An Unbounded Nonblocking Double-ended Queue

Matthew Graichen  Joseph Izraelevitz  Michael L. Scott

Computer Science Department
University of Rochester
Rochester, NY, USA
matt.graichen@gmail.com, {jhi1, scott}@cs.rochester.edu

Abstract—We introduce a new algorithm for an unbounded concurrent double-ended queue (deque). Like the bounded deque of Herlihy, Luchangco, and Moir on which it is based, the new algorithm is simple and obstruction free, has no pathological long-latency scenarios, avoids interference between operations at opposite ends, and requires no special hardware support beyond the usual compare-and-swap. To the best of our knowledge, no prior concurrent deque combines these properties with unbounded capacity, or provides consistently better performance across a wide range of concurrent workloads.

Index Terms—parallel processing; parallel algorithms; nonblocking algorithms;

I. INTRODUCTION

Container classes—stacks, queues, sets, and maps—are among the most widely used abstractions in both sequential and parallel computing. Parallel containers are generally considered correct only if their operations (method calls) are linearizable [1], meaning that they appear, from the perspective of all threads, to take effect atomically at some point between their call and return.

Atomicity of container operations may be achieved using locks or via nonblocking programming techniques. An object (class instance) is said to be nonblocking if no thread, through inaction, can prevent other threads from making progress [2]. Nonblocking objects often scale better than those based on coarse-grain locks. They are also much less prone to performance anomalies caused by preemption.

Nonblocking objects can be classified in terms of liveness properties. A wait-free object guarantees that any thread \( t \) executing one of its methods will complete the operation in a bounded number of steps. A lock-free object guarantees that some thread will complete an operation in a bounded number of \( t \)'s time steps. Finally, an obstruction-free [3] object guarantees that \( t \) will complete its operation in a bounded number of steps in the absence of contention. Wait-free objects preclude starvation. Lock-free objects preclude livelock. Obstruction-free objects require some sort of external mechanism to prevent livelock; the “randomization” inherent in most schedulers typically suffices.

Double ended queues (or deques) are lists that support push and pop operations at both ends. In this work, we introduce a new concurrent, nonblocking deque that avoids most of the limitations of existing implementations. Our work builds on the array-based, bounded HLM deque of Herlihy, Luchangco, and Moir [3]. Like that deque, ours is obstruction free, and allows operations on opposite ends to execute concurrently without interference. We allocate additional arrays as needed, however, to eliminate the bound on capacity. We also incorporate an optional elimination optimization [4], [5] to combine matching pushs and pops. Experimental results confirm that our deque maintains low latency while providing higher throughput than that of any known alternative.

A. Related Work

Nonblocking stacks and queues have been studied extensively, beginning with the Treiber stack [6] and the Michael & Scott queue [7]. Subsequent breakthroughs include the introduction of elimination, which allows matching operations to complete without serializing [4], [8], [9]; flat combining [10], which improves cache locality but reintroduces blocking; and optimistic fetch-and-increment [11], which greatly reduces contention.

Work-stealing queues [12] are commonly used to dispatch tasks to worker threads. Access to one end of the structure is limited to pushes and pops by a distinguished owner thread. The other end supports concurrent access, but only for pops. These restrictions are important, and can be leveraged to significantly enhance performance [13], [14].

The first fully functional nonblocking concurrent deque is due to Michael [15]. It is surprisingly complex, and suffers from contention between the two ends of the queue. The lock-free deque of Sundell and Tsiolas [16] avoids this contention, but it can require lengthy helping operations if threads stall at inopportune times. The recent time-stamp deque of Dodds et al. [5], [17] has significantly better throughput, but at the expense of intentionally elevated latency, introduced to facilitate elimination. Finally, the obstruction-free array-based deque of Herlihy, Luchangco, and Moir [3] (the HLM deque) is refreshingly simple, but bounded. We employ this deque as the basis of a new, unbounded alternative.

II. UNBOUNDED DOUBLE ENDED QUEUE

A. Overview

At a global level, our deque consists of a doubly-linked list of HLM linear bounded deques, with careful handling of linking and unlinking operations. Consequently, we begin by

This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Study. The authors would also like to thank Sean Brennan for his early exploration on this topic.
reviewing the behavior of the HLM deque. In our pseudocode, the keyword *tuple* indicates a single CAS-able value comprising several fields. We use the angle brackets { and } to indicate creating a tuple from individual field values.

1) **HLM Bounded Linear Deque:** Figure 1 shows the structure of the HLM bounded linear deque, and Figure 2 shows its type declarations [3]. The deque consists of a single array of special *hlm_slot* s. Each slot contains a single 64-bit tuple comprising a 32-bit data value and a 32-bit counter. The linear array stores nontrivial data tuples contiguously: the reserved LN (left NULL) and RN (right NULL) tuples on either side of the contiguous data tuples indicate empty slots. A simple obstruction-free two-CAS protocol is used to push and pop values at the edge of the contiguous span of data values. To push_left, for example, a thread first finds the left edge—the boundary between the rightmost LN tuple and the data tuple to its right. It then performs a pair of CASes, first to increment the count of the leftmost data tuple and then to replace the rightmost LN tuple with a new data tuple (see Figure 3). The two-CAS protocol verifies both that the operation was enacted on the left edge and that any concurrent left-edge–changing operation would fail. A similar two-CAS protocol is used for pop_left (see Figure 3), and operations on the right side are symmetric. An arbitrary *oracle* function is used to find edges, but the expectation is that the structure will store hints to the edges that are periodically updated.

2) **Unbounded Deque Structure:** Figure 4 shows the structure of our deque and Figure 5 declares the types. Within each array-based node in our doubly linked list, we store a linear array of *slots* similar to the HLM deque. Once again, the linear array stores data tuples contiguously: the reserved LN and RN tuples on either side of the contiguous data tuples indicate empty slots, but the overall deque may span multiple buffers. Our slots on the interior of a buffer, like those of the HLM deque, contain either a NULL or a datum; we call these *data slots*. The two slots at the ends of the buffer (the *borders*), however, contain either a NULL or a pointer to an adjacent node; we call these *link slots*.

---

1 Object LN, RN; // reserved 32 bit consts
2 int HLM_SIZE;
3 tuple hlm_slot{
4 Object val; // 32 bits
5 int ct; // 32 bits
6 };
7 class hlm_linear_deque {
8 hlm_slot[HLM_SIZE] array;
9 };

---

Given that the deque’s contents are contiguous and accessed at both ends, we should like to discuss the algorithm in an end-agnostic way. Consequently, we use *inward* to mean the direction towards the center of the contiguous span of data values, and we use *outward* to mean the direction away from it. For example, the left edge of the contiguous span of data values lies between the *innermost* LN and the *outermost* data value (Figure 4). Similarly for the right edge.

We use explicit *hints* to find edges, and we update these upon completion of each operation. To accommodate temporarily incorrect hints, we use two auxiliary functions: left_oracle() and right_oracle(). As in the HLM algorithm, these functions return an index that was correct sometime during the call [3]; in our code, they also return a pointer to the appropriate node.

The reserved values LS and RS are “sealed” values, added to facilitate node reclamation. Nodes are sealed in their innermost data slot. A sealed node acts as a “cap” on its end of the deque and prevents outward growth in that direction; e.g., a left seal (LS) may be added in the rightmost data slot of a node (see Figure 9). Once a node has been sealed it can be removed from the deque and subsequently garbage collected.

3) **Transitions:** Modifications to the deque structure are made via a limited set of transition actions, each of which uses a short protocol to ensure that the transition’s preconditions are...
not violated by a concurrent transition. As in the push and pop operations of the HLM deque, our transitions rely on slot counters.

Unlike the HLM deque, our algorithm must accommodate node linking. Consequently, edges may arise in any of three places, as illustrated in Figure 7. The simplest case is said to be **interior**, where the edge occurs on the interior of a buffer. In the most complex, an edge **straddles** a pair of adjacent nodes, that is, it aligns with the link between them. In the third, intermediate case, the edge may coincide with a node **boundary**, where there is no adjacent node, yet the edge is at the border of the buffer between the outer link slot and the outer data slot.

We can group the transitions into three categories: list modifications, empty-checks, and the hint-update. List modifications (e.g., inserting a value) use a two-CAS protocol similar to that of the HLM deque to modify the deque state. Empty checks, which are read-only, employ a multi-step snapshot. Hint update transitions are performance optimizations; they do not affect the underlying list structure. The transitions are:

- **L1** An interior push (line 92 & Figure 3).
- **L2** An interior pop (line 170 & Figure 3).
- **L3** A straddling push (line 125 & Figure 9).
- **L4** A boundary pop (line 225 & Figure 10).
- **L5** Sealing a node (line 198 & Figure 10).
- **L6** Appending a new node (line 106 & Figure 9).
- **L7** Removing a sealed node (lines 132 and 209 & Figures 9 and 10).
- **E1** An interior empty check (line 132 & Figure 11).
- **E2** A straddling empty check (lines 194 and 205 & Figure 11).
- **E3** A boundary empty check (line 221 & Figure 11).
- **H** Updating the hint (line 60).

Our implementation uses these transitions to ensure that the deque is always in a valid state and that transitions are applied when the thread’s perspective of the deque indicates that they are appropriate.

In general, threads attempting an operation first use the hint to find the edge node and slot for their operation (e.g., a push_lefting thread looks for the leftmost value among the contiguous data slots). Once the thread finds the edge data value (called **ln**), it views the next slot outward (called **out**) and expects either a **null** value or a pointer. If it finds a pointer, it accesses the innermost data slot of its neighboring node (called **far**—effectively three slots outward from **ln**), and verifies that its neighbor points back at it by checking the neighbor’s link slot (**back**). Based on these few values (see Figure 8), the thread can determine if it has found an edge and what sort of edge it is (interior, straddling, or boundary). Based on the edge type, it applies the appropriate transition(s) to complete its overall operation.

**B. Implementation Details**

Figure 5 shows how the deque structure is initialized. To start the doubly-linked list we allocate an initial node. The left half of this node’s buffer is filled with **LN** values and the right half with **RN** values. Finally, we point both the **left_node_hint** and **right_node_hint** to this node and set the slot hints appropriately.

Since the push and pop operations are side-agnostic, from here on we will only discuss the details of push_left (Figure 6) and pop_left (Figure 12).
while (true) {
    // find edge
    int edge_idx;
    node edge_nd;
    node_hint hint_cpy = left_node_hint;
    ⟨edge_nd, edge_idx⟩ = l_oracle(hint_cpy);
    slot in_cpy = *in;
    slot* out = &edge_nd[edge_idx-1];
    slot out_cpy = *out;
    // check oracle’s edge
    if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)) {
        hint_l(hint_cpy, edge_nd, 1);
        return OK; }
    // end interior push
    if (edge_idx ≺ 1 && out_cpy.val ≺ LN) {
        continue;
    }||(edge_idx ≈SZ - 1 && in_cpy.val ≺ RN)) {
        return OK; }
    // end boundary or straddling edge
    continue;
    if (back.val ≺ RN) {
        continue;
    } // end method
    if (back.val ≺ LN) {
        continue;
    } else if (far_cpy.val == LS) {
        return OK; }
    if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)) {
        hint_l(hint_cpy, edge_nd, edge_idx-1);
        return OK; }
    // end straddling edge
    if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)) {
        hint_l(hint_cpy, edge_nd, edge_idx-1);
        return OK; }
    // end straddling edge
    if (far_cpy.val == LN) {
        return OK; }
    if (CAS(in, in_cpy, ⟨in_cpy.val, in_cpy.ct+1⟩)) {
        hint_l(hint_cpy, edge_nd, edge_idx-1);
        return OK; }
    // end method
}

Fig. 6: Unbounded deque push_left() (symmetric code for push_right())

1) push_left: Our push_left operation (Figure 6) illustrates the general pattern for operations: find the edge, read the in, out, and far pointers, then apply the appropriate transitions.

If push_left finds an interior edge based on the edge_idx, it simply attempts to apply an interior push which follows exactly the two-CAS protocol of the HLM deque (Figure 3). If the interior push fails, then the edge may have moved, so the entire operation retries (including re-finding the edge).

If push_left instead discovers that the outermost data value is in an outermost data slot, it is possible that the deque’s left edge is in the straddling or boundary case. In the boundary case, the thread uses the append transition (Figure 9) to add a new node to the left of the current deque. The append transition is effectively an HLM push but the pushed “value” is a pointer in the link slot. By using the two-CAS protocol for append, the thread ensures that the boundary edge has not changed.

Alternatively, if the deque’s left edge is in the straddling case, then edge_nd has a left neighbor. We copy the innermost data slot of our left neighbor (far) and verify that the right link slot of our neighbor (back) indeed points back to our starting node. If the edge is a valid straddling edge, far slot is either LS or LN. If it is LN we can push our value into it using the straddling push transition (Figure 9). However, if it is LS then we must remove our left neighbor to progress using the remove transition (Figure 9); then we begin the operation again. Both operations may be interrupted—and forced to start over—by a concurrent straddling operation.

2) pop_left: Our pop_left method (Figure 12), like push_left, follows the established pattern: find the edge, read the in, out, and far pointers, then apply the appropriate
Fig. 9: Non-interior push operations

If \texttt{pop\_left} finds an interior edge based on the \texttt{edge\_idx}, it applies the interior pop of the HLM deque (Figure 3).

If \texttt{pop\_left} finds that the outermost data value is in an outermost data slot, it is again possible that the deque’s left edge is in the straddling or boundary case.

If the deque’s left edge is in the straddling case, \texttt{edge\_nd} has a left neighbor pointed to by the left link slot (\texttt{out}). A straddling edge for a pop operation triggers a series of transitions we call the “straddling pop progression” (see Figure 10). Assuming that we truly have a straddling edge, we read the innermost data slot of our left neighbor (\texttt{far}), and verify that the left neighbor points back (\texttt{back}). If the far slot is not sealed, we seal it using the seal transition.

Now that our left neighbor is sealed (by us or another thread), we remove the left neighbor node with the remove transition. Finally, since we have no left neighbor, we have a boundary edge.

In the boundary edge case, whether because the current thread found a boundary edge or because it followed the straddling pop progression, the thread uses the boundary pop transition to remove from the outermost data slot (Figure 10).

For all transitions in \texttt{pop\_left}, if we notice that the leftmost data value (\texttt{in}) is either RN or RS, we use a snapshot empty check to verify the deque is empty and that our edge remains valid (Figure 11). This empty check also prevents us from having two sealed nodes pointing at each other, since the second node to get sealed will instead return \texttt{EMPTY}.

C. Memory Management

Our pseudocode for the most part ignores memory management, except for the \texttt{retire} method. After a thread has detached a node from the deque, it keeps it in a thread local retirement list. After no threads are able to access the node, we know that the node can be freed.

Our memory reclamation scheme leverages the invariant that removed nodes cannot be accessed from active (unremoved) nodes; the remove operation breaks the link in this direction (Figure 10). Consequently, when a node is retired, we update the global hints to any active node using a CAS. Any future thread that reads these new hints cannot trace to our retired node, and, if all node\_hint updates use a CAS, our node will be inaccessible from all future hinted nodes. We use hazard pointers [18] to track threads with earlier hints as they traverse the chain in the \texttt{oracle} function.

D. Optimizations

We enacted one key optimization not discussed in the algorithm description. As noted by Dodds et al. [5], deques, like stacks, can use elimination arrays [4] on each end—a like-sided push and pop never need to actually access the deque structure and instead can “cancel out” if they overlap in time.

We use modified elimination arrays (Figure 13) to optimize our algorithm, one on each side of the deque (\texttt{l\_elim} and \texttt{r\_elim}). The elimination class exports three methods: an \texttt{insert} method, which stores details of the operation in a thread-local elimination slot, a \texttt{scan} method, which scans the entire elimination array searching for opposite operations to eliminate with, and a \texttt{remove} method, which removes the
operation from the slot, possibly after it has combined with an opposite operation.

Figure 13 shows the use of our elimination arrays for the pop_left operation (other methods are similar). When they begin an operation, threads insert the details of their operation into a thread local slot in the appropriate elimination array. This information remains in the array while the thread searches for an edge in the oracle function. Once a thread finds an edge, it removes itself from its slot. Conceivably, the thread has already eliminated with an overlapping opposite operation, in which case the thread has completed. If the thread does not eliminate, it begins to attempt transitions on the actual deque. Should the thread fail to complete its operation on the actual deque, it scans the elimination array for opposite operations to combine with. Should this scan fail, the thread reinserts itself into the elimination array and retries the entire operation from the beginning.

Our elimination design moves the high overhead elimination scan off the critical path, reduces contention on the main deque, and leverages the latency of the oracle function as a strength. Furthermore, it relieves us of the need to pick an appropriate period for threads to wait in the elimination array before trying the actual deque.

III. Correctness

We here present an informal proof of our algorithm’s correctness, discussing both safety and liveness guarantees. For clarity of exposition, we elide discussion of garbage collection in the proof.

A. Safety

To prove our algorithm correct, we must explain the desired sequential semantics, identify linearization points for our operations, and, finally, demonstrate that any realizable parallel execution has the same observable behavior as a sequential execution performed in linearization order.

1) Sequential Semantics: A sequential deque exports push_left, pop_left, push_right, and pop_right methods with the usual semantics. At any given point in
Elimination optimization of `popleft()`, modifications bolded

time, the deque has an abstract state comprising a finite sequence of elements; initially this sequence is empty. A `push_right(x)` operation moves the abstract state from \( S \) to \( Sx \), by concatenating \( x \) to the end of \( S \). A `push_left(x)` operation moves the abstract state from \( S \) to \( Sx \), by concatenating \( x \) to the beginning of \( S \). When \( S \) is empty, `pop_right` and `pop_left` return `EMPTY` and leave \( S \) unchanged. If the abstract state is \( Sx \), `pop_right` moves to \( S \) and returns \( x \). Similarly, `pop_left` moves from \( xS \) to \( S \) and returns \( x \). A sequential history is valid if we can identify abstract states \( S_i, \forall i \), such that \( S_0 \) is empty and the \( i \)th operation in the history moves from \( S_{i-1} \) to \( S_i \).

2) Linearization points: To prove the safety of our deque, it is sufficient to show that in any realizable concurrent history it is possible to identify linearization points (each between the call and return of its operation) such that the history has the same observable behavior (i.e., return values) as a sequential execution that performs the same operations in linearization order on a sequential deque.

1) A push operation linearizes when the second CAS of its insertion transition succeeds (line 92 for an interior push, line 125 for a straddling push, and line 106 for a node append).

2) A pop operation that finds a nonempty container linearizes when the second CAS of its removal protocol succeeds (line 170 for an interior pop and line 225 for a boundary pop).

3) A pop operation that finds an empty container linearizes when the final read of its empty check succeeds (line 166 for an interior empty check, line 194 or line 205 for a straddling empty check, and line 221 for a boundary empty check).

Theorem 1 (Linearizability). Any realizable well-formed history of our concurrent deque containing only completed operations is equivalent to a valid well-formed history of a sequential deque.

Proof. By induction. In the base case, our deque begins in a quiescent state with a single empty node. The left half of the node’s array contains LN values, while the right side of the array contains RN values. Both exterior hints point to the single node, and both of the node’s interior hints point to the center.

Our induction invariant requires that the deque is well-formed as described below. All non-removed (active) nodes are doubly-linked to their neighbors and form a contiguous chain. For the active nodes, any empty slot on the left contains the LN value, and any empty slot on the right contains the RN value. The innermost non-value slots may be an LS or RS if the associated edge is a straddling edge. Interior slots of active nodes between the edges must contain stored values (non-NULL non-SEALED values). We call the contiguous chain of active nodes the active chain.

Sealed nodes are sealed on either the right or left side (right sealed or left sealed) and lie on either side of the active chain. Sealed nodes cannot interrupt the active chain, but may be on one or both ends. A sealed node, on its inward side, may be singly linked inward toward the active chain (its inward side neighbor might not point back to it). A sealed node, on its inward side, is the neighbor of either an active node, another sealed node which has been sealed on the same side, or another sealed node which has been sealed on the opposite side but does not point back.

Examination of the code confirms that the deque’s state can be changed by a small set of transitions (Section II-A3). These transitions (or concurrent sets of them) are our induction steps; they transition the deque from one well-formed state to another.

All transitions act on one side of the deque and have analogous operations on the other side. While the empty check transitions do not actually change the underlying structure,
they can still be considered a valid “action” on the deque.

The non-structural edge transitions (L1–L5) use the double CAS protocol to change the edge of the deque without changing the underlying doubly linked list. For a given side, these operations will all conflict with each other: only one of these transitions can succeed at once. If the deque contains zero or one elements, these transitions also conflict with the opposite side. The empty check transitions (E1–E3) use a snapshot read protocol to ensure that the edge does not change: they conflict with the non-structural edge transitions (L1–L5) but not with other empty check transitions (E1–E3).

The linking transitions (L6–L7) use a slightly different double CAS protocol, yet this modified double CAS protocol also conflicts with non-structural transitions (L1–L5) and empty check transitions (E1–E3). The append transition (L6) will fail if any other thread changes the edge: moving the edge inward will cause the first CAS to fail, while moving the edge outward will cause the second CAS to fail. The boundary empty check’s triple read protocol will be aborted if it overlaps with an append.

The remove transition (L7) also conflicts with the non-structural transitions (L1–L5) and all empty checks (E1–E3). Based on the initial condition for the remove transition, a non-structural transition (L1–L5) which attempts to move the edge outward or inward will fail due to the sealed slot as will any empty check (E1–E3). Append and remove transitions conflict with each other as they have different initial conditions and they both trivially conflict with themselves. Finally, append and remove transitions will never conflict with their opposite side analogues if the node buffer size is sufficiently large.

The hint update (H) can occur simultaneously with any other transition. Examination of the code confirms that the hint always points to some valid node, either active or inactive. From either node type, a traversal of a well-formed deque can always find the edges of an active chain. Thus a hint update can safely occur simultaneously with any other transitions.

By induction, we know that the deque is always well-formed. The linearized history is the induced order of transitions with non-exported transitions (sealing, detaching, hint updating) dropped. The linearized history is correct since the contents of the active chain are always equivalent to a sequential deque with an equivalent history.

B. Liveness and Contention Freedom

Theorem 2 (Obstruction Freedom). The presented unbounded concurrent deque is obstruction-free.

Proof. Only three loops exist in the code, one each in the push and pop methods, and one in the oracle method. In the absence of contention, the snapshots taken by threads from the local variables in, out, and far remain valid. Examination of the code confirms that, for any well-formed unbounded deque state, some transition will be chosen. For a push operation, assuming no contention, only a SEALED neighbor will cause a retry. After removing the neighbor, a subsequent append will complete the push. For a pop operation, assuming no contention, a series of one or more transitions will be performed, ending in a successful pop.

In the absence of contention, assuming a well-formed deque, we can always follow sealed nodes back to the active chain since they always point to nodes that were sealed after them. Eventually the oracle function will find the edge node and its slot by following this chain.

IV. Performance Results

For data structure evaluation, we used an Intel machine with two eighteen-core, two-way hyper-threaded Intel Xeon E5-2699 v3 processors at 3.6 GHz (i.e., with up to 72 hardware threads). Every core’s L1 and L2 caches are private to that core (shared between hyper-threads); the L3 cache (45 MB) is shared across all cores of a single processor. The machine runs Fedora Core 19 Linux. Tests were performed in a controlled environment when we were the sole users of the machine. Threads were pinned to cores in a consistent order for all experiments: one thread per physical core on the first processor (1–18), then one thread for each additional hyper-thread on that processor (19–36), then one thread per core (37–54) and one per additional hyper-thread (55–72) on the second processor. Code was written in C++ and compiled at the -03 optimization level using g++ 4.8.2. When a nonblocking memory allocator would improve performance, we used one adapted from the Rochester Software Transactional Memory (RSTM) package [19].

Our test comprises a micro-benchmark, run for a fixed period of time, in which every thread repeatedly executes some method of the deque, uniformly and randomly choosing the method at each iteration. We ran experiments in which threads use the deque in a Stack, Queue, and Deque access pattern.
set

Throughput (M ops/sec) on Queue

Throughput (M ops/sec) on Stack

10.0

0.0

5.0

7.5

2.5

Fig. 15: Throughput for stack access pattern

Fig. 16: Throughput for queue access pattern

(e.g., under Stack, threads chose only between push_left and pop_left). Each configuration was tested five times; we report the average.

We tested several different deques—SGLDeque: a deque protected by a single global test-and-test_and_set lock; FCCDeque: a concurrent deque using flat combining with an exponential backoff lock [10]; MDMADeque, STDeque: the lock free deque of Maged Michael [15] and the lock free deque

of Sundell and Tsigas [16] respectively, both deques with and without exponential backoff elimination arrays [4]; TSDeque-FAI, TSDeque-HW: the time stamping deque of Dodds et al. [5], using a fetch-and-increment counter and the hardware cycle counter respectively — both versions use an elimination style optimization; and OFDeque: the obstruction free deque which is the topic of this paper, shown with and without the elimination optimization. We chose 1024 as a representative number of slots in each buffer; no significant performance impact was noted for different buffer sizes.

As can be seen in Figures 14, 15, and 16, our new concurrent OFDeque generally outperforms prior art across the range of thread counts and for most access patterns. Its single-thread throughput also exceeds that of all the nonblocking alternatives.

The elimination technique provides a significant boost for the Deque and Stack access patterns, allowing performance of the OFDeque within one socket to scale with the number of threads. Elimination is a general technique, and can also be applied to the MDMADeque and STDeque (the TSDeque already incorporates an elimination mechanism). Given high single-thread latency, however, the MDMADeque with elimination is still slower than the OFDeque, and in the STDeque, the fact that contention can happen after linearization prevents elimination from helping much.

For the Queue access pattern, where elimination is not generally feasible, flat combining achieves the best performance by maximizing cache locality and reducing contention, but the presented obstruction free deque generally outperforms all nonblocking alternatives. Note that on the Queue access pattern, elimination will generally hurt performance, since operations will never combine in this test, as seen in the OFDeque’s performance, unless it acts as a contention manager, as seen in the MDMADeque.

V. CONCLUSION

In conclusion, our algorithm provides a novel unbounded and obstruction free double ended queue construction. Our structure outperforms across the range of thread counts other state of the art nonblocking solutions and, for certain access patterns, also outperforms blocking solutions.

REFERENCES


