RemoveLeft() operation results in `Ordered` $= \{N_2, \dots, N_l\}$, and returns N_1.
- A `Ordered`.RemoveRight() operation results in `Ordered` $= \{N_1, \dots, N_{l-1}\}$, and returns N_l.

Definition 14 A node N is *ordered* if and only if $N \in$ `Ordered`.

The following conjecture describes the basic properties of the nodes in `Ordered`:

Conjecture 55. Let $|$`Ordered`$| = n + 2$, `Ordered` $= \{N_0, \dots, N_{n+1}\}$. Then:

1. $\forall_{0 \le i \le n}\ N_i \to next = N_{i+1} \wedge N_{i+1} \to prev = N_i$.
2. Exactly one of the following holds:
   (a) $n \ge 0,\ N_0 = N_B, N_n = N_T$.
   (b) $n > 0,\ N_1 = N_B, N_n = N_T$.
   (c) $n = 0,\ N_0 = N_T, N_1 = N_B$.

Corollary 15

1. $|s.$`Ordered`$| \ge 2$.
2. *N_T is ordered and is the second node from the right in* `Ordered`.
3. *N_B is ordered and is either the first or the second node from the left in* `Ordered`.
4. *$N_T \to next$ is* `Ordered`.

Proof Straightforward from Conjecture 55. □

The following invariants and lemmas state different properties of the nodes in `Ordered`.

Invariant 16. *Exactly one of the following holds:*

1. *N_B is the leftmost node in* `Ordered` $\wedge$ $(p@\langle 26 \dots 34, 36 \dots 38\rangle \Rightarrow p.oldBotNode = N_B)$.
2. *$\exists p$ such that $p@\langle 26 \dots 29, 31 \dots 34, 36, 38\rangle \wedge N_B \neq p.oldBotNode \wedge p.oldBotNode$ is the leftmost node in* `Ordered`.

Conjecture 29. All ordered nodes are live.

We now present various properties about the ordered nodes, which we use later to prove Conjecture 29. The proof of Conjecture 29 appears in Sect. 4.3.1.

Invariant 20. *Suppose* `Ordered` $= \{N_0, \dots N_{n+1}\}$. *Then $\forall_{0 \le i, j \le n+1}, i \neq j \Rightarrow N_i \neq N_j$.*

Invariant 24. *If $p@\langle 22 \dots 34, 36 \dots 38\rangle$ then $p.oldBot$ $Node \in$* `Ordered`*, and it is the leftmost node there.*

Invariant 25. *If $p@\langle 5 \dots 7\rangle \wedge p'@18 \wedge p' \neq p \wedge p'.nodeToFree \neq$* `NULL` *then $p.newNode \neq p'.nodeToFree$.*

Invariant 26. *If $p@18 \wedge p'@18$, then $(p'.nodeToFree \neq p.nodeToFree) \vee (p.nodeToFree = p'.nodeToFree =$* `NULL`*).*

Conjecture 27. *If $p@18 \wedge p.nodeToFree \neq$* `NULL` *then $p.nodeToFree \in$* `Live` *$\wedge p.nodeToFree \notin$* `Ordered`.

Invariant 28. *If $p@\langle 5 \dots 7\rangle$, then $p.newNode \in$* `Live`.

4.3.1 Proof of Conjecture 29

Using the above invariants, we can now give the proof of Conjecture 29.

Conjecture 29. *All ordered nodes are live.*

Proof Initially there are two live nodes in `Ordered` (deque constructor pseudo code, depicted in Fig. 8), so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s, that is: s.`Ordered`$\subseteq$ s.`Live`. Clearly, the only operations that may falsify the invariant are deallocation of a node, or addition of a node to `Ordered`. Therefore, there are three statements to consider:

1. p. 7 for some process p: Then s'.`Live`$=$ s.`Live`and by Invariant 28 $p.newNode \in s$.`Live`. Therefore the invariant still holds in s'.
2. p. 18 for some process p: Then a deallocates $p.nodeToFree$ if and only if $p.nodeToFree \neq$ `NULL`. By Conjecture 27 $p.nodeToFree \neq$ `NULL` $\Rightarrow$ $p.nodeToFree \notin s$.`Ordered`, and since s'.`Ordered`$=$ s.`Ordered`, the invariant still holds in s'.
3. p. 34 or p. 38 for some process p: In this case, the statement deallocates $p.oldBotNode$ if and only if it also removes the leftmost node from s.`Ordered`. By Invariant 24, $p.oldBotNode \in s$.`Ordered`and it is the leftmost node there, and therefore $p.oldBotNode \notin s'$.`Live`$\Rightarrow$ $p.oldBotNode \notin s'$.`Ordered`. □

4.4 Legality of shared pool usage

In this section we show that our algorithm uses the shared pool properly, as stated by Conjecture 30.

Conjecture 30. *Consider a transition $s \xrightarrow{a} s'$.*

- *If $N \notin s$.`Live`, then a does not modify any of N's fields.*
- *If a is an execution of* `FreeNode(`N`)`*, then $N \in s$.`Live`.*

Proof We first show that a `FreeNode` operation is always called on a live node. Suppose a is a `FreeNode(`N`)` operation. The only statements which call the `FreeNode` operation are p.18, p.34 and p.38 for some process p. By Conjecture 27, if a is an execution of p.18 then $s.p.nodeToFree \in s$.`Live`, and therefore a does not falsify the invariant. Otherwise if a is an execution of p.34 or p.38, then by Invariant 24 $s.p.oldBotNode \in s$.`Ordered`, and therefore by Conjecture 29 $s.p.oldBotNode \in s$.`Live`, and therefore a does not falsify the invariant.

Next we show that a does not modify a field of node N if $N \notin s$.`Live`. The only statements that might modify node's fields are p.2, p.5 and p.6 for some process p. By Invariant 28 $p@5$ implies that $s.p.newNode \in s$.`Live`, and by Invariant 6 $s.p@\langle 2, 6\rangle$ implies that $s.p.currNode = s.N_B$. By Corollary 15 $s.N_B \in s$.`Ordered`which implies by Conjecture 29 that $s.N_B \in s$.`Live`, and therefore a does not falsify the invariant. □

4.5 Order on cells

Section 4.3 introduced the `Ordered` sequence, which defines an order between a subset of these nodes. This section defines an order between the cells of these nodes, and proves some properties regarding this order.

Definition 31 For a node $N \in$ `Ordered`, $Pos($`Ordered`$, N)$ denotes the index of N in `Ordered`, where the leftmost node in `Ordered` is indexed as 0. (Note that by Invariant 20, $Pos($`Ordered`$, N)$ is well defined.)

Definition 32 For two nodes M and N, and two cells $C_i \in M$ and $C_j \in N$, we define the *order* $\prec_s$ between these cells to be the lexicographic order $\langle Node, CellIndex\rangle$, where the nodes are ordered by the `Ordered` series, and the indices by the whole numbers order. Formally, $C_i \prec_s C_j$ if and only if $M \in s$.`Ordered`$\wedge N \in s$.`Ordered`$\wedge(Pos(s$.`Ordered`$,M) < Pos(s$.`Ordered`$, N)) \vee (M = N \wedge i < j))$.

Note that the $\prec_s$ operator depends on the state s, since it depends on the membership and order of the nodes in `Ordered`. The following lemma implies that the order between two cells cannot be changed unless the node of one of the cells is removed or added to `Ordered`:

Lemma 33 *For any step of the algorithm $s \xrightarrow{a} s'$: $(C_i \prec_s C_j) \Rightarrow \neg(C_j \prec_{s'} C_i)$.*

Proof If C_i and C_j belongs to the same node this is obvious, since the order of cells inside a node is never changed. Otherwise, suppose $C_i \in N \wedge C_j \in M \wedge (N \neq M)$. The only way the order between C_i and C_j can be changed is if the order between N and M in `Ordered` is changed. Since the `Ordered` series only supports addition and removal of nodes (and does not support any swap operation), the order of nodes inside `Ordered` cannot be changed unless one of the nodes is removed from `Ordered` first. Therefore $N \in s'$.`Ordered`$\wedge M \in s'$.`Ordered`$\Rightarrow$ $(C_i \prec_{s'} C_j)$. Otherwise by the definition of $\prec$ we have: $\neg(C_i \prec_{s'} C_j) \wedge \neg(C_j \prec_{s'} C_i)$. □

In the remainder of the proof we sometimes omit the s subscript from the $\prec_s$ operator when considering transitions that do not modify `Ordered`. We are still required to show, however, that cells' nodes are in `Ordered` in order to claim that the cells are ordered by $\prec$.

Definition 34 We define: $C_i \preceq C_j \equiv (C_i = C_j \vee C_i \prec C_j)$.

Definition 35 Let $C_i \in N_k$ and $C_j \in N_l$ be two cells such that $N_k \in$ `Ordered` and $N_l \in$ `Ordered`.

- C_i and C_j are *neighbors* if and only if they are adjacent with respect to $\prec$. We will use the predicate $Neighbors(C_i, C_j)$ to indicate if C_i and C_j are neighbors. $Neighbors(C_i, C_j)$ is false if the order between C_i and C_j is not defined.
- C_i is the *left neighbor* of C_j, denoted by $C_i = LeftNeighbor(C_j)$, if and only if $Neighbors(C_i, C_j) \wedge (C_i \prec C_j)$.

- C_i is the *right neighbor* of C_j, denoted by $C_i = Right Neighbor(C_j)$, if and only if $C_j = LeftNeighbor(C_i)$.

Note that the *LeftNeighbor* and *RightNeighbor* are only partial functions, that is they are not defined for all cells. By the definition of Neighbors and the $\prec$ order, it is easy to see that:

1. *RightNeighbor*(C_i) is defined if and only if $C_i \in N \in$ `Ordered`and either N is not the rightmost node in `Ordered`, or $i \neq$ *DeqeuNode* :: *ArraySize* $-$ *1*.
2. *LeftNeighbor*(C_i) is defined if and only if $C_i \in N \in$ `Ordered`and either N is not the leftmost node in `Ordered`, or $i \neq 0$.

4.5.1 The `IndicateEmpty` *macro*

The `IndicateEmpty` macro, called at Line 10, takes values of type `BottomStruct` and `TopStruct` and indicates whether the deque would be empty if these were the values of `Top` and `Bottom`, respectively, in the state in which the macro is invoked. The code for the macro is depicted in Fig. 12. The following Lemma describe the properties of the macro:

Lemma 36 *Let $bottomVal$ and $topVal$ be two variables of type $BottomStruct$ and $TopStruct$, respectively, and suppose $Cell(topVal) \in N \in$* `Ordered`*and that N is not the rightmost node in* `Ordered`*. Then* `IndicateEmpty` *(bottomVal, topVal)=true if and only if $(Cell(topVal) = Cell(bottomVal)) \vee (Cell(topVal) = LeftNeighbor(Cell(bottomVal)))$.*

We later prove why this property captures exactly the empty deque scenario. Note that as long as $bottomVal$ and $topVal$ are local process's variables, the `IndicateEmpty` macro does at most one read from the shared memory (that is, the read of the next pointer of the node indicated by $topVal$), and therefore is regarded as one atomic operation when called at Line 10. Finally, there is no guarantee on the return value of `IndicateEmpty` if $Node(Cell(topVal))$ is not in `Ordered`, or if it is the rightmost node there.

4.5.2 Crossing states

We say that the deque is in a crossing state when the cell pointed by `Top` is to the left of the cell pointed by `Bottom`, as depicted in Fig. 2b. As we later explain, these states correspond to an empty deque.

Definition 37 The deque is in a *crossing state* if and only if $C_T \prec C_B$.

Note that if s.`Ordered`is in the state described by part c of Conjecture 55, then the deque is in a crossing state (since N_T precedes N_B in `Ordered`). The following is the main invariant describing when and under what conditions the deque may be in a crossing state:

Conjecture 54. If the deque is in a crossing state then:

1. C_T and C_B are neighbors.
2. $\exists p$ such that $p@\langle 26\ldots 29, 31\ldots 33, 36\rangle$.

Note that by Conjecture 55 we already know that if the deque is in a crossing state, then N_T and N_B are either the same node, or adjacent nodes in `Ordered`. The following invariants will be used for the proof of Conjecture 54, which is given in Sect. 4.6.

Lemma 41 *Consider a transition $s \xrightarrow{a} s'$ where a is an execution of p.7, then: $s'.C_B = LeftNeighbor_{s'}(s.C_B)$.*

Corollary 43 *If $s \xrightarrow{a} s'$, where a is an execution of p.33, then: $s'.C_T = s.C_T$.*

Based on Corollary 43, in the rest of proof we do not regard Line 33 as one of the statements that may modify C_T.

Invariant 44. If $p@\langle 25\ldots 29, 31\ldots 34, 36\ldots 38\rangle$, then $Cell(p.newBotVal) = RightNeighbor(Cell(p.oldBotVal))$

Lemma 45 *Let $s \xrightarrow{a} s'$ be a step of the algorithm, and suppose a is an execution of Line 29 or Line 36, then: $s'.C_B = LeftNeighbor_s(s.C_B) = LeftNeighbor_{s'}(s.C_B)$.*

Invariant 46. *If $p@\langle 10, 12\ldots 17\rangle \wedge p.currTop = Top \wedge C_T \preceq C_B$ then:*

1. $(p@10 \wedge (Cell(p.currBottom) = C_T \vee Cell(p.currBottom) = RightNeighbor(C_T)))$, or
2. (a) $C_T = C_B$, and
   (b) $\exists p'$ such that: $p'@26 \vee (p'@\langle 27, 28, 31\ldots 33\rangle \wedge p'.currTop = Top)$.

Corollary 47 $p@\langle 12\ldots 17\rangle \wedge C_T \prec C_B \Rightarrow p.currTop \neq Top$.

Proof Since $p@\langle 12\ldots 17\rangle \wedge C_T \prec C_B$, if $p.currTop$ = `Top` we can apply Invariant 46 and get $C_T = C_B$, which contradict the assumption that $C_T \prec C_B$. □

Invariant 50. *If $p@\langle 50, 17\rangle \wedge (p.currTop = Top)$, then $Cell(p.newTopVal) = LeftNeighbor(Cell(p.currTop))$.*

Invariant 51. *If $p@\langle 27, 28, 31\ldots 33\rangle \wedge (p.currTop \neq Top)$ then $\neg(C_T \prec C_B) \vee (Cell(p.currTop) = C_B)$.*

Invariant 52.

1. $s.p@\langle 29, 30\rangle \Rightarrow (Cell(s.p.currTop) = Cell(s.p.oldBotVal))$.
2. $s.p@\langle 31\ldots 39\rangle \Rightarrow (Cell(s.p.currTop) \neq Cell(s.p.oldBotVal))$.
3. $s.p@\langle 32\ldots 37\rangle \Rightarrow (Cell(s.currTop) = Cell(s.p.newBotVal))$.
4. $s.p@\langle 38, 39\rangle \Rightarrow (Cell(s.p.currTop) \neq Cell(s.p.newBotVal))$.

Invariant 53. *If $p@\langle 15\ldots 17\rangle \wedge p.currTop = Top$ then: $(p.nodeToFree \neq$* `NULL`*$) \Leftrightarrow (p@15 \vee p.currTopNode \neq p.newTopNode)$.*

4.6 Proof of Conjecture 54

In this section we prove Conjecture 54, which is one of the two main invariants of the algorithm. The proof for the other main invariant, Conjecture 55, is given in the full version of the paper [19], Later these invariants will be useful in the linearizability proof.

As explained in Sect. 4.1.3, we must be careful to avoid circular reasoning, because the proofs of some of the invariants proved so far use Conjectures 54 in the post-state of their inductive step. Accordingly, in the proof of Conjecture 54, we use Conjecture 55 both in the induction pre-state and post-state, but all other invariants and conjectures are used only in the induction pre-state.

Conjecture 54. *If the deque is in a crossing state then:*

1. *C_T and C_B are neighbors.*
2. *$\exists p$ such that $p@\langle 26\ldots 29, 31\ldots 33, 36\rangle$.*

Proof Initially the deque is constructed such that $Cell$ (`Bottom`) $= Cell$(`Top`) so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s. Note that by Conjecture 55, $N_T \in$ `Ordered`$\wedge N_B \in$ `Ordered`and therefore: $\neg(C_T \prec C_B) \Leftrightarrow C_B \preceq C_T$. That also means that if we have, for example, $s.C_B \preceq_s s.C_T \wedge s'.C_B \prec_{s'} s.C_B \wedge s'.C_T = s.C_T$, it implies that $s'.C_B \prec_{s'} s'.C_T$, since neither $s.C_T$ nor $s.C_B$ can be removed from `Ordered` by the transition, and therefore by Lemma 33 the order between them cannot be changed.

- We first consider transitions that may establish the antecedent. Then we have $s.C_B \preceq_s s.C_T$ and $s'.C_T \prec_{s'} s'.C_B$, which implies by Lemma 41 that a cannot be the execution of Line 7, and by Lemma 45 that it cannot be the execution of Line 29 or 36. Therefore we have two statements to consider: Line 25 and Line 17.

  1. If a is an execution of p.25: Then by Invariant 44 $s'.C_B = RightNeighbor(s.C_B)$. Since $s'.C_T = s.C_T \wedge s'.$`Ordered`$= s.$`Ordered`, then if the deque is in crossing state in s': $((s'.C_T \prec_{s'} s'.C_B) \wedge (s.C_B \preceq_s s.C_T) \wedge (s'.C_B = RightNeighbor(s.C_B))) \Rightarrow s'.C_T = LeftNeighbor(s'.C_B)$. Also, $s'.p@26$ and therefore the consequent holds in s'.
  2. If a is an execution of p.17: Then by Invariant 50 $s'.C_T = LeftNeighbor_s(s.C_T)$. Therefore if the deque is in crossing state in s': $((s.C_B \preceq_s s.C_T) \wedge (s'.C_T \prec_{s'} s'.C_B)) \Rightarrow (s.C_T = s.C_B = s'.C_B \wedge s'.C_T = LeftNeighbor_s(s.C_B)) \Rightarrow s'.C_T = LeftNeighbor_{s'}(s'.C_B)$.e By Invariant 46, $(s.p@17 \wedge s.C_T = s.C_B) \Rightarrow ((s.p.currTop \neq s.$`Top`$) \vee (\exists p' \neq p\ p'@\langle 26\ldots 29, 31\ldots 33, 36\rangle))$. Therefore it is either that the CAS fails and a does not modify `Top` (and therefore does not establish the antecedent), or that the consequent holds in s'.

- We now consider transitions that may falsify the consequent while the antecedent holds. Because the antecedent and the invariant holds in s, $\exists p\ s.p@\langle 26\ldots 29, 31\ldots 33, 36\rangle \wedge s.C_T = LeftNeighbor_s(s.C_B)$, and since the antecedent holds in s', $s'.C_T \prec_{s'} s'.C_B$. By Conjecture 55 if a falsifies $C_T = LeftNeighbor(C_B)$ then $s'.C_B \neq s.C_B \vee s'.C_T \neq s.C_T$. Therefore there are two types of transitions that might falsify the consequent: A modification of C_T or C_B that results in: $s'.C_T \neq LeftNeighbor_{s'}(s'.C_B)$), or an execution of a statement that results in $\forall p \neg s'.p@\langle 26\ldots 29, 31\ldots 33, 36\rangle$.

  1. If a is a modification of C_T or C_B: Then $s.C_T = LeftNeighbor_s(s.C_B) \wedge s'.C_T \prec_{s'} s'.C_B$ implies by Lemma 41 that a is not an execution of Line 7, and by Lemma 45 that it is not an execution of Line 29 or 36. Therefore it is left to consider executions of Line 25 and 17.

     By Invariant 1, $s.p@\langle 26\ldots 29, 31\ldots 33, 36\rangle \Rightarrow \forall p'\ \neg s.p'@25$, and therefore a is not an execution of Line 25. If a is an execution of p'.17, then by Corollary 47 $s.C_T \prec_s s.C_B \wedge s.p'.17 \Rightarrow s.p'.currTop \neq s.Top$, and therefore $s'.C_T = s.C_T \wedge s'.C_B = s.C_B$, a contradiction.
  2. Otherwise, since a is not a modification of C_T or C_B, then the only transitions that may falsify $p@\langle 26\ldots 29, 31\ldots 33, 36\rangle$ are executions of p.31 or p.33.

     – If a is an execution of p.31: Then a can falsify the consequent only if $s.C_T \prec s.C_B \wedge s'.p@38$. By Invariant 6 $s.$`Bottom`$= s.p.newBotVal$. There are two cases to be considered:

       (a) If $s.p.currTop \neq s.$`Top`, then by Invariant 51 $s.C_T \prec s.C_B \Rightarrow Cell(s.p.currTop) = s.C_B = Cell(s.p.newBotVal)$, and therefore $s'.p@32$ and the consequent holds in s'.

       (b) Otherwise, $Cell(s.p.currTop) = s.C_T$. By Invariant 52: $Cell(s.p.currTop) \neq Cell(s.p.oldBotVal)$. By Invariant 44: $Cell(s.p.oldBotVal) = LeftNeighbor_s(Cell(s.p.newBotVal)) = LeftNeighbor_s(s.C_B)$, and therefore $s.C_T \neq LeftNeighbor_s(s.C_B)$, a contradiction.

     – If a is an execution of p.33: Then $s'.C_B = s.C_B$ and by Corollary 43: $s'.C_T = s.C_T$. By Invariant 52 $Cell(s.p.currTop) = Cell(s.p.newBotVal)$. If $s.p.currTop \neq s.Top$ then the CAS fails and $s'.p@36$, so a does not falsify the consequent. Otherwise, $s.C_T = Cell(s.p.currTop) = Cell(s.p.newBotVal) = s.C_B$. Therefore $s.C_T =$

$s.C_B$, which implies $s'.C_T = s'.C_B$, so the antecedent does not hold in s'. □

Conjecture 55. *Let* $n = |\texttt{Ordered}| - 2$, *and* $\texttt{Ordered} = \{N_0, \ldots, N_{n+1}\}$. *Then:*

1. $\forall_{0 \le i \le n}\, N_i \to next = N_{i+1} \wedge N_{i+1} \to prev = N_i$.
2. *Exactly one of the following holds:*
   (a) $n \ge 0,\ N_0 = N_B, N_n = N_T$.
   (b) $n > 0,\ N_1 = N_B, N_n = N_T$.
   (c) $n = 0,\ N_0 = N_T, N_1 = N_B$.

4.7 Section: linearizability

In this section we show that our implementation is linearizable to a sequential deque. We assume a sequentially consistent shared-memory multiprocessor system[9]. For brevity we will consider only *complete execution histories*:

Definition 56 A *complete execution history* is an execution in which any operation invocation has a corresponding response (that is, the history does not contain any partial execution of an operation).

Since we later show that our algorithm is wait-free, linearizability of all complete histories implies linearizability of all histories as well.

The linearizability proof is structured as follows: In Sect. 4.7.1 we give the sequential specification of a deque, to which our implementation is linearized. In Sect. 4.7.2 we specify the linearization points, and in Sect. 4.7.3 we give the proof itself.

4.7.1 The deque sequential definition

The following is the sequential specification of the implemented deque:

1. Deque state: A deque is a sequence of values, called the *deque elements*. We call the two ends of the sequence the *left end* and the *right end* of the sequence.
2. Supported Operations: The deque supports the PushBottom, PopTop and PopBottom operations.
3. Operations' Sequential Specifications: The following two operations may be invoked *only* by one process, which we'll refer to as the *deque's owner process*:

   - *PushBottom(v):* This operation adds the value v to the left end of the deque, and does not return a value.
   - *PopBottom:* If the deque is not empty, then this operation removes the leftmost element in the deque and returns it. Otherwise, it returns EMPTY and does not modify the state of the deque.

   The following operation may be invoked by *any process*:

   - *PopTop:* This operation can return ABORT, given the rule stated by Property 57; if the operation returns ABORT it does not modify the deque state. Otherwise, if the deque is empty, the operation returns EMPTY and does not modify the state of the deque, and if the deque is not empty, the operation removes the rightmost value from the deque and returns it.

9 In practice, we have implemented our algorithm for machines providing only a weaker memory model, which required insertion of some memory barrier instructions to the code.

Property 57. *In any sequence of operations on the deque, for any PopTop operation that has returned ABORT, there must be a corresponding Pop operation (i.e. a PopTop or PopBottom operation), which has returned a deque element. For any two different PopTop operations executed by the same process that return ABORT, the corresponding successful Pop operations are different.*

We have permitted the PopTop operation to return ABORT because in practical uses of work stealing deques, it is sometimes preferable to give up and try stealing from a different deque if there is contention. As we prove later, our algorithm is wait-free. We also show that if the ABORT return value is not allowed (that is, if the PopTop operation retries until it returns either EMPTY or the rightmost element in the deque), then our algorithm is lock-free.

4.7.2 The linearization points

Before specifying the linearization points of our algorithm we must define the *Physical Queue Content* (henceforth PQC): a subset of the ordered nodes' cells (Sect. 4.3, Definition 14), which as we later show, at any given state stores exactly the deque elements.

Definition 58 The *PQC* is the sequence of cells that lie in the half-open interval $(C_B \cdots C_T]$ according to the order $\prec$.

By the definition of the order $\prec$ (Definition 32), $C \in PQC \Rightarrow Node(C) \in \texttt{Ordered}$. By Corollary 15 $Node(C_B) = N_B \in \texttt{Ordered} \wedge Node(C_T) = N_T \in \texttt{Ordered}$, and therefore $(C_B \preceq C_T) \vee (C_T \prec C_B)$ holds. Also note that the PQC is empty if and only if $C_T \preceq C_B$. Specifically, the PQC is empty if the deque is in a crossing state (Definition 37).

The following claim is needed for the definition of the linearization points:

Claim 59. *Suppose that an execution of the PopTop operation does not return ABORT or EMPTY. Then the PQC was not empty right after the operation executed Line 9.*

Definition 60 *Linearization Points:*

PushBottom *The linearization point of this method is the update of* `Bottom` *at Line 7.*

PopBottom *The linearization point of this method depends on its returned value, as follows:*

- `EMPTY`: *The linearization point here is the read of* `Top` *at Line 26.*
- *A deque entry: The linearization point here is the update of* `Bottom` *at Line 25.*

PopTop *The linearization point of this method depends on its return value, as follows:*

- `EMPTY`: *The linearization point here is the read of* `Bottom` *pointer at Line 9.*
- `ABORT`: *The linearization point here is the statement that first observed the modification of* `Top`*. This is either the CAS operation at Line 17, or the read of* `Top` *at Line 11.*
- *A deque entry: If the PQC was not empty right before the CAS statement at Line 17, then the linearization point is that CAS statement. Otherwise, it is the first statement whose execution modified the PQC to be empty, in the interval after the execution of 9, and right before the execution of the CAS operation at Line 17.*[10]

Claim 61. The linearization points of the algorithm are well defined. That is, for any PushBottom, PopTop, or PopBottom operation, the linearization point statement is executed between the invocation and response of that operation.

Proof By examination of the code, all the linearization points except the one of a PopTop operation that returns a deque entry are well defined, since they are statements that are always executed by the operation being linearized. In the case of a PopTop operation, if the linearization point is the CAS statement, then it is obvious. Otherwise, the PQC was empty right before the execution of this successful CAS operation, and by Claim 57 the PQC was not empty right after the PopTop operation executed Line 9. Therefore there must have been a transition that modified the PQC to be empty in this interval, and this transition corresponds to the linearization point of the PopTop operation. □

4.7.3 The linearizability proof

In this section we show that our implementation is linearizable to the sequential deque specification given in Sect. 4.7.1. For this we need several lemmas, including one that shows how the linearization points of the deque operations modify the PQC, and one that shows that the PQC is not modified except at the linearization point of some operation.

[10] Note that the linearization point of the PopTop operation in this case might be the execution of a statement by a process other then the one executing the linearized PopTop operation. The existence of this point is justified in Claim 59.

Lemma 64 *Consider a transition $s \xrightarrow{a} s'$ where a is a statement execution corresponding to a linearization point of a deque operation. Then:*

- *Case 1: If a is the linearization point of a PushBottom(v) operation, then it adds a cell containing v to the left end of the PQC.*
- *Case 2: If a is the linearization point of a PopBottom operation: let R be the operation returned value:*[11]
  - *If R is EMPTY, then the $s.PQC = s'.PQC = \emptyset$.*
  - *If R is a deque entry, then R is stored in the leftmost cell of $s.PQC$, and this cell does not belong to $s'.PQC$.*
- *Case 3: If a is the linearization point of a PopTop operation: let R be the operation return value:*
  - *If R is EMPTY, then the $s.PQC = s'.PQC = \emptyset$.*
  - *If R is a deque entry, then R is stored in the rightmost cell of $s.PQC$, and this cell does not belong to $s'.PQC$.*

Definition 68 A *successful Pop operation* is either a PopTop or a PopBottom operation that did not return EMPTY or ABORT.

Lemma 69 *Consider a transition $s \xrightarrow{a} s'$ that modifies the PQC (that is, $s.PQC \neq s'.PQC$), then a is the execution of a linearization point of either a PushBottom operation, a PopBottom operation, or a successful PopTop operation.*

The following lemma shows that the ABORT property holds:

Lemma 70 *In any complete execution history, for any PopTop operation that has returned ABORT, there is a corresponding Pop operation (that is, a PopTop or PopBottom operation), which has returned a deque element. For any two different PopTop operations executed by the same process that returned ABORT, the corresponding successful Pop operations are different.*

The Linearizablity Theorem: Using Lemmas 64, 69, and 70, we now show that our implementation is linearizable to the sequential deque specification given in Sect. 4.7.1:

Theorem 71 *Any complete execution history of our algorithm is linearizable to the sequential deque specification given in Sect. 4.7.1.*

Proof Given an arbitrary complete execution history of the algorithm, we construct a total order of the deque operations by ordering them in the order of their linearization points. By Claim 61, each operation's linearization point occurs after it is invoked and before it returns, and therefore the total

[11] We can refer to the returned value of an operation since we're dealing only with complete histories.

order defined respects the partial order in the concurrent execution.

It is left to show that the sequence of operations in that total order respects the sequential specification given in Sect. 4.7.1. We begin with some notation. For each state s in the execution, we assign an *abstract deque state*, which is achieved by starting with an empty abstract deque, and applying to it all of the operations whose linearization points occurs before s in the order in which they occur in the execution.

We say that the PQC sequence *matches the abstract deque sequence* in a state s, if and only if the length of the abstract deque state and the length of the PQC (denoted $length(PQC)$) are equal, and for all $i \in [0..length(PQC))$, the data stored in the ith cell of the PQC sequence is the ith deque element in the abstract deque state.

We now show that in any state s of the execution, the PQC matches the abstract deque sequence in s:

1. Both sequences are empty at the beginning of the execution.
2. By Lemma 69, any transition that modifies the PQC is the linearization point of a successful PushBottom, PopBottom, or PopTop operation, and therefore it also modifies the abstract deque state.
3. By Lemma 64, the linearization point of a PushBottom operation adds a cell containing the pushed element to the left end of the PQC, the linearization point of a successful PopBottom operation removes the cell containing the popped element from the left end of the PQC, and the linearization point of a successful PopTop operation removes the cell containing the popped element from the right end of the PQC.

Therefore by induction on the length of the execution, and the abstract deque operation specification given in Sect. 4.7.1, the PQC sequence matches the abstract deque sequence in any state s of the execution.

By Lemma 70 the ABORT property holds. By Lemma 64 a PopBottom operation returns the leftmost value in the PQC if it is not empty or EMPTY otherwise, and if the PopTop operation does not return ABORT, then it returns the rightmost value in the PQC if it is not empty, or EMPTY otherwise. Therefore, since the PQC sequence matches the abstract deque sequence, the operations return the correct values according to the sequential specification given in Sect. 4.7.1, which implies that our implementation is linearizable to this sequential specification. □

4.8 The progress properties

Theorem 72 *Our deque implementation is wait-free.*

Proof Our implementation of the deque does not contain any loops, and therefore each operation must eventually complete. □

The reason our algorithm is wait-free is that we have defined the ABORT return value as a legitimate one. However, in many cases we may want to keep executing the PopTop operation until we gets either a deque element or the EMPTY return value. The following theorem shows that our implementation is lock-free even if the PopTop operation is executed until it returns such a value.

Definition 73 A *legitimate value* returned by a Pop operation is either a deque element or EMPTY.

Theorem 74 *Our deque implementation, where the PopTop operation retries until it returns a legitimate value, is lock-free.*

Proof The Abort property proven by Lemma 70, implies that every two different PopTop operations by the same process that returned ABORT have two different Pop operations that returned deque elements. Thus if a PopTop operation infinitely retries and keep returning ABORT, there must be an infinite number of Pop operations that returned a legitimate value. Therefore if a PopTop operation fails to complete, there must be an infinite number of a successful Pop operations. □

Theorem 75 *Our algorithm is a lock-free implementation of a linearizable deque, as defined by the sequential specification in Sect. 4.7.1.*

Proof Theorem 71 states that our implementation is linearizable to the sequential specification given in Sect. 4.7.1. Theorem 74 showed that the implementation is lock-free. □

5 Conclusions

We have shown how to create a dynamic memory version of the ABP work stealing algorithm. It may be interesting to see how our dynamic-memory technique is applied to other schemes that improve on ABP-work stealing such as the locality-guided work-stealing of Blelloch et. al. [4] or the steal-half algorithm of Hendler and Shavit [9].

References

1. Lev, Y.: A Dynamic-Sized Nonblocking Work Stealing Deque. MS thesis, Tel-Aviv University, Tel-Aviv, Israel (2004)
2. 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, pp. 237–245. ACM Press (1991)
3. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems **34**, 115–144 (2001)
4. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: ACM Symposium on Parallel Algorithms and Architectures, pp. 1–12 (2000)

5. Flood, C., Detlefs, D., Shavit, N., Zhang, C.: Parallel garbage collection for shared memory multiprocessors. In: Usenix Java Virtual Machine Research and Technology Symposium (JVM '01), Monterey, CA (2001)
6. Leiserson, P.: Programming parallel applications in cilk. SINEWS: SIAM News **31** (1998)
7. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM **46**, 720–748 (1999)
8. Knuth, D.: The Art of Computer Programming: Fundamental Algorithms. 2nd edn. Addison-Wesley (1968)
9. Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (2002)
10. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. Technical report, Sun Microsystems – Sun Laboratories (2004) To appear.
11. Agesen, O., Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Moir, M., Shavit, N., Steele, G.: DCAS-based concurrent deques. Theory of Computing Systems **35**, 349–386 (2002)
12. Martin, P., Moir, M., Steele, G.: Dcas-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories (2002)
13. Greenwald, M.B., Cheriton, D.R.: The synergy between non-blocking synchronization and operating system structure. In: 2nd Symposium on Operating Systems Design and Implementation, pp. 123–136. Seattle, WA (1996)
14. Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments (extended abstract). In: Measurement and Modeling of Computer Systems, pp. 266–267 (1998)
15. Arnold, J.M., Buell, D.A., Davis, E.G.: Splash 2. In: Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 316–322. ACM Press (1992)
16. Papadopoulos, D.: Hood: A user-level thread library for multi-programmed multiprocessors. In: Master's thesis, Department of Computer Sciences, University of Texas at Austin (1998)
17. Prakash, S., Lee, Y., Johnson, T.: A non-blocking algorithm for shared queues using compare-and-swap. IEEE Transactions on Computers **43**, 548–559 (1994)
18. Scott, M.L.: Personal communication: Code for a lock-free memory management pool (2003)
19. Hendler, D., Lev, Y., Moir, M., Shavit, N.: A dynamic-sized non-blocking work stealing deque. Technical Report TR-2005-144, Sun Microsystems Laboratories (2005)