An Efficient Abortable-locking Protocol for Multi-level NUMA Systems

Milind Chabbi
Hewlett Packard Labs
milind.chabbi@hpe.com

Abdelhalim Amer
Argonne National Laboratory
aamer@anl.gov

Shasha Wen, Xu Liu
College of William and Mary
{swen, xl10}@cs.wm.edu

Abstract

The popularity of Non-Uniform Memory Access (NUMA) architectures has led to numerous locality-preserving hierarchical lock designs, such as HCLH, HMCS, and cohort locks. Locality-preserving locks trade fairness for higher throughput. Hence, some instances of acquisitions can incur long latencies, which may be intolerable for certain applications. Few locks admit a waiting thread to abandon its protocol on a timeout. State-of-the-art abortable locks are not fully locality aware, introduce high overheads, and unsuitable for frequent aborts. Enhancing locality-aware locks with lightweight timeout capability is critical for their adoption. In this paper, we design and evaluate the HMCS-T lock, a Hierarchical MCS (HMCS) lock variant that admits a timeout. HMCS-T maintains the locality benefits of HMCS while ensuring aborts to be lightweight. HMCS-T offers the progress guarantee missing in most abortable queuing locks. Our evaluations show that HMCS-T offers the timeout feature at a moderate overhead over its HMCS analog. HMCS-T, used in an MPI runtime lock, mitigated the poor scalability of an MPI+OpenMP BFS code and resulted in 4.3× superior scaling.

Keywords Synchronization, Spin lock, Queuing lock, MCS lock, Hierarchical lock, Abortable lock, Timeout lock.

1. Introduction

Multi-socket systems have become a norm rather than an exception in modern server and HPC establishments. State-of-the-art shared memory machines such as SGI UV [22] and HP Integrity Superdome X [10] are not only multi-socket but also multi-node. Multi-node, multi-socket systems along with the deep processor cache hierarchy lead to Non-Uniform Memory Access (NUMA) latencies. Maintaining locality of reference is critical to achieving high performance on NUMA architectures.

The emergence of NUMA architectures has propelled researchers to develop various "locality-aware" locking algorithms. Locality-aware locks prefer to handover a lock from the lock holder to the "nearest" lock requester instead of the "earliest" lock requester. This kind of locality bias reduces the indiscriminate migration of shared cache lines (accessed in the critical section and the lock's internal data structures) among different locality domains.

Hierarchical locks sacrifice fairness [5] to exploit locality. In a hierarchical lock, on a highly contended system with a deep NUMA hierarchy, a lock requester may have to wait for an unusually long time when a distant NUMA domain holds the lock. While spinning for a short time to acquire the lock from a neighbor is desirable to enhance locality and throughput, idle waiting for a long time hurts parallelism. The following use cases motivate the need for allowing a waiting thread to timeout and abandon the protocol: (1) A thread in an application with sufficient parallel slackness [9, 24] may desire to discontinue a work chunk blocked on long lock waiting and instead take up a different work chunk not subjected to serialization, (2) A low-priority process may want to temporarily abandon (or permanently abort) its lock request to expedite the lock handoff to a high-priority process, and (3) Database systems assume a deadlock when no progress is observed for a long time, and they require the timeout capability in their locks to unroll a transaction.

State-of-the-art abortable locks suffer from various limitations. They introduce blocking wait (e.g., [20]), have unbounded space and time complexities (e.g., [19] and [8]), lack full locality awareness (e.g., [20], [19], [8]), do not ensure starvation freedom (e.g., [8]), and incur high memory management costs (e.g., [19], [8]). Hierarchical queue-based spin locks with a timeout would offer greater flexibility for application programmers to exploit locality of shared data when available and abandon a long wait to exploit parallelism that exists elsewhere in the system. However, state-of-the-art hierarchical queue-based locks (e.g., [5]) lack the timeout capability. Few, if any, verify their correctness.

PPoPP '17 February 04 - 08, 2017, Austin, TX, USA
Copyright © 2017 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4493-7/17/02. . . $15.00
DOI: http://dx.doi.org/10.1145/3018743.3018768

We have designed the HMCS-T lock, which allows a waiting thread to abandon on timeout. The HMCS-T lock addresses the aforementioned challenges while being mindful of the principles of the HMCS lock—exploiting locality. It is challenging to incorporate the timeout capability in the HMCS lock because of the dependency of successors at each level on their predecessors to hand-off the lock through the hierarchical tree. A timeout may happen when a thread is waiting to acquire a lock at any level in the tree. Abandoning from an interior node in the tree is particularly challenging since it involves releasing already acquired lower-level locks. Wait loops at various places in the protocol pose daunting challenges in making the protocol non-blocking on timeout. Furthermore, attention needs to be paid to maintain locality in every step of the design. A design that addresses these complications needs rigorous correctness (mutual exclusion and livelock and deadlock freedom) and progress (starvation freedom) guarantees.

The key idea in the HMCS-T lock is that an abandoning thread leaves its queue node record [5] (aka `QNode`) intact in the MCS queue structure with a special status flag indicating that the owner has abandoned. When a lock holder notices an abandoned successor, it passes the lock to the next unabandoned successor. If no unabandoned successor is found at a level, the release proceeds to find a waiting thread at an ancestral level, if any, in the hierarchical tree. A disciplined, locality-preserving abandonment policy handles timeout while retaining the high performance of HMCS.

In HMCS-T, when an abandoned thread requests the same lock again, instead of bringing a new `QNode` record, it reuses its abandoned `QNode`. This design ensures a space complexity linear in the number of participants. This "reuse" design allows an abandoned thread to re-secure its original position in the queue where it had previously abandoned. Such thread can resume its wait notwithstanding its prior abandonment if the lock was not already passed beyond its position. A consequence of this design is that a thread can opportunistically express its "locking intent" well before the lock may be actually needed, *possibly* reducing the waiting time when the need for the lock becomes imminent.

The HMCS-T lock ensures starvation freedom for threads that can wait and ensures a bounded number of steps to abandon (on timeout) or release the lock. Having all these traits is uncommon in lock designs that admit a timeout [8, 19, 20]. For threads that do not admit a timeout, HMCS-T retains the same starvation freedom and fairness guarantees as the HMCS lock, even when intermixed with threads that admit timeouts. We have proved the correctness of HMCS-T via formal verification using the Spin [12] model checker.

The contributions of this paper are the following:

1. Design of a high-throughput queue-based hierarchical spin lock with a timeout capability, which exploits locality and ensures low overheads for abort and reentry.
2. Formal verification of correctness and progress guarantees of the intricate algorithm.
3. Demonstration of the superiority of HMCS-T over other abortable locks and demonstration of 34% speedup and 4.3× superior scaling in an MPI+OpenMP Graph500 BFS kernel by using the HMCS-T lock.

2. Background and Related Work

MCS [17] and CLH [15] queue-based locks do not allow abandoning once a thread has enqueued itself. Scott and Scherer [20] devised MCS and CLH locks with a timeout, which were not non-blocking. Scott designed the non-blocking variants (*MCS_NB* and *CLH_NB* locks [19]), which, unfortunately, introduced an unbounded worst-case space and time complexity for a given number of threads. The designs in [19] need explicit memory management before each acquisition with potential remote memory accesses, which becomes a bottleneck as we show later in our experiments (cf. § 6.2). In contrast, HMCS-T ensures starvation freedom—the threads that do not timeout eventually acquire the lock, and the threads that timeout terminate their protocol in a bounded number of steps. HMCS-T has no loop that cannot admit a timeout; it has a bounded space complexity, and it does not require continuous memory management. The improvements come with an intricate design and slightly reduced performance compared to the baseline HMCS.

Jayanti [14] proposed a token-based abortable lock with linear space bounds and logarithmic time bounds. Pareek et al. [18] proposed a randomized algorithm to abortable locks that can achieve sub-logarithmic remote memory references. Marathe et al. [16] combined a fixed-length queue lock with a back-off lock, relieving the design from expensive memory management. None of these locks are NUMA aware.

Cohort locks [8] employ two levels of locks, treating a system as a two-level NUMA hierarchy. The authors present two timeout-capable cohort locks—a 2-level back-off lock and a global backoff lock coupled with a local CLH lock (A_C_BO_CLH). Aborting a 2-level backoff lock is straightforward since there is no per-thread state. To abort an A_C_BO_CLH lock, the authors use Scott's CLH_NB lock [19] at the local level, which also suffers from repeated memory management but isolates the problem to a socket. These two abortable locks neither ensure starvation freedom (for threads do not abort) nor bound the steps after timeout (for threads that abort), unlike HMCS-T.

***HMCS lock** [5].* Since we are concerned with the timeout capability in the HMCS lock [5], we provide the details of HMCS first. HMCS employs a tree of MCS locks to leverage multiple levels of locality in a NUMA hierarchy (Figure 1). The tree structure mirrors the underlying NUMA hierarchy; each locality domain has an MCS lock of its own, which is contested by its subdomains. For example, all SMT threads on a core may compete for an MCS lock designated for that CPU core. Not all NUMA levels need be mirrored in the tree.

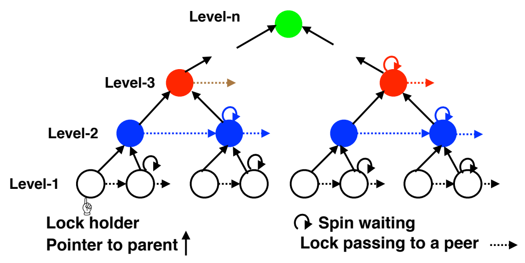


Figure 1: HMCS hierarchical tree. Threads arrive with their context nodes (empty circles) and compete for their designated level-1 MCS locks. Higher-level context nodes (solid circles) are pre-allocated in the nearest NUMA domains.

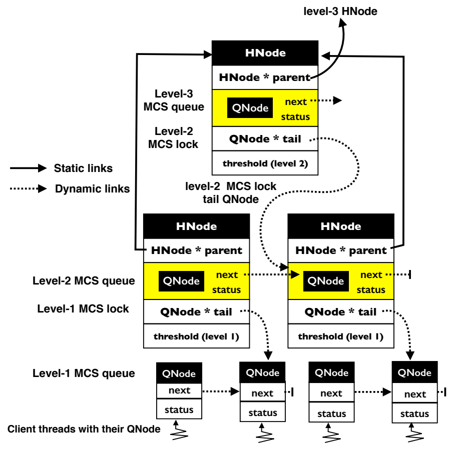


Figure 2: HMCS data structures.

Lock acquisition for each thread begins with competing for a designated leaf-level lock that corresponds to the thread's innermost NUMA domain. When a thread acquires an MCS lock in the tree in an uncontended manner (no predecessor), it proceeds to contend for its parent lock (if any) in the tree. A thread that acquires the lock at the root of the tree immediately enters the critical section. Any thread that fails in immediately acquiring a lock in the tree waits and receives that lock from its predecessor. The release protocol also starts at a leaf of the tree implicitly passing all locks held by the owner to its local successor, if any. A waiting thread at the deepest node on a thread's path to the root, typically, inherits all locks held by its predecessor. As a result, when a successor is granted a lock, it immediately enters the critical section *without explicitly contending for other locks along its path to the root.* To prevent starvation, threads may pass an HMCS lock within a locality domain for a bounded number of times. HMCS does not admit a timeout.

Each interior node of the HMCS tree (`HNode`) encapsulates an MCS lock, a queue node (`QNode`), and a pointer to its parent `HNode` (Figure 2). Each `HNode` is pre-allocated in the corresponding NUMA domain. Threads arrive with their own `QNode`s and a reference to their leaf-level `HNode`. When competing for a non-leaf-level lock, a thread uses a `QNode` present inside the `HNode` dedicated for that domain.

Terminology. We consider a leaf lock of HMCS to be at *level* 1 and the root lock to be at *level* n. We represent an arbitrary level with the letter l. The *tail pointer* of an MCS lock refers to the lock word [5, 17]. The tail pointer is `null` when the lock is free and *non-*`null` otherwise. Although a level-l MCS lock's tail pointer is physically placed a level-$l+1$ `HNode` data structure, for the ease of prose, level-l lock, level-l queue, and level-l `QNode` refer to the lock at level-l, the MCS queue used for competing for the level-l lock, and the constituent `QNode`s in that queue, respectively. For example, a level-1 `QNode` is inside a level-1 queue, which is used to compete for a level-1 lock. An HMCS lock that has n lock levels is represented as HMCS$\langle n \rangle$ and the analogous HMCS-T lock is represented as HMCS-T$\langle n \rangle$. MCS≈HMCS$\langle 1 \rangle$.

Two or more `QNode`s (and hence their owner threads) are *peers* at level l, if their lowest common ancestral lock is at level l. Two or more `QNode`s *belong* to the same domain at level l, if they share a common lock at level $\leq l$. In a sequence of locks $(l)(l+1)\cdots(m)$, where $1 \leq l < m \leq n$, from level-1 to the root, a lock *prefix* refers to the sequence $(l)(l+1)\cdots(k)$, where $l \leq k \leq m$, and a lock *suffix* refers to the sequence $(k+1)\cdots(m)$. If a lock holder hands off the lock to a waiting thread, we refer to it as "lock passing". A thread "relinquishes" the lock if no thread is waiting during the lock-passing attempt. A "release" refers to either lock passing or relinquishing. `SWAP` refers to an atomic exchange operation. `CAS` refers to an atomic compare exchange operation. On timeout a thread "aborts" or "abandons".

3. Design of One-level HMCS-T

We first describe the HMCS-T$\langle 1 \rangle$ protocol, which allows abandoning in an MCS lock (i.e., HMCS$\langle 1 \rangle$). In §4, we generalize the design to an n-level HMCS-T lock.

Overview. The key idea of HMCS-T is that a thread can abandon by updating its already enqueued `QNode` by `SWAP`ing a special flag, leaving the `QNode` behind in the MCS queue. On noticing an abandoned successor, a lock-releasing predecessor takes the additional responsibility of passing the lock to an unabandoned successor in the queue. The protocol provisions readmission of a previously abandoned requestor into its earlier secured position, if possible. Figure 3 depicts some important steps in the protocol.

Details. The HMCS-T$\langle 1 \rangle$ encodes special values—waiting (`W`), unlocked (`U`), abandoned (`A`), and recycled (`R`)—in the status field of a `QNode`. Every `QNode` is initialized with `R` value for its status field and `null` for its `next` field on creation. *Typically*, a thread waiting to acquire the lock has a `W` status in its `QNode`; a lock holder passes the lock to its successor by `SWAP`ing an `U` into its successor's status field. Symmetrically, a waiting thread on observing its `QNode`'s status change from `W` to `U` infers the lock ownership; a thread updates its `QNode` status to `A` when it aborts on a timeout. A lock releaser that observes an `A` in its successor's status while passing the lock infers successor's abandonment and performs the additional work of releasing the lock to an unabandoned successor, if any.

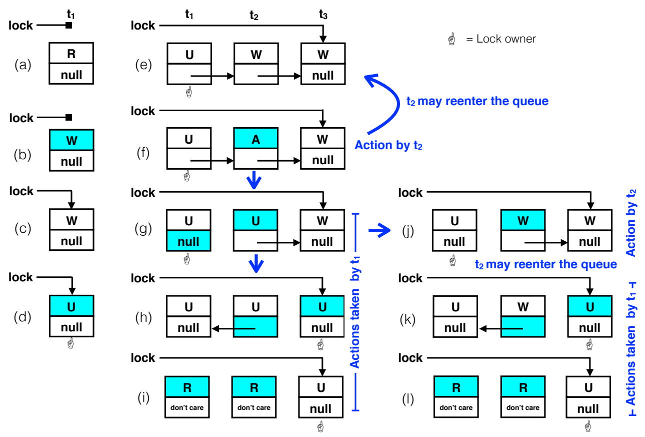


Figure 3: (a) The lock is free, and the thread t_1 arrives with a virgin `QNode`. (b) t_1 swaps its status to `W`. (c) t_1 swaps the tail pointer to itself. (d) Since t_1 has no predecessor, it updates its status flag to unlocked (`U`) and becomes the lock owner. (e) Threads t_2 and t_3 follow a similar protocol and get enqueued. (f) t_2 times out and `CAS`es `A` in its status flag. (g) t_1 begins releasing the lock by swapping t_2's status and notices that t_2 has abandoned. (h) t_1 proceeds to t_2's successor and passes the lock to t_3. Also, t_1 updates the next pointers to point to their predecessors in steps (g) and (h). (i) t_1 updates the status of t_2 and t_1 to `R` indicating their readiness for next round of acquisition. After (f), t_2 may re-enter the queue and transition the system back to (e). Also, t_2 may re-enter after (g) by swapping its status to `W` in (j) but it cannot acquire the lock or reuse the `QNode` until t_1 has passed the lock to t_3 as shown in (k) and updated t_2's status to `R` as shown in (l).

QNode maintenance. In HMCS-T, the `next` and `status` fields of a `QNode` will be accessed by a predecessor even after the owner thread has aborted its acquisition. Hence, an owner thread may not deallocate its `QNode` until the node is marked for recycling. We use the C++ destructor to ease the `QNode` maintenance. For convenience, each participating thread allocates an object (`QNodeObject`) that encapsulates a `QNode` along with a reference to the parent lock. The object's destructor waits until the object is marked for recycling. The object's constructor initializes the `next` and `status` fields and sets the parent pointer. The acquire and release protocols use a reference to this per-thread object. A thread can reuse the same object across multiple episodes of lock requests without having to reinitialize; for high performance, the reuse is necessary. Listing 1 shows an example usage model, which is applicable to any n-level HMCS-T. The object can reside anywhere—stack, heap, or thread-local storage. Heap allocation allows returning from a callee stack frame without waiting (in the destructor) for the object to be recycled. When used in C, the user must explicitly call the constructor and destructor routines after allocation and before deallocation, respectively.

Acquisition. A thread t begins its lock entry protocol by `SWAP`ing the value `W` to the status field of the thread's `QNode` q. The previous value of the status field distinctly indicates the state of the `QNode`, which governs the next action taken by t. Depending on whether the `QNode` is ready to use, abandoned previously, or in an intermediate state of being updated by a predecessor, the following three scenarios arise:

```
// Initialize the lock giving it the machine topology
HMCST<3> hmcstLock (machineTopology);
#pragma omp parallel
{
//Assume threads are pinned to cores.
//Create and initialize the per-thread QNode object.
 QNodeObject ctxt(hmcstLock);
 for ( ... ) { // ctxt is reused many times
   if (hmcstLock.Acquire(&ctxt, timeout)==SUCCESS) {
     CriticalSection();
     hmcstLock.Release(&ctxt, timeout);
   } else {
     NonCriticalSection();
   }
 } // end for-loop
} // The destructor ensures ctxt is ready to free.
```

Listing 1: An example 3-level HMCS-T lock in an OpenMP C++ code.

Recycled node (`R`): The `QNode` `q` is ready for use. In this case, t, by-and-large, follows the MCS lock's enqueue protocol: swaps the tail pointer to point to `q` and if t has no predecessor, it becomes the lock owner, overwriting the status flag to `U` (not done in the original MCS), otherwise, t spins either until the lock is granted (some predecessor changes `q.status` to `U`) or a timeout occurs. On timeout, t `CAS`es the value `A` to `q.status` and abandons the protocol if `CAS` succeeds. `CAS` failure implies lock acquisition.

Previously abandoned node (`A`): It implies t is attempting to re-acquire a lock that it had previously abandoned. *In this case, the lock has not yet been passed beyond t by any of its predecessors.* Hence, t resumes its wait by directly jumping to the wait loop in the MCS acquisition protocol. Note that swapping the tail pointer and updating the predecessor's `next` field are elided. While waiting for the lock, if t times out once again, it `CAS`es an `A` into its `q.status` and exits if `CAS` succeeds. `CAS` failure implies lock acquisition.

Node unlocked after abandonment (`U`): It implies t must have abandoned previously, and some predecessor p must have tried to grant the lock to t by `SWAP`ing a `U` value. Since t had already abandoned the node, p is now in the process of passing the lock to a waiting successor in-line past t. Once, p has passed or relinquished the lock, it will change the status of t's `QNode` `q` to `R`. In this case, t waits for `q.status` to change to `R`. After `q.status` becomes `R`, t goes through the full enqueue process following the steps listed for a "recycled node". While waiting for the status to change to `R`, however, if t times out, it `CAS`es an `U` to revert `q.status` and abandons the wait on `CAS` success. Failure to `CAS` implies `q` is recycled, and t may choose one of the following options: **(1) [optimistic]** follow the steps listed in the "recycled node" case and attempt to acquire the lock hoping to get it immediately without waiting or **(2) [pessimistic]** return as if the lock waiting timed out. We have implemented the optimistic strategy.

The status flag shall never be `W` when `SWAP`ing at the entry.

Release. A lock releasing thread performs actions that allow abandoned `QNode`s in the queue to become reusable by their owner threads. When a thread, r, is releasing the lock by updating the status field of its successor s with a value `U`, the successor could be simultaneously updating its status field with an `A` in an effort to abandon. *HMCS-T's release* *protocol is modified to use a SWAP on successor's status field unlike an unconditional store used in the original MCS lock.* The releaser r may have one of the following cases based on the previous value observed in the status field of the successor:

Waiting successor (`W`): It implies r successfully granted the lock to s, which was still waiting.

Abandoned successor (`A`): It implies s abandoned the lock before r performed its release. In this case, r "impersonates" as if s were releasing the lock to its successor and r repeats one of these two steps, if s has a successor.

In case the releaser r finds no waiting successor in the MCS queue, it `CAS`es the tail pointer to `null`, similar to the MCS lock. In the "abandoned successor" case above, once a releaser takes the role of its successor, it unconditionally overwrites its successor's `next` field to its predecessor, which aids in the reverse traversal. Thus, the `next` pointer is reused as a predecessor pointer. It is legal to trash the next pointer and reuse for another purpose since no one will be using it beyond this point. Once r has successfully passed or relinquished the lock, it follows the chain of predecessor pointers (now stored in the `next` fields of abandoned `QNode`s), unconditionally setting the status in each abandoned `QNode` to `R`. This step essentially marks abandoned (and potentially re-entered) `QNode`s to be ready for reuse. Any attempt to reuse an `U` marked `QNode`s by their owning threads for an acquisition during this intermediate state will result in the acquire protocol to land into the "node unlocked after abandonment (`U`)" case.

The order of recycling the abandoned `QNode`s is LIFO. One can remember the first node and the last node and follow FIFO ordering; we chose LIFO with the intention of allowing newcomers a better opportunity since they will have more time left to reenqueue and likely succeed. The releaser r, does not eagerly flip the status of abandoned `QNode`s to `R` during its left-to-right forward journey because doing so allows abandoned threads to reenqueue and abandon repeatedly making r's time complexity unbounded. Marking the `QNode`s after the lock release bounds the release protocol's time complexity to linear in the number of participants.

Non-blocking release. The MCS lock has an unbounded wait in its release protocol. In MCS, if a lock releaser, r, notices no successor, it `CAS`es the tail pointer to `null`. However, in the meantime, a new thread, s, might have swapped the tail pointer, and s may be in the process of enqueueing. On `CAS` failure, while the original MCS protocol waits indefinitely until the successor advertises itself by updating the `next` field of its predecessor, *HMCS-T deviates from this behavior and makes the release protocol non-blocking.* To accomplish the non-blocking release, HMCS-T obeys the following protocol (see Figure 4 for an example):

Release: On failure to `CAS` the tail pointer to `null`, the releaser r attempts to `CAS` a special value `M` (`iMpatient`) on to the next field of its `QNode` q_r. Note that q_r may not be owned by r; if successors of r had abandoned, r would be impersonating the last one in the chain of abandoners.

If the `CAS` succeeds, r's forward journey is over. However, r does not toggle q_r`.status` to `R` on its return journey through the predecessors because it is unsafe to recycle q_r until s has published itself by updating q_r`.next`. If the `CAS` fails, because s published itself by updating q_r`.next` in the meantime, then r follows the next pointer to s and `SWAP`s q_s`.status` to `U`. The procedure may recur if s aborts before r updates q_s`.status`.

Acquire: To perform a symmetric handshake with the release protocol, the acquire protocol of a successor s performs a `SWAP` on the next field of its predecessor. If s sees the special value `M` in its predecessor's `next` field (i.e., q_r`.next`), it infers that the predecessor abandoned having become impatient waiting for its successor. Consequently, s updates its predecessor `QNode`'s status to `R` indicating that the predecessor node (i.e., q_r) can now be recycled. Also, since s had no "legal" predecessor, it implies lock acquisition. s updates its status to `U` and enters the critical section.

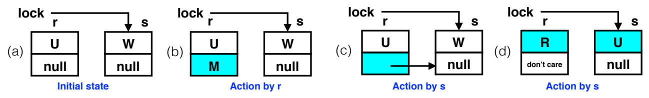


Figure 4: (a) Thread r owning the `QNode` q_r is releasing the lock when another thread s arrives, but s is slow in updating q_r's next pointer to itself. (b) r times out and `CAS`s a special value `M` in q_r`.next` and leaves. (c) Later s `SWAP`s a pointer to itself into q_r`.next`. (d) Having observed the special value `M`, s updates q_r`.status` to `R` indicating that q_r can now be reused. Also, s updates its status to unlocked (`U`) and becomes the lock owner.

4. Design of N-level HMCS-T

In principle, if a thread wants to abort in HMCS-T$\langle n \rangle$, it updates the status flag of the `QNode` it is waiting on to `A` and releases all of its lower-level locks. Symmetrically, if a lock releaser encounters an abandoned successor, it continues looking for a waiting thread until it has found one at the current level; otherwise, it continues to search at the parent level. The details are more involved. Figure 5 presents the key steps of the HMCS-T protocol; we provide the detailed description below.

Details. If a thread (say α) times out while waiting for a level-l lock, it updates the status of its `QNode` (the one it used for competing for the level-l lock) to `A`. A level-l predecessor of α will ensure passing the level-l (and above) locks to a level-l waiting successor past α, similar to HMCS-T$\langle 1 \rangle$.

If α times out while competing for a level $l > 1$ lock, it must have already acquired the locks at levels $[1, l-1]$. By now, there could be other threads waiting at these lower levels (possibly some already abandoned) relying on α to pass the global lock, which α failed to acquire. Hence, after abandoning at level l, α should release locks at levels $[1, l-1]$ (see Figure 5(a)). Naively, α could release all locks

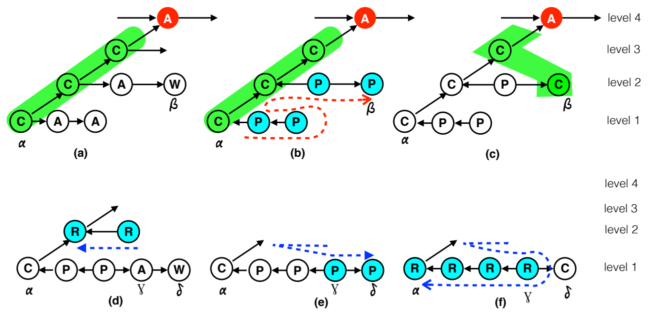


Figure 5: (a) The thread α abandons at level 4 having acquired the level 1-3 locks. Every `QNode` in the left frontier has the `C` status. There are several abandoned `QNodes` in the same domain as α under level 4. (b) α attempts to pass the level 1-3 locks to a successor at level 1 but finds none. α ascends to level 2 and passes the level 2 and 3 locks to the waiting successor β. α marks the status of all `QNodes` along the path to `P`, temporarily disallowing their reuse. (c) β updates its status from `P` to `C`. β recognizes (not shown) that it owns the level 3 lock by observing the `QNode` status of `C` on ascending to level 3. (d) concurrently, α traverses the predecessors in level 2 and marks them ready for reuse. By the time α descends to level 1, γ and δ enqueue at level 1 with γ already abandoned. (e) α passes the level-1 lock to the first waiting successor δ, (f) α traverses the predecessors in level 1 and marks them ready for reuse. Concurrently, δ updates its status to `C` and proceeds to compete at level 2.

from level $l - 1$ to 1, in that order, to a first waiting successor, forcing all of its successors at each level to compete at their parent levels. This naive design, however, suffers from the following problems: 1) releasing all $l - 1$ locks is extra work for α compared to simply passing a prefix of locks (suffix from α's viewpoint) to the lowest common descendent, 2) top-down traversal makes high-latency releases compulsory, and 3) every successor at each level has the additional work of competing at the next level.

Our solution is a disciplined, two-pass lock-passing arrangement that exploits locality and minimizes work. In a forward pass, starting from leaf to level $l - 1$, α finds the lowest common waiting descendent β and passes all locks it has held starting from that level till level $l - 1$ to β. For example, in Figure 5, α abandons at level 4 and passes the already acquired level 2 and 3 locks to β, which was waiting at level 2. β, now, skips acquisitions at levels that it inherited from α by noticing the special value (`C`) left by α in the status fields of the `QNodes` at levels that are already acquired. If α found β at a level $i < l - 1$, α's forward journey stops at level i. In a reverse path from the level $i - 1$ down to level 1, α releases the locks it had not passed during its upward journey. During the forward pass, α would have marked every abandoned successor along the way as temporarily not reusable; it marks them as reusable during its reverse pass.

Deviation in internals from HMCS. HMCS protocol uses its status field to encode the *passing count*—the number of times the lock has been circulated locally [5]. The passing count is incremented on each successive passing to a peer, and the global lock is passed to a successor s by signaling it with a symbolic value `V` $\in [2, \theta]$, where θ is the threshold of local passing. A releaser signals its successor s with a special value `ACQUIRE_PARENT` (`P`), either when the threshold (θ) of local passing is reached or when s arrives after the releaser already released the global lock at an enclosing level. A waiting thread s that is signaled with the special value (`P`), immediately resets its `status` to `COHORT_BEGIN` (`C`, numerically equal to 1), which indicates: (1) s's ownership of the local lock alone and hence the need to compete at the parent level, and (2) the start of a new round of local passing.

In HMCS-T, the meaning of `P` is generalized to signal a successor s that it now inherits a *prefix of locks* (instead of the local lock alone) starting at the current level on s's path to the root. Symmetrically, the status `C` is generalized to indicate that the thread owns a *prefix* of locks starting at the current level but *not all locks on the path to the root.* The value `P` can be used either when a thread aborted at an ancestor level (and hence passing a lock prefix) or when it reached the local passing threshold. In either case, the successor does not inherit the global lock; the successor turns its status to `C` and proceeds to compete at ancestral levels, similar to HMCS.

Acquisition. Acquisition begins with competing for a designated level-1 lock. The acquisition protocol of any level-l lock starts by `SWAP`ing the value `W` to the status field of the `QNode q`; remember that the leaf-level `QNodes` are provided by the client threads and the interior `QNodes` at each level are pre-allocated within `HNodes`. The previous value of the status field of the `QNode` governs the subsequent actions taken by the acquire protocol for a thread t. HMCS-T's non-root-level lock has more status flags to handle. The following four cases arise based on the previous value of the status flag:

Recycled node (`R`): This implies `q` is reusable. HMCS-T enqueues `q` by `SWAP`ing the tail pointer of the corresponding level-l lock. Subsequently, one of the following ensues:

1. *Uncontended acquisition (tail was `null`):* t updates the `q.status` to `C` (the beginning of a new cohort) and proceeds to compete for the parent level lock,
2. *Inheritance of all locks (status becomes a legal passing value, `V`):* acquisition ends successfully,
3. *Acquisition of a prefix of locks (status becomes `P`):* t updates `q.status` to `C` and proceeds to the next level,
4. *Timeout:* t `CAS`es an `A` into `q.status` and returns failure if `CAS` succeeds. `CAS` failure means successful acquisition.

Previously abandoned node (`A`): This implies the `QNode q` was previously abandoned either by t itself or a thread belonging to the same domain. HMCS-T, in this case, skips enqueuing `q` and resumes the spin wait on `q.status`.

Inherited ancestral lock (`C`): This happens when the current level (l) lock was already acquired by a thread p belonging to t's domain at level l or below. Since `q.status` was `C`, p must have started a new round of local passing at level l, which implies p did not inherit all locks on its path to the root, and instead it must have proceeded to compete at higher levels (see "Recycled node" case 1, 3). Also, p must have abandoned at some level $j > l$ and subsequently, p found t as its lowest-common descendent at a level $k < l$

and passed a prefix of locks to it. Hence, t already owns the lock at level l. Since t swapped a `W` during its entry, it reverts `q.status` to `C` and proceeds to level $l+1$. Reverting is necessary to allow a subsequent thread to acquire this level lock if t also were to abort at a higher level.

Node unlocked after abandonment (`V` and `P`): `V` $\in [2, \theta]$ symbolizes that a predecessor already stepped upon an abandoned node. This case is analogous to the `U` case of HMCS-T$\langle 1 \rangle$: either t itself or some other thread belonging to the same domain as t at level l must have previously abandoned at level l; t's predecessor p (a level l peer) must have tried to pass the lock to t; p, having noticed that t had abandoned, is now in the process of passing the lock to a waiting successor past t. p will, eventually, revert `q.status` to `R` once it releases the lock. Hence, t waits for `q.status` to change to `R` and then re-enqueues `q` afresh into the MCS queue at level l obeying the protocol already listed under "recycled node". While waiting for `q.status` to become `R`, if t times out, it `CAS`es the value `P` into `q.status` and aborts if `CAS` succeeds. `CAS` failure happens iff `q.status` becomes `R` allowing t to reenqueue `q` optimistically hoping to acquire the lock without waiting.

The status flag shall never be `W` when `SWAP`ing at the entry. Also, if a thread ends up acquiring the lock at a level when it is about to abandon but does not inherit all locks, it continues to attempt for the next level lock, optimistically hoping to acquire the next level lock in an uncontended manner. If, however, it fails to acquire the next level lock, it begins the abandonment process at that level.

Abandonment. To abandon, while waiting for a lock at level l, the abandoner α obeys the following protocol:

1. α, first, `SWAP`s the value `A` into the status flag of its `QNode` used for competing at level l.
2. If at non-leaf level (that is $l > 1$), α attempts to pass its already acquired locks (levels $[1, l-1]$) to the lowest common waiting descendent thread. That is, to retain maximum locality, α starts looking for a waiting successor starting at the leaf level (`curLevel=1`). Lock passing is similar to HMCS-T$\langle 1 \rangle$, looking for the first unabandoned successor. α `SWAP`s the special value `P` into its successor s's status field, which signals s (if waiting) to acquire the necessary suffix locks on its path to the root starting at `curLevel+1`.
3. If α succeeds in passing the lock prefix to a waiting peer at `curLevel`, it goes to Step #4, otherwise (the `next` pointer of the right-most peer is `null`), α ascends to the next level (if any) and recursively performs Step #3 with `curLevel=curLevel+1`.
4. By now, α either passed its lock to a waiting successor or relinquished the lock by `CAS`ing the tail pointer to `null`. α traverses the `QNode`s at the `curLevel` from right to left flipping the flags of each `QNode` to `R`, indicating that those `QNode`s can now be recycled.
5. If `curLevel>1`, α descends to its children level (`curLevel=curLevel-1`). Having already passed its acquired locks at levels $>$ `curLevel`, α now needs to release the lock at `curLevel`. Let `m` be the `QNode` that had its `next` pointer `null` when α ascended to the next level in Step #3. It is possible for more threads to have enqueued past `m` when α was busy passing the upper-level locks. Starting at `m.next`, if any, α attempts to pass the `curLevel` lock to a waiting `curLevel` peer. Since passing the lock at `curLevel` in this phase does not correspond to passing all upper-level locks (which were already released by now), α `SWAP`s the special status flag `P`, indicating the successor to compete at the next level. If no peer is found, α relinquishes the `curLevel` lock by `CAS`ing the tail pointer to `null`. Having released the `curLevel` lock, α performs a right-to-left traversal of all `QNode`s whose flags it had `SWAP`ed to `P` and flips them to `R`, indicating that they can now be recycled. Figure 5(d)-(f) captures this case. α now repeats Step #5.

At any point, if a lock releasing thread times out while waiting for a successor to update its next pointer, it uses the aforementioned non-blocking technique. Also, at each level, once a releasing thread steps past its abandoned successor node, it uses the next pointer to serve as a predecessor pointer for its return journey.

Release. Lock release is analogous to the steps followed in the abandonment process, except,

1. The ascendance continues till the tree root if no waiting successor is found, and
2. If the value `V` of the releaser r's representative `QNode`'s status flag is less than the local passing threshold at level l, then r signals its successor with `V+1`, similar to HMCS. If the passing threshold has reached, then the r releases the parent lock followed by signaling the successor with `P` to indicate that the successor should compete at the next level.

5. Discussion

Fast-path optimization. To avoid the overhead of multiple levels of lock acquisitions under no contention, we have adapted the *fast-path* strategy described in [6] into HMCS-T. We omit the details of the adaptation for brevity.

Atomic operations. Our use of `SWAP` and `CAS` operations may look heavy handed, but they are applied on nearer neighbors, which do not generate remote coherence traffic. Recent studies [7] show that the atomic operations introduce no more than $1.15\times$ overhead compared to loads and stores. Not all atomic operations are on the critical path. Uncontended acquisition in HMCS-T$\langle 1 \rangle$ is same as the uncontended acquisition in MCS with an additional `SWAP` on the status field. Uncontended release in HMCS-T$\langle 1 \rangle$ is same as the uncontended release in MCS. Uncontended acquisitions in deeper HMCS-T locks cost no more than HMCS-T$\langle 1 \rangle$ due to the fast path optimization.

Memory overhead and lock abuse. HMCS-T, like other hierarchical locks, trades memory for speed. A prior work [19] on abortable locks traded speed for memory. Our evaluation shows that the node recycling is so time consuming on NUMA machines that the locks described in [19] offer little competition to HMCS-T. Preallocated `QNodes` achieve a lower memory footprint compared to [19] under high abort rates. However, one must use precaution when using a deep HMCS-T. HMCS-T must be curated for a given architecture. HMCS-T is not a universal replacement for all locks in a program. One should opt into HMCS-T when contention is expected. A contended HMCS-T's memory consumption is no worse than that of a contended MCS lock since in the worst case HMCS-T would require $2\times$ more space; much less in practice. Modern many-core processors with 10s of MBs of cache can accommodate thousands of HMCS-T locks in less than 1% of their last-level cache.

Design options. On finding an abandoned successor, instead of traversing subsequent successors, one may choose to ascend to ancestral levels to increase the chance of finding a successor and reduce the critical path length. However, the cost of remote access is often much higher than several local accesses. The naive design of releasing ancestral locks instead of releasing the peers first causes up to $3.2\times$ performance loss compared to our optimized strategy. It is straightforward to enforce a bound on how many peers to inspect before ascending to the parent level based on an analytical machine model similar to the one discussed in [5].

Hierarchical locks are best served when threads are pinned to cores. Thread migration makes HMCS-T start its new acquisition in its new domain iff the leaf-level `QNode` is already recycled.

IsLocked() Interface. One can expose an `IsLocked()` API over HMCS, which returns the boolean `LOCKED` or `NOT_LOCKED` based on whether or not any thread already holds the lock. A "try lock" can be built using such interface. If `IsLocked()` returns `NOT_LOCKED`, the user still has to decide whether to acquire the lock with a timeout (HMCS-T) or wait until the lock is granted (HMCS). In summary, the timeout capability is indispensable.

Correctness. HMCS-T is a very involved protocol; the system is stateful. HMCS-T guarantees mutual exclusion, livelock and deadlock freedom under any situation, and starvation freedom for non-aborting threads. We follow a multistep approach to prove these guarantees. We provide a detailed proof in a separate document: `http://github.com/HMCST/hmcst`, which relies on the Spin [12] model checker.

Complexity of HMCS-T. Let us assume the system has n lock levels and each lock at a level i is contested by k_i descendants. Hence, the total number of participants (maximum threads) in the system is $N = \prod_{i=1}^{n} k_i$.

Space complexity: The space complexity depends on the pre-allocated `HNodes` in the tree. A deeper tree has a larger space complexity. For a given number of participants, the tree depth depends on the machine topology. A loose bound assumes a binary tree. Hence, for a system with N participating threads, there can be at most N pre-allocated `HNodes` and the protocol does not allocate any more `HNodes`. Hence, the space complexity per lock is $O(N)$.

Time complexity: Discussion about time complexity is useful only in light of timeout. Once a timeout occurs, we would like to know how many steps are needed to either abandon or release the lock. Due to its optimistic design, a thread that times out waiting for a leaf-level lock may acquire all locks in n steps followed by releasing all n locks. Releasing n locks, in the worst case, can take k_i steps in each of i^{th} level from 1 to n in a bottom-to-top sweep and k_i more steps in a top-to-bottom sweep. Hence, the worst case acquire followed by a release after timeout is bounded by $n + \sum_{i=1}^{n} 2k_i$ steps. The worst case abandonment happens when a thread that times out waiting for a leaf-level lock acquires $n-1$ locks but fails to acquire the root-level lock and subsequently releases all $n-1$ locks. Hence, the worst case abandonment is bounded by $n - 1 + \sum_{i=1}^{n-1} 2k_i$ steps.

The asymptotic cost on the critical path when all threads, except the lock holder, have aborted is $\sum_{i=1}^{n} k_i$. The bound can be tightened to $(n-1) + k_n$ via the previously proposed alternative of ascending to the parent level on finding an aborted successor.

6. Evaluation of HMCS-T

In this section, we evaluate HMCS-T by comparing it with various locks. We use an 8-blade, 16-socket HP Integrity Superdome X [10] shared-memory machine. Each socket is an 18-core, 2-way SMT Intel Xeon E7-8890V3 processor clocked at 2.5GHz. Blades are interconnected with a custom ASIC. The system has a total of 576 hardware threads. L1, L2, and L3 cache sizes are respectively 32KB, 256KB, and 45MB. A pair of sockets on the same blade are connected over QPI [13]. The system has two on-chip, 2-channel DDR4 memory for a total of 12TB. We used `g++ v4.8.3` compiler and parallelized code via `OpenMP`. Unless stated otherwise, all experiments use a compact binding of threads to cores; a thread will not be bound to another NUMA domain until the current domain is fully populated; the next chosen NUMA domain will be the nearest possible one.

Locks: We use the HMCS-T locks of 1, 2, 3, and 4 levels of hierarchy in our studies. A shallower lock, HMCS-T$\langle 1 \rangle$ does not respect locality. An HMCS-T$\langle 2 \rangle$ exploits the locality within a node shared by two sockets but ignores the locality within a socket. An HMCS-T$\langle 3 \rangle$ exploits the locality within a socket shared by 18 cores, in addition to the behavior of HMCS-T$\langle 2 \rangle$ but ignores the locality of SMTs sharing a core. An HMCS-T$\langle 4 \rangle$ exploits the locality within a core shared by two SMTs, in addition to the behavior of HMCS-T$\langle 3 \rangle$. An HMCS$\langle n \rangle$ has a symmetric configuration as an HMCS-T$\langle n \rangle$ but lacks the timeout capability.

We also compare with a Test-And-Test-And-Set lock with timeout (TATAS-T), CLH_NB and MCS_NB locks [19, 21], and A_C_BO_CLH lock [8]. Our A_C_BO_CLH lock mimics the implementation described in [8]; we form a 2-level hierarchy where cores sharing the same socket form a cohort. For the locks with a cohort property (HMCS, HMCS-T, and A_C_BO_CLH), we set the passing threshold to 64 [8].

§6.1 compares HMCS-T with HMCS using a micro benchmark. §6.2 compares HMCS-T with other abortable locks via a splay tree case study. §6.4 demonstrates the virtue of HMCS-T by exploiting parallel slackness in an MPI+OpenMP Graph500 code and improves its communication-computation overlap.

6.1 HMCS vs. HMCS-T

An HMCS$\langle n \rangle$ serves as an upper (lower) bound for the throughput (latency) of HMCS-T$\langle n \rangle$ when the timeout is infinite. To assess the overhead that HMCS-T adds atop HMCS, we compare them via a micro benchmark: a tight loop of lock acquire and conditional release. We precisely measure the CPU cycles for acquire and release via the x86 `rdtsc` instruction. The benchmark while synthetic is appropriate for the intended comparison.

Figure 6 and 7, respectively, show the throughput and round trip latency (successful acquisition plus release) when the timeout is set to 1K, 10K, and infinite CPU cycles. It is evident from Figure 6 that every HMCS-T instantiation closely follows the throughput of its corresponding HMCS counterpart. The gap narrows as the timeout increases. Figure 7 shows that the latency for successfully acquiring a lock is strictly bounded by the chosen timeout. If lock acquisition is not possible within a given timeout, thread aborts in HMCS-T, and thus the latency of successful acquisitions reaches a plateau, unlike HMCS. At infinite timeout, HMCS-T$\langle n \rangle$'s latency is very close to that of its HMCS$\langle n \rangle$ counterpart. In-general, the throughput of HMCS$\langle 4 \rangle$ > HMCS-T$\langle 4 \rangle$ > HMCS$\langle 3 \rangle$ > HMCS-T$\langle 3 \rangle$ > HMCS$\langle 2 \rangle$ > HMCS-T$\langle 2 \rangle$ > HMCS$\langle 1 \rangle$ > HMCS-T$\langle 1 \rangle$. Not all latency appears on the critical path. The wait time during an acquire is not on the critical path. Once global lock is handed over to a legal successor, rest of the release protocol is not on the critical path. Hence, the throughput is unaffected after 10K CPU cycles timeout.

Table 1 shows the throughput degradation in HMCS-T$\langle n \rangle$ over the corresponding HMCS$\langle n \rangle$ at timeout=∞. The data shows the pure protocol overhead that an HMCS-T incurs to support the abort feature. Under no contention (1 thread), the overhead is fairly low (about 1.25×) in any HMCS-T lock; the degradation does not increase noticeably with the increase in the depth of the lock, which is due to the aforementioned fast path. The worst case degradation is 3.85× observed for HMCS-T$\langle 4 \rangle$ at four threads. This is because of the often missed opportunity to pass to an SMT peer and the presence of subsequent lock levels leads to a lengthy path of relinquishing parent locks preventing local passing benefits. HMCS-T$\langle 4 \rangle$ is an overkill at such low thread count setting.

Table 1: Throughput degradation in HMCS-T$\langle n \rangle$ wrt HMCS$\langle n \rangle$ at timeout=∞.

Threads:	**1**	**2**	**4**	**8**	**18**	**36**	**72**	**144**	**288**	**576**	**GeoMean**
HMCS-T<1>	1.22x	1.32x	1.12x	0.97x	0.99x	0.97x	1.04x	0.85x	1.11x	1.09x	**1.06x**
HMCS-T<2>	1.27x	1.79x	1.1x	0.99x	0.97x	0.97x	1.21x	1.19x	1.19x	1.19x	**1.17x**
HMCS-T<3>	1.26x	1.92x	1.21x	1.02x	1.00x	0.95x	0.98x	1.00x	1.00x	1.00x	**1.11x**
HMCS-T<4>	1.27x	2.47x	3.85x	1.68x	1.73x	1.75x	1.74x	1.75x	1.76x	1.79x	**1.89x**

All other locks incur no more than 1.92× overhead. The average overhead is under 1.11×, except for HMCS-T$\langle 4 \rangle$.

To further understand the overheads involved, Figure 8 and Figure 9 decompose the round-trip latency into the latency for successful acquisition and release. We make the following observations:

1. Comparing Figures 8 and 9, acquisition accounts for the bulk of the latency (up to 10^6 cycles) and outweighs the latency for release (under 10^3 cycles) by orders of magnitude between the same pairs of HMCS and HMCS-T locks .

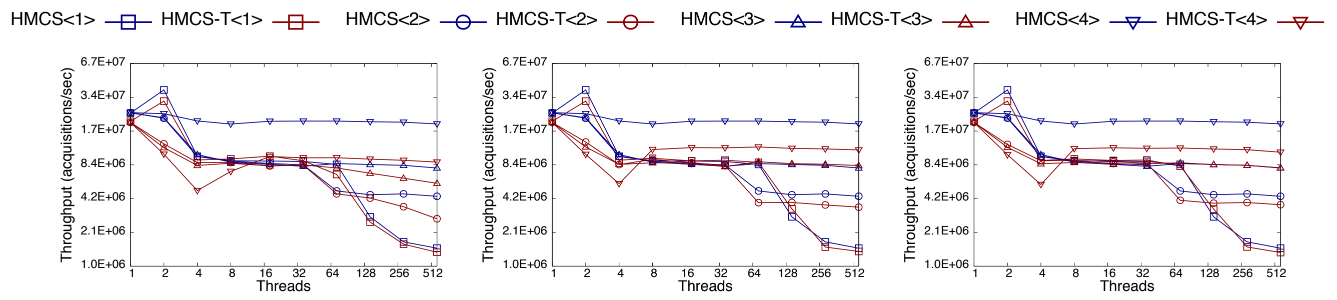
2. Figure 9 shows that the difference in release latency between an HMCS and the corresponding HMCS-T can be high. While HMCS stabilizes at about 25 cycles, HMCS-T$\langle 3 \rangle$ and HMCS-T$\langle 4 \rangle$ stabilize at about 200 cycles (8× higher). The worst case happens for HMCS-T$\langle 1 \rangle$ at 576 threads, which is up to 28× more than HMCS$\langle 1 \rangle$. Most importantly, the latency does *not* grow either with the increase in the number of threads or by changing the timeout. We note that the release latency of 200 cycles for HMCS-T$\langle 3 \rangle$ and HMCS-T$\langle 4 \rangle$ despite their longer critical paths is faster than MCS_NB and CLH_NB, which respectively take about 2300 and 1100 CPU cycles (not shown).

3. Figure 9 shows that the release latency of a shallower HMCS-T is higher than the release latency of a deeper HMCS-T despite the additional cost of releasing parent-levels in a deeper HMCS-T. This is because of two reasons: (1) it is faster to release the lock in a deeper HMCS-T due to the locality of successors, and (2) ascending to ancestral levels in the tree during lock release exponentially increases the chances of finding a waiting successor and avoids inspecting aborted threads in a peer domain.

With zero timeout (not shown), the time to abandon the protocol from the start of acquisition is under 140 cycles for HMCS-T$\langle 1 \rangle$, HMCS-T$\langle 2 \rangle$, and HMCS-T$\langle 3 \rangle$ and 1K cycles for HMCS-T$\langle 4 \rangle$. Beyond 1K cycles timeout, all HMCS-T locks accurately honor the client-chosen timeout. Overall, the HMCS-T locks maintain their competence even after adding the abort feature and offer high timing fidelity.

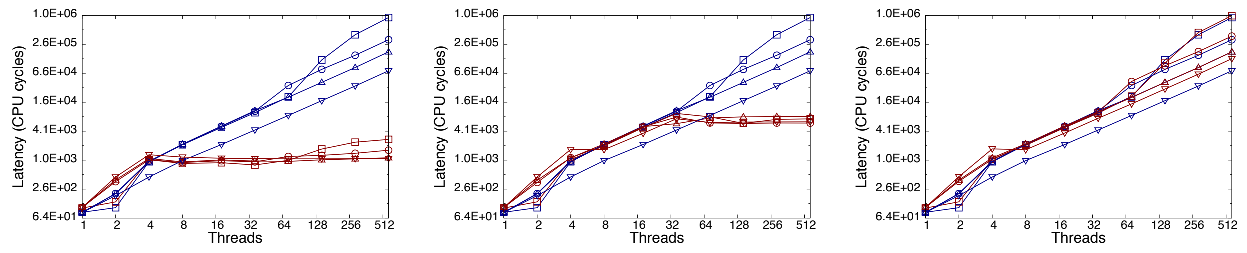
6.2 HMCS-T vs. Other Abort Locks on Splay Trees

Splay trees [23] are self-adjusting binary search trees with a caching behavior—the last searched item is brought to the tree root and recently searched items appear near the root. Any operation on a splay tree, including a lookup, may involve tree rotations; hence locks are necessary.



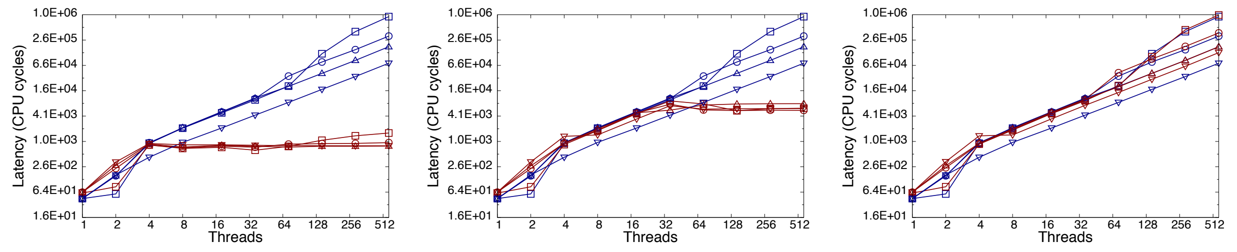
(a) Timeout:1K CPU cycles. (b) Timeout:10K CPU cycles. (c) Timeout:∞ CPU cycles.

Figure 6: Lock throughput of HMCS-T vs. HMCS. Each HMCS-T closely follows its HMCS counterpart.



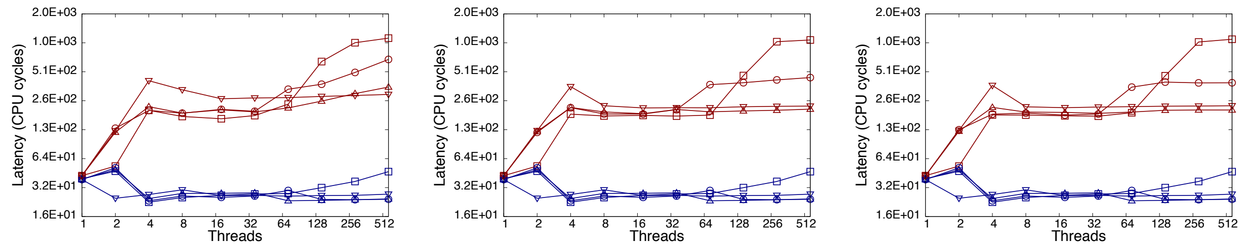
(a) Timeout:1K CPU cycles. (b) Timeout:10K CPU cycles. (c) Timeout:∞ CPU cycles.

Figure 7: Round trip latency of HMCS-T vs. HMCS. Each HMCS-T closely follows its HMCS counterpart at ∞ timeout and offers bounded latency at other timeout values.



(a) Timeout:1K CPU cycles. (b) Timeout:10K CPU cycles. (c) Timeout:∞ CPU cycles.

Figure 8: Acquisition only latency of HMCS-T vs. HMCS. Acquisition latency governs the total latency.



(a) Timeout:1K CPU cycles. (b) Timeout:10K CPU cycles. (c) Timeout:∞ CPU cycles.

Figure 9: Release only latency of HMCS-T vs. HMCS. The lock release in HMCS-T is slower than HMCS but its contribution to the overall latency is insignificant.

We create a setup where all threads attempt to perform a lookup operation on a *global* splay tree when holding a lock (*critical work*). On lock timeout, a thread performs some "other" work (*non-critical work*). The non-critical work is a *thread-local* splay tree and hence it does not require any locking. Using splay tree lookups both inside and outside the critical section allows us to compare both critical and non-critical work accomplished by various locks in a uniform way. Also, instead of idle spinning outside the critical section, switching to a different work chunk pollutes the cache, which represents real world situations.

Our global and thread-local splay trees each have 8K nodes. For both kinds of trees, 90% of the lookups are served by the same key, and the remaining 10% are for the 64 numerically adjacent keys (not necessarily nearby in the tree organization). Note, however, that not all the 90% of the global tree lookups hit at the root since a previous lookup, either by the same thread or by another thread, could have rotated the tree. Furthermore, we change the "hot node" every second to a randomly chosen node from the tree.

A superior lock should deliver high lock throughput and abort swiftly on a timeout; that is, *it shoud show high throughput on both global and local tree operations.* Figure 10(a) compares the throughput of various abortable locks at 10K CPU-cycle timeout. Throughput is measured as total tree lookup operations performed per second, which includes both local and global tree operations. Figure 10(b) and Figure 10(c) respectively decompose the throughput into global (critical) and local (non-critical) work components. We have clipped the lines when the throughput fell below 10K acquisitions per second for the global tree.

We notice that a deeper HMCS-T lock offers higher throughput than a shallower HMCS-T on the global tree. The global-tree throughput drops with increasing concurrency (Figure 10(b)) because the working set increases linearly with the number of threads, which causes the threads compete for the shared cache. For locks that can abort swiftly on timeout (HMCS-T, TATAS, and A_C_BO_CLH) the throughput of the local tree operations increases linearly as the number of threads increase beyond 18. All locks except MCS_NB and CLH_NB show comparable throughput on their local-tree operations.

CLH_NB and MCS_NB [19] locks have dramatically lower throughput compared to HMCS-T locks both on the global tree and local trees. This is because their memory management overhead is excessive. Frequent aborts severely affect node recycling. In the CLH_NB lock, for example, when a thread aborts, it leaves its `QNode` in the CLH queue to be freed by its successor. However, if the successor also aborts, the `QNode` is not freed, and the task is delegated to some other successor. When aborts are common, many `QNodes` are left unreclaimed in the CLH queue. Each thread maintains a list of `QNodes` it allocated (`local_list`). Every new acquisition sweeps its `local_list` looking for a recycled node. When successors keep aborting, none of the previously allocated `QNode` become available for reuse, and hence almost every new acquisition allocates a new `QNode` (via malloc) and appends it to its `local_list` The `local_lists` keep growing increasing both memory footprint and time spent unsuccessfully searching for a free node on each acquisition. Since the `local_list` searching time keeps growing, the timeout happens immediately after allocation, leaving no time to recycle an aborted predecessor. Frequent abort and lengthy allocation time feed back into each other further aggravating the problem.

The average time to sweep the list and allocate a new `QNode` took $\sim$900K CPU cycles in CLH_NB at 10K and 100K cycle timeout thresholds. The code spent more than 90% of the time trying to recycle `QNodes` in the `alloc_local_qnode` procedure incurring more than 97% of its L2 cache misses in the same procedure. As a result, CLH_NB and MCS_NB locks are unsuccessful in acquiring the lock within 10K CPU cycles. The allocation time reduced to 700 cycles only after the timeout was above 10^6 CPU cycles. Since the lock overhead consumes most of the execution time, it results in reduced throughput of non-critical work also. In contrast, HMCS-T's preallocated `QNodes` and their reuse keep the protocol overhead small.

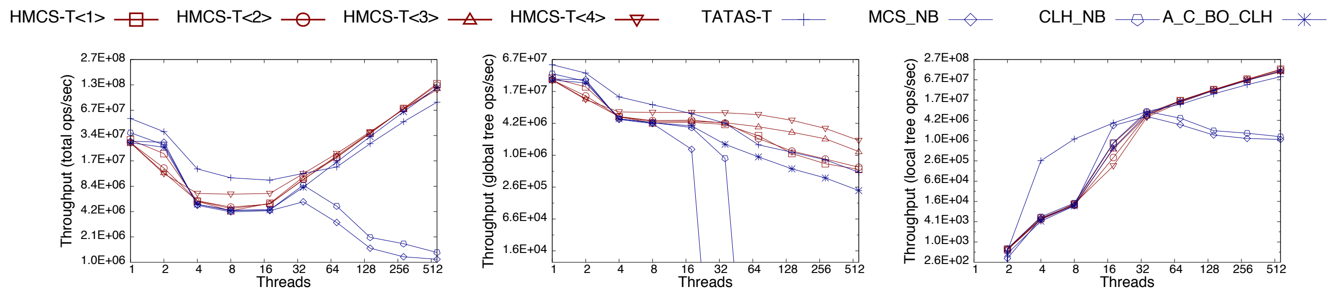
The memory usage and its overhead wrt TATAS-T at 10K timeout with 576 threads is shown in the left half of Table 2. CLH_NB and MCS_NB locks have very high memory usage due to repeated allocations when the nodes cannot be recycled. The numbers remain the same at 100K timeout (not shown). We measured that TATAS-T has a huge skew in the lock distribution—on 70% occasions, the lock holder is the same as the previous one causing thread starvation [19, 20]. A_C_BO_CLH also suffers from starvation.

To quantify the readmission into a previously abandoned `QNode`, we profiled how often a thread resumes from its abandoned node. HMCS-T$\langle 1\rangle$, HMCS-T$\langle 2\rangle$, HMCS-T$\langle 3\rangle$, and HMCS-T$\langle 4\rangle$ resumed their wait from a previously abandoned node on 99%, 97%, 92%, and 47% occasions, respectively at 10K timeout. A shallower lock is slow and has a longer queue, which provides more chance of readmitting an abandoned thread. The readmission fraction progressively reduces with increased depth. HMCS-T$\langle 4\rangle$ offers the least opportunity since the leaf-level queue length is at most two in our setting.

Figure 11(a) and 11(b) respectively show the throughput of both global and local tree operations at different abort thresholds ranging from 10^1 to 10^8 CPU cycles for a 576-threaded experiment. HMCS$\langle 4\rangle$ has the highest global-tree throughput followed by the other shallower hierarchical locks. Throughput over the global tree can drop by about $2\times$ between 10^1 to 10^6 cycles timeout. MCS_NB and CLH_NB become competitive for the global-tree operations only beyond 10^6 CPU cycles, which is a very high threshold. The local-tree throughput drops as the timeout value increases because few acquisitions timeout and hence few local-tree operations are performed.

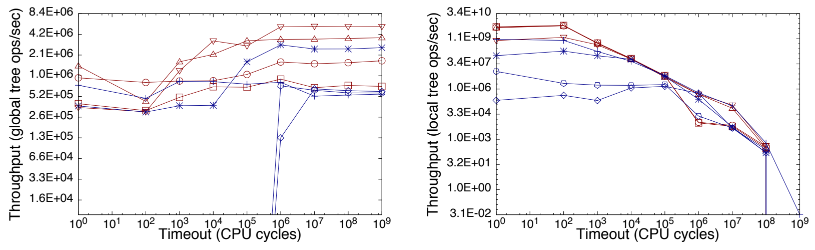
6.3 Lock Efficiency

To gain a unified view of the total work accomplished by a lock among a set of locks under study, we compute the following relative metrics for each lock k. $\boldsymbol{\mathcal{G}(k)}$: the ratio of lock k's global-tree throughput (critical work) to the maximum global-tree throughput by any lock under study with the same settings, and $\boldsymbol{\mathcal{L}(k)}$: the ratio of lock k's local-tree throughput (non-critical work) to the maximum local-tree throughput by any lock under study with the same settings. The sum of these two metrics can reach a theoretical maximum of 200%. We normalize it to 100%, which represents the efficiency $\boldsymbol{\mathcal{E}(k)}$ of a lock k. $\mathcal{E}$ quantifies the ability of



(a) Total splay operations. (b) Global splay operations. (c) Local splay operations.

Figure 10: Lock throughput of different abortable locks. Timeout=10K CPU cycles.



(a) Throughput of global tree operations. (b) Throughput of local tree operations.

Figure 11: Impact of timeout on the throughput of local and global tree operations. The experiment has 576 threads.

Table 2: Left: resident memory and memory overhead w.r.t. a TATAS-T lock. Right: efficiency ($\mathcal{E}$) of locks— a unified metric of critical and non-critical work accomplished. In both experiments, the timeout is set to 10K CPU cycles, and both experiments use 576 threads.

	Resident memory		% peak throughput		Efficiency
	KB	Overhead	Global (G)	Local (L)	E = (G+L)/2
HMCS-T<1>	271,060	1x	29%	100%	64%
HMCS-T<2>	271,160	1x	31%	96%	64%
HMCS-T<3>	271,192	1x	62%	88%	75%
HMCS-T<4>	271,188	1x	100%	84%	92%
TATAS-T	271,020	1x	25%	60%	43%
MCS_NB	3,746,764	14x	1.62E-06%	0.83%	0.41%
CLH_NB	1,783,500	6.6x	1.62E-06%	1%	0.50%
A_C_BO_CLH	271,170	1x	11%	90%	51%

a lock to deliver high throughput of successful lock acquisitions and also to expeditiously exit on timeout.

We computed $\mathcal{G}$, $\mathcal{L}$, and $\mathcal{E}$ metrics at 576 threads among eight locks at 10K cycle timeout (right side of Table 2). HMCS-T$\langle 4\rangle$ has the highest critical path throughput ($\mathcal{G}$ is 100%). Deep hierarchy helps in higher lock throughput, and the throughput falls off for shallower locks. CLH_NB and MCS_NB achieve only a small fraction of the peak throughput. TATAS-T and A_C_BO_CLH, respectively, achieve 25% and 11% of HMCS-T$\langle 4\rangle$'s global tree throughput.

HMCS-T$\langle 1\rangle$ has the highest throughput on the non-critical work ($\mathcal{L}$ is 100%). This is because HMCS-T$\langle 1\rangle$ is the quickest to abandon on timeout and switch to perform local tree operations. Abandoning in HMCS-T$\langle 1\rangle$ is a simple local swap operation at one level, whereas abandoning in an interior node of a deeper HMCS-T may involve more operations. $\mathcal{L}$ decreases for a deeper HMCS-T lock; HMCS$\langle 4\rangle$ achieves 84% of HMCS$\langle 1\rangle$'s non-critical work. Surprisingly, TATAS-T has lower $\mathcal{L}$ despite the fact that an abort takes no memory operation. This is because the `CAS` operation in TATAS-T has a very high latency. If a thread issues a `CAS` closer to its timeout, the `CAS` may complete thousands of cycles after the timeout, which leads to a high protocol overhead.

In the combined metric ($\mathcal{E}$), the deeper hierarchical locks—HMCS-T$\langle 3\rangle$ and HMCS$\langle 4\rangle$—deliver higher efficiency due to their exceeding superiority in critical work over shallower locks. TATAS-T and A_C_BO_CLH have respectable efficiency, but both are unfair locks. CLH_NB and MCS_NB have the lowest efficiency.

6.4 HMCS-T in an MPI+Threads BFS

Parallel breadth-first search (BFS) is an irregular, dynamic, sparse data exchange (DSDE) algorithm. A process communicates with a small subset of processes (neighbors) while the neighborhoods change dynamically over iterations. One of the best-known algorithms for solving DSDE problems is $\mathcal{NBX}$ [11]. In $\mathcal{NBX}$, most of the data exchanges rely on nonblocking point-to-point primitives and a process has to poll manually for communication progress via APIs such as `MPI_Iprobe` or `MPI_Test` in the context of MPI. We use Amer et al. [2]'s multi-threaded $\mathcal{NBX}$ implementation of the Graph500 BFS kernel, where threads in a process perform both computation and communication.

We use the MPICH-3.2 [1] MPI implementation, which has a single lock to protect its core functionality [3, 4]. In a highly threaded BFS run, the MPICH global lock becomes a point of contention because of repeated calls to `MPI_Test` by different threads. Since `MPI_Test` is only a progress checking API, it can be aborted in favor of computation or non-polling APIs, such as `MPI_Isend`. With this intuition, we substituted the MPICH-3.2 Pthread mutex lock with various abortable locks allowing the `MPI_Test` implementation to abort; *rest of the APIs do not abort.* On abort, `MPI_Test` behaves as if the communication has not yet completed. While the progress semantics of MPI require a continuously checking thread to succeed if a matching operation has been posted, such progress guarantee is conceivable in HMCS-T by bounding the number of successive aborts by a thread.

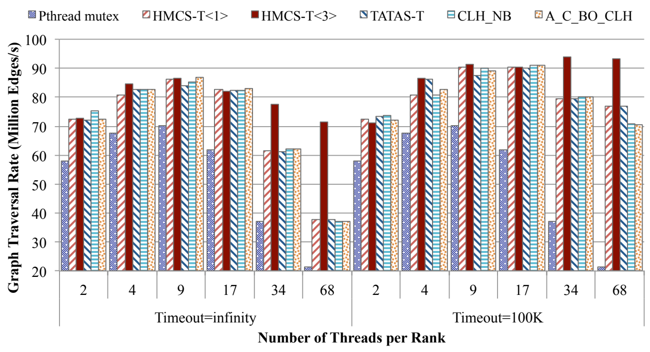


Figure 12: Scalability of BFS with 2^{25} vertices using 4 MPI processes and varying number of threads per MPI process. Pthread mutex, used by the original MPICH, does not support timeout but is only shown for a comparison. HMCS-T$\langle 3 \rangle$ has the best absolute performance and superior scaling at 10K timeout.

We partition the 8-node shared-memory machine into four MPI processes each with 72 cores; we do not use SMTs. Further, we do not load all sockets fully to leave some cores to run MPI daemons and scripts. We measured the BFS graph traversal rate from 16 different root vertices and report the average rate per root vertex in Figure 12 for various locks (including the baseline Pthread mutex) at two timeout values: 100K CPU cycles and infinity (for comparison with the traditional lock waiting regime). Our analysis showed that the overall performance is strongly correlated to the time spent in `MPI_Test` and that changing the lock implementation has little effect on the other parts of the algorithm, e.g., the computation or the amount of data transferred. The degree of computation-communication overlap dictates the performance variation across lock implementations.

At an infinite timeout, the performance peaks at nine threads for all locks. Beyond 17 threads, the performance collapses since more threads cause more wait in `MPI_Test`, hence more polling and less real communication. The only lock that proved to be more robust is HMCS-T$\langle 3 \rangle$, which exploits locality, however, it still suffers 17% performance loss at full concurrency compared to its peak.

With a timeout of 100K, every abortable lock performs superior to its timeout=∞ counterpart (the base Pthread mutex shown only for reference). At 100K timeout, the locks reach their peaks at 17 cores but this time the peaks are higher than those at timeout=∞. This indicates that bounding the lock wait time improves the ability to overlap communication with computation. All locks except HMCS-T$\langle 3 \rangle$ collapse in their performance beyond 17 cores since: 1) the cost of aborts is expensive in other locks due to inter-socket and inter-node communication, and 2) when the lock is passed to a remote requester, the locality is lost. HMCS-T$\langle 3 \rangle$ by virtue of its fast abort and locality-aware lock hand-off scales to a higher throughput (93.2 M edges/s) at 34 and 68 threads and alleviates the performance collapse seen in the other locks. In particular, HMCS-T$\langle 3 \rangle$ with a 100K timeout achieves 30% higher peak performance than its unbounded counterpart. Furthermore, HMCS-T$\langle 3 \rangle$ with 100K timeout retains the same 30% speedup even when compared against HMCS$\langle 3 \rangle$ (not shown) although HMCS$\langle 3 \rangle$ is expected to have slightly less protocol overhead compared to HMCS-T$\langle 3 \rangle$ with infinite timeout.

The top performance of HMCS-T$\langle 3 \rangle$ (at 34-68 threads) is 34% higher than the top performance of the original Pthread mutex (at 9 threads). At full concurrency, HMCS-T$\langle 3 \rangle$ delivers 88% and 21% higher throughput than the next best performing abortable lock (HMCS-T$\langle 1 \rangle$) with an infinite and 100K timeout values, respectively. This justifies the virtue of both locality awareness and abortability. HMCS-T$\langle 3 \rangle$ at a 100K timeout (with 93.2×10^6 edges/s) shows an impressive 330% (4.3$\times$) speedup over the original Pthread mutex-based MPICH (with 21.5×10^6 edges/s) at full concurrency. We also experimented with a `CAS`-based try-lock (not shown) and observed a very poor scalability and a very low peak performance since it is locality agnostic and generates indiscriminate cache coherency traffic.

We experimented the previously mentioned `IsLocked()` primitive with both HMCS-T$\langle 3 \rangle$ (at several timeout values) and HMCS$\langle 3 \rangle$. We invoked the underlying lock iff the `IsLocked()` returned `NOT_LOCKED`. We observed that the graph traversal rate initially increased up to 140 million edges/sec, but once the number of threads crossed a socket, the throughput collapsed dramatically down to below 90 million edges/sec because of the lost data locality. `IsLocked()`+HMCS-T$\langle 3 \rangle$ showed slightly worse performance than merely using HMCS-T$\langle 3 \rangle$ for the same timeout values. `IsLocked()`+HMCS$\langle 3 \rangle$ performed worse than HMCS-T$\langle 3 \rangle$ with $\leq$ 10K cycles timeout regardless of the thread count.

These results clearly indicate that bounding lock waiting times through an abort feature can help scale an otherwise non-scalable code by exploiting parallel slackness.

7. Conclusions

In this paper, we designed the HMCS-T lock, a hierarchical queuing lock that can abandon the lock wait on a timeout. Hierarchical queue-based locks combined with the timeout feature offer two key advantages on many-core NUMA systems—data locality and enhanced concurrency. HMCS-T has bounded space and time complexity together with starvation freedom, which are rare in other abortable locks. HMCS-T has high timing fidelity and outperforms several state-of-the-art abortable locks in latency and throughput.

8. Acknowledgments

John Mellor-Crummy and Karthik Murthy shared their model checking experience, which inspired us to develop our solution. Ram Swaminathan made valuable editorial suggestions. This research was supported by the National Science Foundation under grant number 1464157 and by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] A. Amer, P. Balaji, W. Bland, W. Gropp, R. Latham, H. Lu, L. Oden, A. Pena, K. Raffenetti, S. Seo, T. Rajeev, and Z. Junchao. MPICH User's Guide, Version 3.2. `http://www.mpich.org/static/downloads/3.2/mpich-3.2-userguide.pdf`, 2015.

[2] A. Amer, H. Lu, P. Balaji, and S. Matsuoka. Characterizing MPI and Hybrid MPI+Threads Applications at Scale: Case Study with BFS. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1075–1083. IEEE, 2015.

[3] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. MPI+Threads: Runtime Contention and Remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 239–248, New York, NY, USA, 2015. ACM.

[4] A. Amer, H. Lu, Y. Wei, H. Jeff, S. Matsuoka, and P. Balaji. Locking Aspects in Multithreaded MPI Implementations. Technical Report ANL/MCS-P6005-0516, 2016.

[5] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 215–226, 2015.

[6] M. Chabbi and J. Mellor-Crummey. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 22:1–22:14, New York, NY, USA, 2016. ACM.

[7] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[8] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 247–256, 2012.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[10] Hewlett Packard Enterprise. HP Integrity Superdome X. `http://www8.hp.com/h20195/v2/GetPDF.aspx/c04383189.pdf`.

[11] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable Communication Protocols for Dynamic Sparse Data Exchange. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 159–168, New York, NY, USA, 2010. ACM.

[12] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering — Special issue on formal methods in software practice*, 23(5):279–295, May 1997.

[13] Intel Corp. An Introduction to the Intel$^{®}$ QuickPath Interconnect. `http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html`, 2009.

[14] P. Jayanti. Adaptive and Efficient Abortable Mutual Exclusion. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 295–304, New York, NY, USA, 2003. ACM.

[15] P. S. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, 1994.

[16] V. Marathe, M. Moir, and N. Shavit. Composite Abortable Locks. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 1–10, April 2006.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[18] A. Pareek and P. Woelfel. RMR-efficient Randomized Abortable Mutual Exclusion. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 267–281, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] M. L. Scott. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 31–40, New York, NY, USA, 2002. ACM.

[20] M. L. Scott and W. N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 44–52, New York, NY, USA, 2001. ACM.

[21] Scott, Michael. Non-Blocking Timeout in Scalable Queue-Based Spin Locks. `https://www.cs.rochester.edu/research/synchronization/pseudocode/nb_timeout.html`.

[22] SGI. SGI UV The World's Most Powerful In-Memory Supercomputers. `https://www.sgi.com/products/servers/uv/`.

[23] D. D. Sleator and R. E. Tarjan. Self-adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[24] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.