# High Performance Locks for Multi-level NUMA Systems

Milind Chabbi Michael Fagan John Mellor-Crummey
Rice University, Houston, TX, USA, 77005
milind.chabbi, mfagan, johnmc @rice.edu

## Abstract

Efficient locking mechanisms are critically important for high performance computers. On highly-threaded systems with a deep memory hierarchy, the throughput of traditional queueing locks, e.g., MCS locks, falls off due to NUMA effects. Two-level cohort locks perform better on NUMA systems, but fail to deliver top performance for deep NUMA hierarchies. In this paper, we describe a hierarchical variant of the MCS lock that adapts the principles of cohort locking for architectures with deep NUMA hierarchies. We describe analytical models for throughput and fairness of Cohort-MCS (C-MCS) and Hierarchical MCS (HMCS) locks that enable us to tailor these locks for high performance on any target platform *without empirical tuning*. Using these models, one can select parameters such that an HMCS lock will deliver better fairness than a C-MCS lock for a given throughput, or deliver better throughput for a given fairness.

Our experiments show that, under high contention, a three-level HMCS lock delivers up to 7.6× higher lock throughput than a C-MCS lock on a 128-thread IBM Power 755 and a five-level HMCS lock delivers up to 72× higher lock throughput on a 4096-thread SGI UV 1000. On the K-means clustering code from the MineBench suit, a three-level HMCS lock reduces the running time by up to 55% compared to the C-MCS lock on a IBM Power 755.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming

***General Terms*** Algorithms, Design, Performance

***Keywords*** NUMA, MCS, Hierarchical locks, Spin locks, Analytical modeling, Lock fairness, Lock throughput

## 1. Introduction

Over the last decade, the number of hardware threads in shared-memory systems has grown dramatically. Multiple hardware threads executing on a core share one or more levels of private cache. Multiple cores share caches on the same die. Multi-socket systems share memory on a node, and multi-node shared-memory systems (e.g, SGI UV 1000 [9]) share memory across the entire system.

Modern architectures organize hardware threads into clusters known as NUMA (Non-Uniform Memory Access) domains. Each NUMA domain of threads experiences a spectrum of memory access latencies, depending upon the level of the memory hierarchy where the data is resident; the core, socket, or node where the data is resident; and the NUMA domain that is the home location for the data in memory. A memory location gains proximity to a thread when the location is accessed by the thread itself *or by other threads in the same NUMA domain*. Conversely, a memory location loses proximity when the location is modified by a thread farther away in the NUMA hierarchy. Thus, the latency when a thread accesses a datum depends on the NUMA domain of the thread that wrote it last. Efficient use of deep NUMA systems requires exploiting locality. Once a memory location is cached in a NUMA domain, it is beneficial to reuse it multiple times. Ideally, the same thread or its NUMA peers would reuse the data. The next best alternative is access by threads in a NUMA domain nearby.

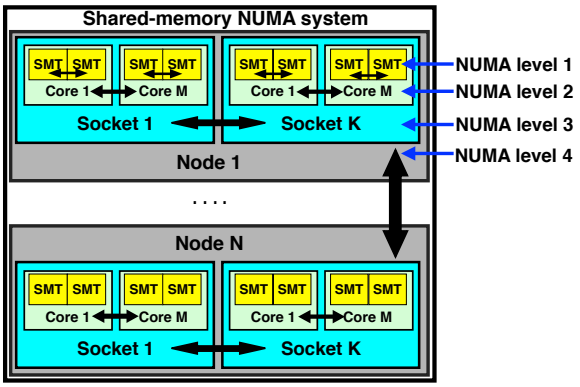



**Figure 1.** Shared-memory NUMA system.

Figure 1 illustrates a typical 4-level NUMA system. Each pair of SMT threads (**S**imultaneous **M**ulti **T**hreading [10]—the most common style of hardware multithreading) sharing a core form the first level of NUMA hierarchy. M different CPU cores sharing the same socket form the second level of NUMA hierarchy. K sockets on the same node form the third level of NUMA hierarchy. All nodes together form the fourth and last level of NUMA hierarchy.

Deep NUMA hierarchies pose a challenge for efficient mutual exclusion in multi-threaded programs. Software queueing locks, namely the MCS [7] lock and CLH [6] lock, are known as scalable locks for shared-memory systems. An advantage of queueing locks is that after a thread enqueues itself for a lock, it spins waits locally, which avoids clogging the interconnect. For this reason, queuing locks scale well in the presence of contention. However, throughput for these locks falls off on NUMA systems.

On deep NUMA hierarchies, a lock and data accessed in a critical section protected by the lock ping-pong between NUMA domains, resulting in lower lock throughput. Figure 2 shows how



*PPoPP'15*, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688503

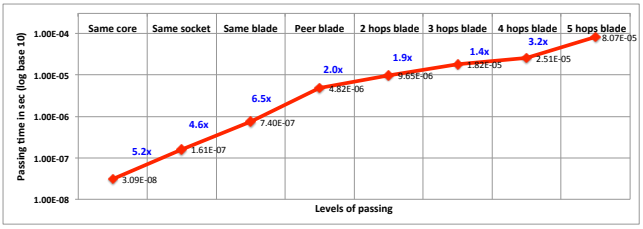



**Figure 2.** MCS lock passing times on SGI UV 1000.

the lock-passing latency grows on a SGI UV 1000 as the data becomes farther away from the requesting processor core.

Dice et al. [4] partially address this problem with "lock cohorting", which increases lock throughput by passing the lock among threads within the same NUMA domain before passing the lock to any thread in a remote NUMA domain. This design addresses only a two-level NUMA hierarchy arising from multiple sockets on a node. For the system in Figure 1, cohort locks would only exploit locality between threads sharing the same socket, leaving other levels of locality unexploited. Deep NUMA hierarchies offer additional opportunities for exploiting locality between SMT threads sharing a cache or within a node.

In this paper, we present hierarchical MCS locks (HMCS)—a full generalization of lock cohorting that takes advantage of each level of NUMA hierarchy. The HMCS lock is designed to provide efficient mutual exclusion for highly-contended critical sections in NUMA architectures. The HMCS lock is modeled as a composition of classical MCS locks at each level of NUMA hierarchy. By usually passing a lock to a waiting thread in the same or a nearby NUMA domain, the HMCS lock fully exploits locality on multi-level NUMA systems.

The contributions of this paper are: (1) a novel, fully-general, queuing lock suitable for mutual exclusion in highly contended critical sections in multi-level NUMA systems, (2) analytical performance models for throughput and fairness of queuing locks on NUMA systems that eliminate empirical tuning, (3) proofs for throughput and fairness superiority of multi-level queuing locks over state-of-the-art two-level locks under high contention, (4) demonstration of up to $7.6\times$ higher throughput over state-of-the-art C-MCS locks on a 128-thread IBM Power 755 and up to $72\times$ higher throughput on a 4096-thread SGI UV 1000, and (5) demonstration of $9.2\times$ speedup compared to the original non-scaling K-means clustering code (55% improvement compared to the C-MCS lock) on a IBM Power 755.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 explains the HMCS algorithm. Section 4 discusses analytical performance models. Section 5 proves properties of HMCS locks. Section 6 presents an empirical evaluation of HMCS locks. Finally, Section 7 summarizes our conclusions.

## 2. Background and Related Work

Queuing lock algorithms form the basis for the current work. In this section, we define some terminology used in the paper, followed by details about the MCS and CLH queuing locks. We also describe the state-of-the-art in analyzing lock characteristics.

***Terminology:*** We refer to a system with N-levels of NUMA hierarchy as an *N-level system*. We refer to a NUMA domain as simply a *domain*. When the domain is important, we will refer to a *level-k domain*. We also freely abbreviate "critical section" as *CS*. Where necessary, we distinguish one thread from another with dot-separated hierarchical numbering, with left-most number representing the cardinality of the outer-most NUMA-domain. For example, in Figure 1, thread 1.2.8.3 means the $3^{rd}$ SMT thread of the $8^{th}$ core of the $2^{nd}$ socket on the $1^{st}$ node.

***MCS locks:*** The MCS [7] lock acquire protocol enqueues a record in the queue for a lock by: 1) swapping the queue's tail pointer with a pointer to its own record and 2) linking behind a predecessor (if any). If a thread has a predecessor, it spins locally on a flag in its record. Releasing an MCS lock involves setting a successor's flag or resetting the queue's tail pointer to null if no successor is present.

The Cohort MCS (C-MCS) lock by Dice et al. [4] for NUMA systems employs two levels of MCS locks, treating a system as a two-level NUMA hierarchy. (Dice et al. refer to this lock as C-MCS-MCS; we use the shorter C-MCS for convenience.) One MCS lock is local to each NUMA domain and a global MCS lock is shared by all NUMA domains. Each thread trying to enter a critical section acquires the local lock first. The first thread to acquire the local lock in each domain proceeds to compete for the global lock, while other threads in each domain spin wait for their local lock. A releasing thread grants the global lock to a local successor by releasing the local lock. A lock passes within the same domain for a *threshold* number of times at which point the domain relinquishes the global lock to another domain. Dice et al. [4] also explore other lock cohorting variants beyond C-MCS that employ various locking protocols at each of the two levels, with the local and global locking protocols selected independently. We focus on MCS lock at each level of the HMCS lock, and leave exploring different locks at different levels for the future work. Moreover, even Dice et al.'s best performing backoff and MCS lock combination (C-BO-MCS) is only marginally better than the second best C-MCS-MCS lock. Furthermore, the C-BO-MCS lock has unbounded unfairness, where as the C-MCS-MCS lock has bounded unfairness.

Dice et al. also devised a flat-combining MCS lock [3] (FC-MCS). FC-MCS builds local queues of waiting threads and employs a designated thread to join them into a global MCS queue.

***CLH locks:*** The CLH lock [6] is analogous to the MCS lock, with the following two differences: (1) each thread enqueues its own record into a queue when acquiring a lock, but does not reclaim its record on release; instead, its successor is responsible for its record, and (2) a thread waiting to acquire a lock waits on a flag in its predecessor's record rather than its own.

The HCLH lock [5] is a variant of the CLH lock that is tailored for 2-level NUMA systems. Threads on cores of the same chip form a local CLH queue. The thread at the head of the queue splices the local queue into the global queue. The splicing thread may have to wait for a long time or splice a very short queue of local threads, both of which lengthen the critical path.

While the CLH lock could be generalized into a hierarchical lock with similar properties, in the remainder of this paper we focus solely on MCS locks. We note that *none* of the aforementioned locks exploit more than two levels of locality in a multi-level NUMA system.

***Analysis of lock characteristics:*** There is limited prior work in analytical modeling of lock *throughput*. Boyd-Wickizer et al. [1] use Markov models for ticket-based spin locks to reason about an observed collapse in performance of several threaded applications on Linux. We are unaware of any analytical modeling of lock *fairness*. Buhr et al. [2] conduct an empirical study of fairness of various locks. To the best of our knowledge, our paper is the first to provide *combined throughput-fairness* analytical models for locks.

## 3. HMCS Lock Algorithm

The HMCS lock extends the two-level cohorting strategy of Dice et al. [4] to a general N-level NUMA hierarchy. The HMCS lock

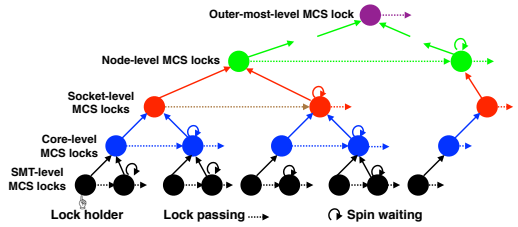



**Figure 3.** Hierarchical MCS lock.

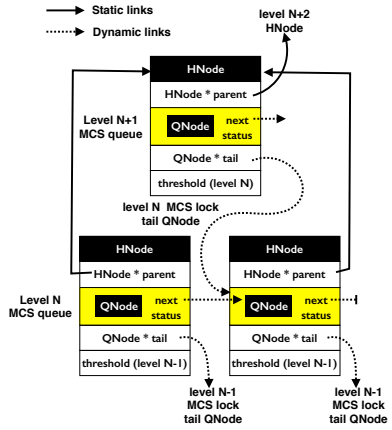



**Figure 4.** HMCS lock key data structures.

employs MCS locks at multiple levels of the hierarchy to exploit locality at each level. An HMCS lock is an n-ary tree of MCS locks, shown in Figure 3. A critical section can be entered only when all locks along a path from a leaf to the root of the tree are held by the acquiring thread. *Not all threads, however, explicitly acquire all locks along a path from leaf to root to enter the CS.* A thread holding the lock, after finishing its CS, passes the lock to a successor in its NUMA domain. After a *threshold* number of lock transfers within a NUMA domain, a thread passes the lock at the lowest enclosing ancestor NUMA domain where a thread from a peer domain is waiting. This chain of passing to a successor thread in the current or peer domain continues throughout the protocol. *Under high contention, threads acquiring the lock via lock passing, incur the cost of only a single leaf-level MCS lock acquisition.* The HMCS lock, like the C-MCS and HCLH locks, does not ensure system-wide FIFO ordering; however, it ensures FIFO ordering within requesting threads at the same level of the NUMA hierarchy.

***Initial setup:*** Initialization of an HMCS lock involves creating a tree of MCS locks, with a lock for each domain at each level of the NUMA hierarchy. If exploiting locality at each level of the hierarchy is not profitable, one may create a shallower tree. For every non-leaf node of the tree, we preallocate a lock record within the corresponding NUMA domain. These preallocated records remain for the lifetime of the lock.

***Key data structures:*** The HMCS algorithm employs two key data structures: `QNode` and `HNode`. The `QNode` is a modification of the original MCS lock's `QNode`, with its boolean status field replaced by a 64-bit integer. All `HNodes` representing a lock form a tree as shown in Figure 3. As shown in Figure 4, each `HNode` has the following fields:

- `lock`: the tail pointer of the MCS lock at this level,
- `parent`: a pointer to the parent `HNode` (if any),
- `QNode`: a node to enqueue when acquiring the parent `HNode`'s MCS lock. This `QNode` is shared by all threads in the subtree of the NUMA hierarchy represented by this `HNode`, and

```
1  enum {COHORT_START=0x1, ACQUIRE_PARENT=UINT64_MAX-1, WAIT=UINT64_MAX};
2  enum {UNLOCKED=0x0, LOCKED=0x1};
3
4  template<int depth> struct HMCS {
5    inline void AcquireReal(HNode *L, QNode *I) {
6      // Prepare the node for use.
7      I->status = WAIT; I->next = NULL;
                 ---------- release fence ----------
8      QNode * pred = (QNode *) SWAP(&(L->lock), I);
9      if (pred) {
10       pred->next = I;{
11       uint64_t curStatus;
12 ▶     while( (curStatus=I->status) == WAIT); // spin
13 ▶     if(curStatus < ACQUIRE_PARENT)  return; // Acquired, enter CS
14     }
15     I->status = COHORT_START;
16     HMCS<depth-1>::AcquireReal(L->parent, &(L->node));
17   }
18
19   inline void Acquire(HNode *L, QNode *I) {
20     AcquireReal(L, I);
21 ▶     ---------- acquire fence ----------
22   }
23
24   inline  void ReleaseReal(HNode *L, QNode *I) {
25 ▶   uint64_t curCount = I->status;
26     // Lower level releases
27 ▶   if (curCount == L->GetThreshold()) {
28       // reached threshold, release to next level
29 ▶     HMCS<depth-1>::ReleaseReal(L->parent, &(L->node));
                 ---------- release fence ----------
30       // Ask successor to acquire next level lock
31       ReleaseHelper(L, I, ACQUIRE_PARENT);
32       return; // Released
33     }
34     // Not reached threshold
35 ▶   QNode * succ = I->next;
36 ▶   if (succ) {  // Pass within cohorts
37 ▶     succ->status = curCount + 1;
38       return;
39     }
40     // else: No known successor, release to parent
41     HMCS<depth-1>::ReleaseReal(L->parent, &(L->node));
                 ---------- release fence ----------
42     // Ask successor to acquire next level lock
43     ReleaseHelper(L,I, ACQUIRE_PARENT);
44   }
45
46   inline void Release(HNode *L, QNode *I) {
47 ▶     ---------- release fence ----------
48     ReleaseReal(L, I);
49   }
50 };
51
52 void ReleaseHelper(HNode *L, QNode *I, uint64_t val){
53 ▶   QNode * succ = I->next;
54 ▶   if (succ) {
55 ▶     succ->status = val; // pass lock
56     } else {
57       if (CAS(&(L->lock), I, NULL)) return;
58       while( (succ=I->next) == NULL);  // spin
59       succ->status = val; // pass lock
60     }
61 }
62
63 template <> struct HMCS<1> {
64   inline void AcquireReal(HNode *L, QNode *I) {
65     // Prepare the node for use.
66     I->status = LOCKED; I->next = NULL;
67     QNode * pred = (QNode *) SWAP(&(L->lock), I);
68     if (!pred) {
69       I->status = UNLOCKED;
70     } else {
71       pred->next = I;
72 ▶     while(I->status == LOCKED); // spin
73     }
74   }
75
76   inline void Acquire(HNode *L, QNode *I) {
77     AcquireReal(L, I);
78 ▶     ---------- acquire fence ----------
79   }
80
81   inline void ReleaseReal(HNode *L, QNode *I) {
82     ReleaseHelper(L, I, UNLOCKED);
83   }
84
85   inline void Release(HNode *L, QNode *I) {
86 ▶     ---------- release fence ----------
87     ReleaseReal(L, I);
88   }
89 };
```

**Listing 1.** Hierarchical MCS lock.

- `threshold`: the lock passing threshold at this level.

***Acquire protocol:*** The HMCS lock algorithm is shown in Listing 1. The acquire protocol begins at the leaf (level-1) of the tree. Each thread arrives with its own `QNode` and a pointer to the lock—an `HNode` used by a group of peer threads at level-1 in the hierarchy. Each thread attempts to acquire the local level-1 MCS lock. The first thread to enqueue at each level-1 domain acquires the local

MCS lock; other threads in the same domain spin wait on the status field of their QNode. The threads that acquire the level-$i$ MCS lock in their respective domains proceed to acquire the level-$(i+1)$ MCS lock, competing with threads from peer NUMA domains. This continues until a single thread acquires all MCS locks along the path to the root (inclusive), at which point the acquisition is complete and the thread enters the critical section.

A thread, T, that is not the first to arrive at a level (say $K$) spin waits on the status field of the enqueued QNode at level $K$ (line 12 in Listing 1). Eventually one of the following happens for a thread waiting at any level:

- (Line 13 in Listing 1) T's predecessor in the MCS lock at level $K$ passes the lock to T, which completes the acquire protocol and T immediately enters the CS.
- (Line 13 Listing 1) The quantum of lock passing exhausts at level $K$, in which case T proceeds to compete for the level-$(K+1)$ MCS lock (Line 15, 16 Listing 1). Thread T also prepares for a full quantum of passing for the next round within the domain by resetting the status field of that domain's QNode.

***Release protocol:*** While the classical MCS lock uses a boolean status flag to pass a lock to its successor, the HMCS lock uses a 64-bit integer in the QNode to encode the current pass count. When passing a lock to a successor level of the NUMA hierarchy, the releasing thread writes the current pass count for that level into the status field of its successor. This serves both to release the waiting successor as well as convey the local pass count. Furthermore, this technique eliminates the need for a shared counter.

The release protocol begins at the leaf of the tree. If the passing threshold has not yet been reached at a leaf lock, and there is a waiting successor within that domain, then the releaser (R) passes the lock to its waiting peer within the domain. Otherwise, R relinquishes the lock at the deepest node along the path to the root where a peer is waiting and the passing threshold has not been exceeded.

When releaser R encounters a threshold that has been exceeded at some L->lock other than the tree root, R recursively performs the release protocol (Line 29 in Listing 1) at L->parent before signaling its successor S at L->lock. If S is non-null, R signals S by setting its status to ACQUIRE_PARENT, which indicates that S must compete for the MCS lock at the parent level (L->parent->lock). If R were to signal S before releasing L->parent, there would be a data race between R and S for the QNode they both use (L->node) to interact with the MCS lock at the parent level.

Attempting to pass a lock to a thread in the same NUMA domain before relinquishing the lock to the next level enhances reuse of shared data. Limiting the number of local passes ensures starvation freedom. If the passing threshold is exceeded, but there is a successor at the current level whereas no threads in other domains are waiting, then we retain the lock within the domain for another round. This optimization is not shown in Listing 1.

***Relaxed memory models:*** Listing 1 shows the fences necessary for the HMCS lock on systems with processors that use weak ordering.

### 3.1 Discussion

***Latency vs. throughput:*** Clearly, the cost of occasionally manipulating a sequence of locks along a path in a tree is more expensive than working with a single lock. For highly contended locks, however, this cost is outweighed by the benefits of increased locality that result from sharing within a NUMA domain. The HMCS design improves throughput at the expense of latency and it is *best suited for highly contended locks.* To reduce latency, a client can use a shallower HMCS tree, though doing so would reduce the lock's peak potential throughput. With the templated design, we can instantiate the HMCS lock as a classical MCS lock (HMCS$\langle 1 \rangle$), a 2-level cohort lock (HMCS$\langle 2 \rangle$), or use a larger template constant for a deeper hierarchy. Although, the code in Listing 1 uses compile time instantiation, one could write a non-templated HMCS lock by using recursion.

***Memory management:*** The HMCS lock needs no explicit memory management after initialization. On the other hand, Dice et al.'s cohort MCS lock requires maintaining a pool of free nodes. Hence, HMCS lock's memory management is superior.

***Cache policy:*** Performance benefits of the HMCS lock can vary based on the caching policies such as inclusive vs. exclusive. By passing the lock to a nearby thread more often, the HMCS lock benefits from data locality *irrespective* of the caching policy.

***Thread binding:*** The code in Listing 1 is agnostic to operating-system (OS) thread to hardware (HW) thread bindings. The thread calling the HMCS lock's Acquire and Release routines is responsible for using the correct L—the pointer to its leaf-level HNode. If the OS thread is bound to a HW thread belonging to the same innermost NUMA domain, then the program can set the value of L once during lock initialization and use it for the duration of the program.

To address thread migration across NUMA domains, we replace the L argument in the Acquire and Release routines to point to an array of pointers to leaf-level HNodes. The Acquire *indexes* into this array to obtain the pointer to its HNode. The Acquire will also remember the inferred value of the index, which the matching Release will use. The index itself will be a cached, thread-local value derived by querying the CPU that the thread is currently running on. We register a signal handler with Linux perf events for task migration. On task migration, the handler invalidates the migrating thread's cached CPU index. An acquire following a migration, gets the new CPU index either via an architecture-specific instructions such as RDTSCP or via the getcpu system call.

## 4. Performance Metrics

Two key governing design criteria for any locks are their *throughput* and *fairness*. The HMCS and C-MCS locks are designed to deliver high throughput under heavy contention. For this reason, we evaluate throughput and fairness of these locks in the fully-contended case. In the rest of this paper, reference to C-MCS simply means HMCS$\langle 2 \rangle$ .

One can evaluate locks under full contention in various ways. Mellor-Crummey et al. [7] suggest continuous lock acquisitions and releases with an empty critical sections to assess the *pure overhead* of locks. Dice et al. suggest adding a delay outside the CS and accessing a few cache lines of shared data inside the CS. Each evaluation method has its own advantages and disadvantages. Dice et al.'s technique of accessing sharing data can give a false impression of reduced lock overhead since the shared data is also passed between NUMA domains. The larger the shared data accessed in the CS, the lower the observed overhead of the lock. Further, there is never an appropriate delay to choose for the time spent outside a CS. On the other hand, Mellor-Crummey's tight loop ignores the actual gain possible on a real program, which would certainly access some shared data inside CS; passing shared data within NUMA domains has nontrivial benefits. Finally, if the data accessed inside a CS is so large that it evicts critical lock data structures, both Dice and Mellor-Crummey's techniques are moot. Being aware of these differences and deficiencies, we adopt Mellor-Crummey's tight loop technique to quantify the pure lock overhead in the absence of external interference from the program.

Let $n_i$ be the number of peers in domain i. The lowest possible domain is the L1 cache of a core, so $n_1$ is the number of hardware threads per core. Analogously, $n_2$ is the number of cores per chip, $n_3$ is the number of chips / socket, etc. Given this definition, it is clear that the total number of threads in the system is $\prod_{i=1}^{N} n_i$.

### 4.1 Throughput

**Definition 1** (Lock throughput)**.** *Lock throughput, $\mathcal{T}_k$, of a lock $k$, is the average number of lock acquisitions by $k$ per unit time.*

**Definition 2** (Critical path)**.** *The critical path is the longest sequence of weighted operations in the execution of a concurrent program, where operation weights correspond to their latencies.*

Naturally, all statements in the critical section (CS) are on the critical path. The time to execute the statements in the CS is a property of the program. For assessing lock overhead, however, we assume an empty CS. All statements that are executed from the time the lock is granted to the time the lock is passed to the next waiting thread contribute to the critical path. Not all statements in the lock-acquire and lock-release protocols contribute to the critical path. Under high contention, a large number of threads will be waiting to acquire the lock. The thread releasing the lock will have a successor linked behind it at each level in the hierarchy. Each waiting thread will be spin-waiting at line 12 in Listing 1 in the acquire protocol. We have marked all statements that *may* appear on the critical path with a "▶" symbol in Listing 1.

The following statements in the acquire protocol appear on the critical path: 1) the last trip through the spin-wait loop after the lock is granted, and 2) the check to ensure that the passing threshold was not exceeded.

The following statements in the release protocol appear on the critical path: 1) loading of the status field, 2) checking if the passing threshold has reached, 3) relinquishing the global lock to the parent level when the passing threshold is reached, and 4) checking for the presence of successor and subsequent global lock passing to the successor when the passing threshold has not reached.

If the lock is relinquished to the parent level, the sequence of statements executed until the global lock is granted to another domain contribute to the critical path. Subsequent signaling of successors at lower levels is *not* on the critical path. In a fully contended system, there is always a successor waiting to acquire the lock, hence the case of not having a successor (line 56 in Listing 1) is not on the critical path.

Our implementation takes advantage of C++ template's value specialization to unroll recursion. Template specialization allows the deepest level of recursion which manipulates the root lock to be implemented with lower overhead. When the acquisition involves several layers of recursion, tail recursion reduces the critical path length since the lock acquiring thread can enter the CS with a single return statement from the last level of recursion. Furthermore, we engineered the code to reduce branches on the critical path (not shown in Listing 1.)

***A variant of the C-MCS lock:*** While Dice et al.'s C-MCS lock formed cohorts within threads on the same socket, it ignored the other levels of NUMA hierarchy inside a socket. We propose a variant of C-MCS lock where cohorts are formed only at the inner level (e.g., among SMT threads). All other levels of the NUMA hierarchy are ignored. We call Dice et al.'s original C-MCS lock an *outer* cohorting lock (C-MCS$_{\text{out}}$). We call aforementioned variant of the C-MCS an *inner* cohorting lock (C-MCS$_{\text{in}}$).

For simplicity, in Section 4.1.1- 4.1.3 we assume a 3-level system and build analytical models of throughput for C-MCS$_{\text{in}}$, C-MCS$_{\text{out}}$ and HMCS$\langle 3\rangle$ locks. We provide throughput models of the HMCS lock for any N-level system at the end of Section 4.1.3.

#### 4.1.1 Throughput of the C-MCS$_{\text{in}}$ Lock

In a C-MCS$_{\text{in}}$ lock, threads at level-1 of the NUMA hierarchy (e.g., SMT threads) form cohorts. On reaching the passing threshold, however, a C-MCS$_{\text{in}}$ lock may relinquish the lock either to a level-2 NUMA peer (e.g., a thread on the same socket) or to a level-3 NUMA peer (e.g, a thread on another socket). Consider a chain of


**Figure 5.** Lock passing in the C-MCS$_{\text{in}}$ lock.

successive lock passing as shown in Figure 5. The first thread in the cohort acquires the lock either from a level-2 or level-3 peer in the NUMA hierarchy. Let $Acq_2$ and $Acq_3$ be the critical path lengths if the lock is acquired from a level-2 or level-3 peer respectively. Since the level-3 memory access latency is larger than level-2 memory access latency, $Acq_3 > Acq_2$. The expected critical path length to acquire the lock either from a level-2 or level-3 NUMA domain peer is given by:

$$\begin{aligned} Acq_{2\oplus 3} =& Pr[Acquisition\ from\ level\ 2]Acq_2 \\ &+(1 - Pr[Acquisition\ from\ level\ 2])Acq_3 \end{aligned} \tag{1}$$

There are $(n_2 n_3 - 1)$ other domains in the system of which $(n_2 - 1)$ are peers at level 2. Hence,

$$Pr[Acquisition\ from\ level\ 2] = \frac{n_2 - 1}{n_2 n_3 - 1} \tag{2}$$

$$\implies Acq_{2\oplus 3} = \frac{n_2 - 1}{n_2 n_3 - 1} Acq_2 + \frac{n_2 n_3 - n_2}{n_2 n_3 - 1} Acq_3 \tag{3}$$

Let $Rel_1$ and $Acq_1$ be the critical paths in the release and acquire protocols when releasing and acquiring at level-1 respectively. We represent $Rel_1 + Acq_1 = p_1$ as the *lock passing* time at level 1. If $c_{in}$ is the passing threshold for the lock, there will be $(c_{in} - 1)$ lock passings at level 1, each adding $p_1$ to the critical path length. At the end, the lock will be released to a thread belonging to either level-2 or level-3 domains and the expected cost of releasing ($Rel_{2\oplus 3}$) is defined similar to $Acq_{2\oplus 3}$. We represent $Acq_{2\oplus 3} + Rel_{2\oplus 3}$ as $p_{2\oplus 3}$, where $p_2$ and $p_3$ are the passing times at level 2 and 3 respectively. The expected passing time is:

$$p_{2\oplus 3} = \frac{n_2 - 1}{n_2 n_3 - 1} p_2 + \frac{n_2 n_3 - n_2}{n_2 n_3 - 1} p_3 \tag{4}$$

The cycle of remote acquisition, local passing, and remote release repeats in each domain. The C-MCS$_{\text{in}}$ lock acquires $c_{in}$ number of locks in time $p_{2\oplus 3} + (c_{in} - 1)p_1$. Hence the throughput of a C-MCS$_{\text{in}}$ lock, $\mathcal{T}_{in}$, in a 3-level system is given by:

$$\mathcal{T}_{in}(3) = \frac{c_{in}}{p_{2\oplus 3} + (c_{in} - 1)p_1} \tag{5}$$

As $c_{in} \to \infty$, $\mathcal{T}_{in}(3) \to 1/p_1$. Maximum throughput of the C-MCS$_{\text{in}}$ lock, $\mathcal{T}_{in}^{max}$, in a 3-level system is given by:

$$\mathcal{T}_{in}^{max}(3) = \frac{1}{p_1} \tag{6}$$

This bound holds for any N-level system.

#### 4.1.2 Throughput of the C-MCS$_{\text{out}}$ Lock

In this lock, cohorts are formed among threads belonging to level-1 and level-2 NUMA domains aggregated and no distinction is made between these two levels of hierarchy. Let $c_{out}$ be the passing threshold. Let the cost of acquisition and release from an outside domain (at level-3) be $Acq_3$ and $Rel_3$ respectively. We represent $Acq_3 + Rel_3$ as $p_3$ — the lock passing time at level 3. The expected cohort passing time within level-1 and level-2 NUMA domains is:

$$\begin{aligned} p_{1\oplus 2} =& Pr[passing\ within\ level\ 1]p_1 \\ &+(1 - Pr[passing\ within\ level\ 1])p_2 \end{aligned}$$

There are $(n_1 n_2 - 1)$ other threads within level-2 of the NUMA hierarchy, of which $(n_1 - 1)$ are level-1 NUMA peers. Hence, the probability of cohort passing between level-1 NUMA peers is $(n_1 - 1)/(n_1 n_2 - 1)$.

$$\implies p_{1\oplus 2} = \frac{n_1 - 1}{n_1 n_2 - 1} p_1 + \frac{n_1 n_2 - n_1}{n_1 n_2 - 1} p_2 \tag{7}$$

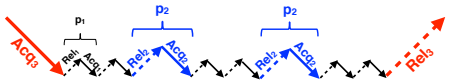



**Figure 6.** Lock passing in the HMCS⟨3⟩ lock.

The C-MCS$_{\text{out}}$ lock passes the lock $(c_{out} - 1)$ times within its cohorts adding a total of $(c_{out} - 1)p_{1\oplus 2}$ length to the critical path. $c_{out}$ lock acquisitions acquire time $p_3 + (c_{out} - 1)p_{1\oplus 2}$. The cycle of remote acquisition at level-3, local passing within either level-1 or level-2 and remote release to level-3 repeats in each domain. Hence throughput of the C-MCS$_{\text{out}}$ lock, $\mathcal{T}_{out}$, in a 3-level system is given by:

$$\mathcal{T}_{out}(3) = \frac{c_{out}}{p_3 + (c_{out} - 1)p_{1\oplus 2}} \tag{8}$$

As $c_{out} \to \infty$, $\mathcal{T}_{out} \to 1/p_{1\oplus 2}$. Maximum throughput of a C-MCS$_{\text{out}}$ lock, $\mathcal{T}^{max}_{out}$, in a 3-level system is given by:

$$\mathcal{T}^{max}_{out}(3) = \frac{1}{p_{1\oplus 2}} \tag{9}$$

This bound holds for any N-level system. In general, if the innermost cohort formation is at level $k$, the peak throughput is $1/p_{1\oplus 2\oplus \ldots \oplus k}$.

Since $p_2$ (the lock passing time at level 2) is higher than $p_1$ (the lock passing time at level 1), $p_{1\oplus 2}$ (the expected lock passing time between level-1 or level-2) is higher than $p_1$. From Eqn 6 and 9, it follows that $\mathcal{T}^{max}_{out}(3) < \mathcal{T}^{max}_{in}(3)$, Hence, *C-MCS$_{in}$ achieves higher peak throughput than C-MCS$_{out}$*. Experiments on an IBM Power 755, a 3-level system, corroborate this finding (Sec. 6).

#### 4.1.3 Throughput of HMCS Locks

The HMCS⟨3⟩ lock forms cohorts both at level 1 and level 2 of the NUMA hierarchy in a 3-level system. Let $h_1$ and $h_2$ respectively be the passing thresholds at level 1 and level 2. In a 3-level lock, a full cycle begins by obtaining the lock at level-3 ($Acq_3$). Subsequently, the lock is passed $(h_1 - 1)$ times within level-1, releasing the lock at level-2 ($Rel_2$), followed by an acquisition at level-2 ($Acq_2$). The cycle of passing within level-2 happens for $(h_2 - 1)$ times. On reaching the passing threshold at level 2, the HMCS⟨3⟩ lock relinquishes the global lock to a level-3 peer adding $Rel_3$ to the critical path. There will be a total of $(h_1 - 1)h_2$ lock passes within level 1, each one adding $p_1$ length to the critical path. There will be $(h_2 - 1)$ lock passes at level 2, each one adding $\hat{p}_2$ length to the critical path. There will be one lock passing at level 3 adding $\hat{p}_3$ length to the critical path. In the entire cycle, $h_1 h_2$ locks will be acquired. Hence throughput of the HMCS⟨3⟩ lock, $\mathcal{T}_{hmcs\langle 3\rangle}$, in a 3-level system is given by:

$$\mathcal{T}_{hmcs\langle 3\rangle}(3) = \frac{h_1 h_2}{\hat{p}_3 + (h_2 - 1)\hat{p}_2 + h_2(h_1 - 1)p_1} \tag{10}$$

$\hat{p}_3$ and $\hat{p}_2$ are the lock passing times respectively at level-2 and level-3 in the HMCS⟨3⟩ lock. The level-2 lock passing in the HMCS⟨3⟩ lock has an additional critical path component that loads and compares the status flag (Lines 25-27 in Listing 1) with the $threshold$, which is not needed by optimal implementations of C-MCS locks. If this additional cost is $\epsilon_0$, then $\hat{p}_2 = p_2 + \epsilon_0$. On reaching the passing threshold at level 2, the HMCS⟨3⟩ lock needs to traverse to the parent lock before releasing the global lock to a level-3 peer (Line 29 in Listing 1)—a cost not incurred in C-MCS locks. Let $\epsilon_1$ be the cost of traversing from level-2 lock to the level-3 parent lock (all necessary datum will be in the nearest level cache). For that reason, $\hat{p}_3 = p_3 + \epsilon_1$. Hence, the throughput of the HMCS⟨3⟩ lock is:

$$\mathcal{T}_{hmcs\langle 3\rangle}(3) = \frac{h_1 h_2}{p_3 + (h_2 - 1)p_2 + h_2(h_1 - 1)p_1 + \epsilon} \tag{11}$$

where $\epsilon = \epsilon_1 + (h_2 - 1)\epsilon_0$.

As $h_1 \to \infty$, $\mathcal{T}_{hmcs\langle 3\rangle} \to 1/p_1$. Maximum throughput of the HMCS⟨3⟩ lock, $\mathcal{T}^{max}_{hmcs\langle 3\rangle}$, in a 3-level system is given by:

$$\mathcal{T}^{max}_{hmcs\langle 3\rangle}(3) = \frac{1}{p_1} \tag{12}$$

It is straightforward to infer from Eqn 6, 9 and 12 that:

$$\mathcal{T}^{max}_{in}(3) = \mathcal{T}^{max}_{hmcs\langle 3\rangle}(3) > \mathcal{T}^{max}_{out}(3) \tag{13}$$

Hence, *the C-MCS$_{in}$ lock—a C-MCS variant with inner cohorting and the HMCS⟨3⟩ lock have the same peak throughput and both of them can deliver higher throughput than Dice et al.'s C-MCS lock with outer cohorting.*

From Eqn 10, it is straight forward to generalize the throughput of an N-level HMCS lock, $\mathcal{T}_{hmcs\langle N\rangle}$, for an N-level NUMA system as:

$$\mathcal{T}_{hmcs\langle N\rangle}(N) = \frac{\prod_{i=1}^{N-1} h_i}{p_N + \sum_{i=1}^{N-1}\left(p_i(h_i - 1)\prod_{j=i+1}^{N-1} h_j\right)} \tag{14}$$

***Discussion:*** While our performance models assumed empty critical sections for assessing lock overhead, we can model non-empty critical sections by including an additional $cs$ term to account for the cost of the CS in each lock passing term ($p_i$). While the C-MCS$_{\text{in}}$ and HMCS locks can theoretically deliver top throughput, in practice, threads may not be able to achieve the maximum possible throughput when the delay outside the critical path is large. The average number of threads enqueuing for the same lock acquisition at a level per unit time is defined as the *arrival rate* at that level. For a lock representing a NUMA domain in the HMCS tree to be useful, the arrival rate for that domain must exceed the rate at which the domain is selected for service. Furthermore, the reduction in access latency realized by passing a lock and the data it protects within a domain should outweigh the cost of adding an addition lock level.

### 4.2 (Un)Fairness

If a lock serves threads in the first-in first-out (FIFO) order, then the lock is considered fully fair. The MCS lock ensures FIFO ordering once a requesting thread has swapped the lock's tail pointer. HMCS and C-MCS locks do not ensure FIFO ordering since they allow multiple turns for threads within the same domain even when threads are already enqueued elsewhere in the lock hierarchy.

We devise a metric of *unfairness* to compare HMCS and C-MCS locks. In this scheme, *each subdomain can acquire the lock as many times as the total number of threads in its subdomain; every additional acquisitions counts towards a unit of unfairness.* Under high contention, a thread that just released the lock must wait until all other threads already waiting at that level are serviced, which is ensured by the FIFO property of MCS lock at each level. Hence, under high contention, from the viewpoint of a waiting thread, this penalizes a threader for multiple acquisitions while the lock is passed within a domain, but does not penalize for not maintaining the system-wide FIFO order.

**Definition 3** (Unfairness:)**.** Unfairness *is the maximum possible total number of additional rounds of lock acquisitions over the designated quota of one, by other threads in the system when a thread is still waiting to acquire the lock. Formally, if $T$ is the set of all threads in the system, the unfairness, $\mathcal{U}$, is given by:*
$\mathcal{U} = Max\Big(\forall j \in T, \sum_{i \in T - j}\big(\textit{ acquisitions by } i \mid j \textit{ is waiting} - 1\big)\Big)$.

The worst case wait for a thread when using an MCS lock is $(\prod_{i=1}^{N} n_i - 1)$ lock acquisitions and its unfairness is $0$. In fully contended cohort locks, if the threshold of each level $h_i$ equals to

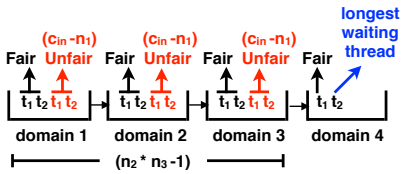



**Figure 7.** Unfairness in the C-MCS$_{in}$ lock, when $c_{in} \geq n_1$.

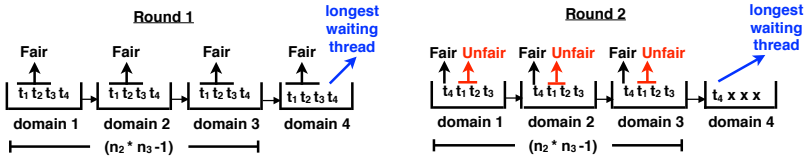



**Figure 8.** Unfairness in the C-MCS$_{in}$ lock, when $c_{in} < n_1$.

the number of contenders $n_i$ at level $i$, then the longest waiting thread waits for a maximum of $(\prod_{i=1}^{N} n_i - 1)$ lock acquisition, which is same as the longest wait in the classical MCS lock. However, if any $h_i > n_i$, threads will be served more than once causing unfairness for some waiting threads in the system.

### 4.2.1 Unfairness of the C-MCS$_{in}$ Lock

Consider the case when $c_{in} \geq n_1$. The maximum unfairness will be incurred by the last thread—shown in blue color in Figure 7, enqueued in the last domain $D_{last}$ enqueued in the outer queue of the C-MCS$_{in}$ lock. Each of the first $(n_2 n_3 - 1)$ NUMA domains acquires $c_{in}$ locks of which only $n_1$ are fair and the remaining $(c_{in} - n_1)$ are unfair. The acquisitions in the $D_{last}$ domain don't contribute to the unfairness since by the time repetitions happen, the longest waiting thread would have already been serviced. Hence the unfairness of the C-MCS$_{in}$ lock in a 3-level system is:

$$\mathcal{U}_{in}(3) = (c_{in} - n_1)(n_2 n_3 - 1) \tag{15}$$

In Figure 7, $n_1 = 2, n_2 = 2, n_3 = 2$ and $c_{in} = 4$. Total unfairness is $(4 - 2)(2 \times 2 - 1) = 6$.

When $c_{in} < n_1$, each domain goes through multiple rounds of service before the longest waiting thread in the last domain can be serviced. We demonstrate this with the example in Figure 8, where $n_1 = 4, n_2 = 2, n_3 = 2$, and $c_{in} = 3$. Only 3 out of 4 threads will be served in each domain in the the first round through all domains. In the worst case, it takes a total of two rounds through all domains before the longest waiting thread can be serviced. In the second round through all the domains, the first thread $(t_4)$ in domains 1-3 will be serviced for the first time, but the remaining 2 threads ($t_1$ and $t_2$) will be taking their additional round adding to two units of unfairness. As before, no repetitions happen in the $D_{last}$ domain. Each domain gets lock acquisitions in $c_{in}$ quantum; from the point of view of longest waiting thread, each of the other $(n_2 n_3 - 1)$ domains acquire $\lceil \frac{n_1}{c_{in}} \rceil c_{in}$ locks of which $n_1$ are fair. Hence, the total unfairness is:

$$\mathcal{U}_{in}(3) = \left( \left\lceil \frac{n_1}{c_{in}} \right\rceil c_{in} - n_1 \right)(n_2 n_3 - 1) \tag{16}$$

When $c_{in} \geq n_1$, Eqn 16 simply reduces into Eqn 15. In Figure 8, the unfairness is $(\lceil 4/3 \rceil 3 - 4)(2 \times 2 - 1) = 6$.

### 4.2.2 Unfairness of the C-MCS$_{out}$ Lock

The C-MCS$_{out}$ lock collapses level-1 and level-2 of the NUMA hierarchy into a single domain of $n_1 n_2$ threads and assumes that there are $n_3$ such domains. Following the same argument as before, it is straight forward to show that the unfairness of the C-MCS$_{out}$ lock in a 3-level system is:

$$\mathcal{U}_{out}(3) = \left( \left\lceil \frac{n_1 n_2}{c_{out}} \right\rceil c_{out} - n_1 n_2 \right)(n_3 - 1) \tag{17}$$

### 4.2.3 Unfairness of HMCS Locks

First we compute the unfairness when, $h_1 \geq n_1$ and $h_2 \geq n_2$. Consider a system where $n_1 = 2, n_2 = 2$ and let $h_1 = 4, h_2 = 4$ as shown in Figure 9. Maximum unfairness will be observed by the last enqueued thread in the last level-1 domain ($D_{last_{L1}}$) belonging to the last enqueued level-2 domain ($D_{last_{L2}}$). There will

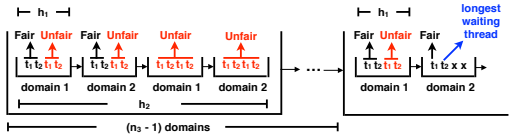



**Figure 9.** Unfairness in the HMCS$\langle 3 \rangle$ lock, when $h_1 \geq n_1$ and $h_2 \geq n_2$.

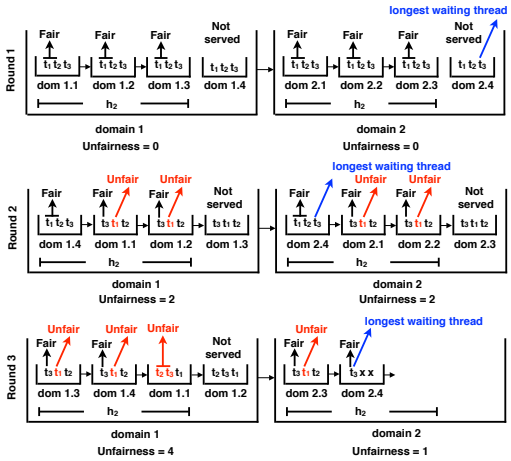



**Figure 10.** Unfairness in the HMCS$\langle 3 \rangle$ lock, when $h_1 < n_1$ and $h_2 < n_2$.

be $h_1 h_2$ lock acquisitions inside each of the first $(n_3 - 1)$ level-2 domains of which $(h_1 h_2 - n_1 n_2)$ are unfair. Inside the $D_{last_{L2}}$ domain, there will be $(h_1 - n_1)$ unfair acquisitions in each of the first $(n_2 - 1)$ level-1 domains. The $D_{last_{L1}}$ domain adds no unfairness since any repetitions will happen only after the longest-waiting thread is already serviced. Thus the total unfairness of the HMCS$\langle 3 \rangle$ lock in a 3-level system when $h_1 \geq n_1$ and $h_2 \geq n_2$ is:

$$\mathcal{U}_{hmcs\langle 3 \rangle}(3) = (h_1 h_2 - n_1 n_2)(n_3 - 1) + (h_1 - n_1)(n_2 - 1) \tag{18}$$

We use Figure 10 to provide intuition for deriving unfairness when $h_1 < n_1$ and $h_2 < n_2$. In Figure 10, $n_1 = 3$, $n_2 = 4$, $n_3 = 2$, $h_1 = 2$, and $h_2 = 3$. In the first round, three out of four level-1 domains will be served in each of the level-2 domains. Within each serviced level-1 domains, two out of three threads will be serviced. Round 1 accrues no unfairness. In round 2, two threads inside the domain 1.4 will be served for the first time; In domains $1.1$ and $1.2$ one thread $(t_3)$ will be served for the first time, whereas one thread $(t_1)$ will be enjoying its second round of service, while the longest waiting thread is still waiting. This accumulates an unfairness of 2 units in domains 1. The same repeats in domain-2 also. In the third round, one thread $(t_3)$ in the domains $1.3$ and $1.4$ will be served fairly, whereas one thread $(t_1)$ will be served for the second time, hence unfair. Furthermore, due to a remaining one round in the $h_2$ quantum, we serve two threads ($t_2$ and $t_3$) in the domain $1.1$ for the $3^{rd}$ time accruing a total of 4 units of unfairness in domain 1. In domain $2.3$, one thread $(t_3)$ is served fairly, whereas one thread $(t_1)$ will be served for the second

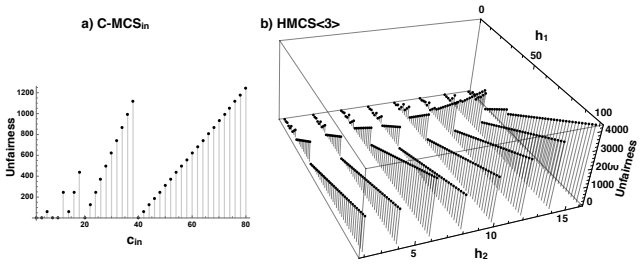



**Figure 11.** Impact of threshold on unfairness.

time. Thus total unfairness in round 3 is 5. The total unfairness in all rounds taken together is 9.

Since each level-2 lock acquisition provides a chunk of $h_1$ locks at a time, each level-1 domains has to experience $\lceil \frac{n_1}{h_1} \rceil$ level-2 lock acquisitions before the longest-waiting thread is served. Let us first focus on a level-2 domain to which the longest waiting thread does not belong to arrive at the number of level-3 acquisitions that need to happen. Since there are $n_2$ peers at level-2, a total of $\lceil \frac{n_1}{h_1} \rceil n_2$ level-2 lock acquisitions need to happen. Level-2 locks are given in a chunk of $h_2$, which means each level-2 domain needs to experience $\lceil \lceil \frac{n_1}{h_1} \rceil \frac{n_2}{h_2} \rceil$ level-3 lock acquisitions. Each round of level-3 lock acquisition serves a total of $h_1 h_2$ locks inside its subdomain. Hence, the total lock acquisitions will be $\lceil \lceil \frac{n_1}{h_1} \rceil \frac{n_2}{h_2} \rceil h_1 h_2$, of which only $n_1 n_2$ are fair. Hence, the unfairness in each domain to which the longest waiting thread does not belong is $\lceil \lceil \frac{n_1}{h_1} \rceil \frac{n_2}{h_2} \rceil h_1 h_2 - n_1 n_2$. There are a maximum of $(n_3 - 1)$ such domains.

In the level-2 domain to which the longest-waiting thread belongs, each of the $(n_2 - 1)$ domains need to experience $\lceil \frac{n_1}{h_1} \rceil h_1$ lock acquisitions, of which only $n_1$ will be fair. Thus, the total unfairness in the last level-2 domain will be $(\lceil \frac{n_1}{h_1} \rceil h_1 - n_1)(n_2 - 1)$. Combining the unfairness from both cases, we arrive at the total unfairness of the HMCS$\langle 3 \rangle$ lock in a 3-level system as:

$$\begin{aligned} \mathcal{U}_{hmcs\langle 3 \rangle}(3) = &\left( \left\lceil \left\lceil \frac{n_1}{h_1} \right\rceil \frac{n_2}{h_2} \right\rceil h_1 h_2 - n_1 n_2 \right)(n_3 - 1) \\ &+ \left( \left\lceil \frac{n_1}{h_1} \right\rceil h_1 - n_1 \right)(n_2 - 1) \end{aligned} \tag{19}$$

For the example in Figure 10, the unfairness is:

$$(\lceil \lceil 3/2 \rceil 4/3 \rceil 2 \times 3 - 3 \times 4)(2-1) + (\lceil 3/2 \rceil 2 - 3)(4-1) = 9.$$

When $h_1 \geq n_1$ and $h_2 \geq n_2$, Eqn 19 degenerates into Eqn 18. Eqn 18 also covers the cases of $h_1 \geq n_1$ and $h_2 < n_2$, as well as $h_1 < n_1$ and $h_2 \geq n_2$. We omit the details for brevity. We note that when $h_1 < n_1$ unfairness is $0$, if either $h_1$ divides $n_1$ and $h_2$ divides $n_2$ or $h_2$ divides $\lceil \frac{n_1}{h_1} \rceil n_2$.

Figure 11 provides visualization of unfairness in the C-MCS$_{\text{in}}$ and HMCS$\langle 3 \rangle$ locks with varying values of thresholds. Both graphs assume a hypothetical machine where $n_1 = 40, n_2 = 8$, and $n_3 = 4$. For the C-MCS$_{\text{in}}$ lock, the unfairness is $0$, when the threshold $c_{in}$ divides $n_1$. For the HMCS$\langle 3 \rangle$ lock, the unfairness is 0, when the threshold $h_1$ divides $n_1$ and $h_2$ divides $n_2$ and grows linearly with both increase in $h_1$ and $h_2$. Finally, we extend the Eqn 19 to generalize the unfairness of an N-level HMCS lock, $\mathcal{U}_{hmcs\langle N \rangle}$, for any N-level NUMA system as:

$$\mathcal{U}_{hmcs\langle N \rangle}(N) = \sum_{i=1}^{N-1} \left( \psi_i \prod_{j=1}^{i} h_i - \prod_{j=1}^{i} n_i \right)(n_{i+1} - 1) \tag{20}$$

$$\text{where } \psi_i = \left\lceil \left\lceil \left\lceil \frac{n_1}{h_1} \right\rceil \frac{n_2}{h_2} \right\rceil \ldots \frac{n_i}{h_i} \right\rceil$$

When $h_i \geq n_i, \forall i$, the $\psi_i$ term in Eqn 20 vanishes.

# 5. HMCS Properties

In this section we prove some of the important attributes of HMCS locks in comparison with the C-MCS locks. From Eqn 13 the peak throughput of the HMCS lock is same as the peak throughput of the C-MCS$_{\text{in}}$ lock. Hence the competition between them is for fairness when delivering the same throughput. We prove, in Section 5.1, that on a 3-level NUMA system, when the difference in latencies between two consecutive levels of the NUMA hierarchy is sufficiently large, a 3-level HMCS lock delivers higher fairness than a C-MCS$_{\text{in}}$ lock for any user-chosen throughput.

The competition between the C-MCS$_{\text{out}}$ and HMCS$\langle 3 \rangle$ locks is for the throughput at the same level of fairness. In this regard, in Section 5.2, we prove that on a 3-level system, when the difference in latencies between two consecutive levels of NUMA hierarchy is sufficiently large, a 3-level HMCS lock delvers higher throughput than the C-MCS$_{\text{out}}$ lock for any user-chosen fairness level.

## 5.1 Fairness Assurance of HMCS Over C-MCS$_{\text{in}}$

Let $\alpha \in [0 - 1]$ be the user chosen level of throughput represented as a fraction of peak throughput. If the value of $c_{in}$ in the C-MCS$_{\text{in}}$ lock needed to achieve this throughput is less than $n_1$, then, the HMCS$\langle 3 \rangle$ lock can set $h_1 = n_1$ and $h_2 = n_2$, which not only achieves more throughput but also delivers 0 unfairness, thus proving its superiority. Hence the comparison is needed only when $c_{in} \geq n_1$. The value of $c_{in}$ to achieve the expected throughput is derived by solving Eqn 5:

$$\frac{c_{in}}{p_{2 \oplus 3} + (c_{in} - 1)p_1} = \alpha \, \mathcal{T}_{in}^{max}(3) \tag{21}$$

$$\implies c_{in} = \frac{\alpha(p_{2 \oplus 3} - p_1)}{p_1(1 - \alpha)} \tag{22}$$

In the HMCS$\langle 3 \rangle$ lock we set $h_2 = n_2$. The value of $h_1$ to achieve the expected throughput $\alpha$ is derived by solving Eqn 11.

$$\frac{h_1 n_2}{p_3 + (n_2 - 1)p_2 + n_2(h_1 - 1)p_1 + \epsilon} = \alpha \, \mathcal{T}_{hmcs\langle 3 \rangle}^{max}(3) \tag{23}$$

$$\implies h_1 = \frac{\alpha(p_3 + p_2(n_2 - 1) - p_1 n_2 + \epsilon)}{n_2 p_1(1 - \alpha)} \tag{24}$$

Again, only values of $h_1 \geq n_1$ are of interests, since smaller values can be replaced with $h_1 = n_1$ to obtain $0$ unfairness but superior throughput. We note in passing that practical comparisons can only be made when the thresholds in both Eqn 22 and 24 take integer values. Substituting $c_{in}$ from Eqn 22 in Eqn 15, we get:

$$\mathcal{U}_{in}(3) = \left( \frac{\alpha(p_{2 \oplus 3} - p_1)}{p_1(1 - \alpha)} - n_1 \right)(n_2 n_3 - 1) \tag{25}$$

Substituting $h_2 = n_2$ and $h_1$ from Eqn 24 in Eqn 18, we get:

$$\mathcal{U}_{hmcs\langle 3 \rangle}(3) = \left( \frac{\alpha(p_3 + p_2(n_2 - 1) - p_1 n_2 + \epsilon)}{n_2 p_1(1 - \alpha)} - n_1 \right)(n_2 n_3 - 1) \tag{26}$$

From Eqn 25 and 26, the necessary condition for the HMCS$\langle 3 \rangle$ lock to deliver higher fairness than the C-MCS$_{\text{in}}$ lock is:

$$Eqn\ 25 \geq Eqn\ 26 \tag{27}$$

$$\implies p_{2 \oplus 3} - p_1 \geq \frac{p_3 + p_2(n_2 - 1) - p_1 n_2 + \epsilon}{n_2} \tag{28}$$

$$\implies (p_3 - p_2)\left(1 - \frac{n_2}{n_2 n_3 - 1}\right) \geq \epsilon_o + \frac{\epsilon_1}{n_2 - 1} \tag{29}$$

All terms on the LHS and RHS are positive in Eqn 29. On any NUMA system, $n_1, n_2, n_3 \geq 2$; otherwise it does not form a new NUMA domain at a level. It can be shown that the minimum value of the second term on LHS is $1/3$. Hence, the sufficiency condition for the HMCS$\langle 3 \rangle$ lock to deliver higher fairness than the C-MCS$_{\text{in}}$

lock at the same throughput is:

$$\frac{(p_3 - p_2)}{3} \geq \epsilon_o + \frac{\epsilon_1}{n_2 - 1} \tag{30}$$

From Eqn 29 and Eqn 30 we make the following observations:

1. As long as the cost of additional check of status flag and amortized cost of traversing to parent level lock once in $h_2 = n_2$ quantum is less than one third the difference in passing time between level-3 and level-2, it calls for having an additional level in the HMCS lock hierarchy.
2. Larger difference in passing time (also related to the access latency) between two consecutive levels in the NUMA hierarchy (first term in LHS of Eqn 29) governs the necessity for additional level in the HMCS lock hierarchy.
3. From the second term in LHS of Eqn 29, it follows that if a NUMA domain $d$ has many subdomains, then one can benefit from having an HMCS lock for an NUMA subdomain of $d$.

### 5.2 Throughput Assurance of HMCS Over C-MCS$_{\text{out}}$

Let $k$ be the threshold value of the C-MCS$_{\text{out}}$ lock that achieves the user chosen unfairness value of $\beta$. Then from Eqn 17, we know that

$$\mathcal{U}_{out}(3) = \left(\left\lceil \frac{n_1 n_2}{k} \right\rceil k - n_1 n_2\right)(n_3 - 1) \tag{31}$$

We set $h_1 = n_1$ and $h_2 = k/n_1$ in the HMCS$\langle 3\rangle$ lock. The concern here is whether a fractional $h_2$ value is possible? The answer is yes. For example, assume $n_1 = 4$, $n_2 = 3$, and $k = 13$. We want to set the following configuration: $h_1 = n_1 = 4$, $h_2 = 13/4 = 3.25$. The HMCS$\langle 3\rangle$ lock can hand out 3 rounds of level-2 quanta each containing $h_1 (= 4)$ lock acquisitions, totaling 3 × 4 = 12 acquisitions. But in the $4^{th}$ round, the level-2 lock should curtail level-1's threshold from $n_1 = 4$ to $0.25 \times n_1 = 0.25 \times 4 = 1$. Notice that $0.25 \times 4 = 1 = 13 \bmod 4 = k \bmod n_1$. With this, we would have given out $12 + 1 = 13 = k$ acquisitions with exactly the same amount of unfairness as the C-MCS$_{\text{out}}$ lock. It is no accident that for the last round we chose $k \bmod n_1$ threshold; it simply follows from algebra that $\lfloor \frac{k}{n_1} \rfloor n_1 + k \bmod n_1 = k$.

Intuitively, since level-1 locks go in quantum of $n_1$, we want to curtail the last quantum to a smaller value, i.e., $k \bmod n_1$. With this knowledge, we can slightly alter the acquire protocol. Instead of starting the count from 1 till $n_1$ to reach the threshold, we start the counter (the `status` field of the `QNode`) from $(k - k \bmod n_1)$ for the last round of level-1 acquisitions when there is need to achieve the fractional $h_2$ value. This adds one extra compare on the critical path, but we note that this modification is only of theoretical interest to demonstrate that the fractional $h_2$ value is achievable.
Substituting $h_1 = n_1$ and $h_2 = k/n_1$ in Eqn 19 yields:

$$\begin{aligned}\mathcal{U}_{hmcs}(3) =& \left(\left\lceil \left\lceil \frac{n_1}{n_1} \right\rceil \frac{n_1 n_2}{k} \right\rceil n_1 \frac{k}{n_1} - n_1 n_2\right)(n_3 - 1) \\ &+ \left(\left\lceil \frac{n_1}{n_1} \right\rceil n_1 - n_1\right)(n_2 - 1) \\ =& \left(\left\lceil \frac{n_1 n_2}{k} \right\rceil k - n_1 n_2\right)(n_3 - 1) = Eqn\ 31\end{aligned}$$

Thus, when $h_1 = n_1$ and $h_2 = k/n_1$, C-MCS$_{\text{out}}$ and HMCS$\langle 3\rangle$ locks deliver the same fairness.

Now, we derive the conditions for $\mathcal{T}_{hmcs\langle 3\rangle}(3)$ to be greater than $\mathcal{T}_{out}(3)$ in this configuration. Substituting $c_{out} = k$ in the throughput equation for the C-MCS$_{\text{out}}$ lock (Eqn 8), we get:

$$\mathcal{T}_{out}(3) = \frac{k}{p_3 + (k-1)p_{1\oplus 2}} \tag{32}$$

Substituting $h_1 = n_1$ and $h_2 = k/n_1$, in the throughput equation for the HMCS$\langle 3\rangle$ lock (Eqn 11), we get:

$$\mathcal{T}_{hmcs\langle 3\rangle}(3) = \frac{n_1 \frac{k}{n_1}}{p_3 + (\frac{k}{n_1} - 1)p_2 + \frac{k}{n_1}(n_1 - 1)p_1 + \epsilon} \tag{33}$$

From Eqn 32 and 33, to show that $\mathcal{T}_{hmcs}(3) \geq \mathcal{T}_{out}(3)$, we need to simply show that:

$$(k-1)p_{1\oplus 2} \geq (\frac{k}{n_1} - 1)p_2 + \frac{k}{n_1}(n_1 - 1)p_1 + \epsilon \tag{34}$$

Substituting for $p_{1\oplus 2}$ from Eqn 7 in Eqn 34, we obtain:

$$\begin{aligned}(p_2 - p_1)\Bigg(&\frac{k(n_1 n_2 - n_1 - n_2) + (n_1 - 1)}{(n_1 n_2 - 1)(\frac{k}{n_1} - 1)} \\ &+ \frac{k}{n_1(n_1 n_2 - 1)(\frac{k}{n_1} - 1)}\Bigg) \geq \epsilon_0 + \frac{\epsilon_1}{\frac{k}{n_1} - 1}\end{aligned} \tag{35}$$

Eqn 35 forms the basic constraint to ensure that the HMCS$\langle 3\rangle$ lock has higher throughput compared to the C-MCS$_{\text{out}}$ lock. Each term on the LHS and RHS of Eqn 35 is positive. When $k$ increases, the LHS increases and RHS decreases, ensuring that beyond a certain value of $k$, the inequality is always true. Hence, we need to find the sufficiency condition at the smallest value that $k$ can assume. Any value of $k < n_1 n_2$ increases unfairness in both locks, and decreases the throughput. Hence if the C-MCS$_{\text{out}}$ chooses $k < n_1 n_2$, for the HMCS$\langle 3\rangle$ lock we simply choose $h_1 = n_1, h_2 = n_2$, which guarantees no unfairness and yet delivers higher throughput than the C-MCS$_{\text{out}}$ lock. Hence the smallest value that a C-MCS$_{\text{out}}$ lock can choose for $k$ is $n_1 n_2$. Substituting $k = n_1 n_2$ in Eqn 35, we arrive at

$$\begin{aligned}(p_2 - p_1)\Bigg(&\frac{n_1 n_2(n_1 n_2 - n_1 - n_2) + (n_1 - 1) + n_2}{(n_1 n_2 - 1)(n_2 - 1)}\Bigg) \\ &\geq \epsilon_0 + \frac{\epsilon_1}{n_2 - 1}\end{aligned} \tag{36}$$

On any system, $n_1, n_2, n_3 \geq 2$, otherwise it does not form a new NUMA domain at that level. It can be shown that the minimum value of the second term on LHS in Eqn 36 is 3. Hence the sufficiency condition for the HMCS$\langle 3\rangle$ lock to deliver higher throughput than the C-MCS$_{\text{out}}$ lock at the same fairness level is:

$$(p_2 - p_1)(3) \geq \epsilon_0 + \frac{\epsilon_1}{n_2 - 1} \tag{37}$$

From Eqn 35, 36, and 37 we make following observations:

1. There is always a passing threshold value in the C-MCS$_{\text{out}}$ lock beyond which the HMCS$\langle 3\rangle$ lock is always guaranteed to deliver higher throughput at the same fairness. This also follows from the fact that $\mathcal{T}^{max}_{out}(3) < \mathcal{T}^{max}_{hmcs\langle 3\rangle}(3)$.
2. When the first condition is not met, as long as the cost of additional check of the status flag and amortized cost of traversing to parent level lock once in $h_1 = n_1$ quantum is less than three times the difference in passing time between level-1 and level-2, it is beneficial to have additional level in the lock hierarchy. The RHS is, typically, significantly less than the minimum possible LHS.

***Discussion:*** It is straight forward to show that an N-level HMCS lock offers the same throughput and fairness superiority guarantees over an (N-1)-level HMCS lock. The argument follows similar to the aforementioned derivations; we omit the details for brevity.

## 6. Experimental Evaluation

We conducted our experimental evaluation on two platforms—an IBM Power 755 and an SGI UV 1000. In all our experiments, we densely packed OS threads and bound them to HW threads.

**Table 1.** Experimental setup.

| | IBM Power 755 | SGU UV 1000 |
|---|---|---|
| Processor | POWER7 @ 3.86 GHz | Intel Xeon X7560 @ 2.27 GHz |
| SMT | 4-way | 2-way |
| Cores/Socket | 8 | 8 |
| Sockets/node | 4 | 2 |
| L1-D-Cache | 32KB private | 32KB private |
| L2-Cache | 256KB private (L1 inclusive) | 256KB private (L1 inclusive) |
| L3-Cache | 8 × 4MB victim | 24MB shared (L2 inclusive) |
| Memory controller | 2 on-chip, 4-channel DDR3 | 2 on-chip, 2-channel DDR3 |
| Compiler | `xlc++ v1.4.3` | `icc v.14.0.0` |

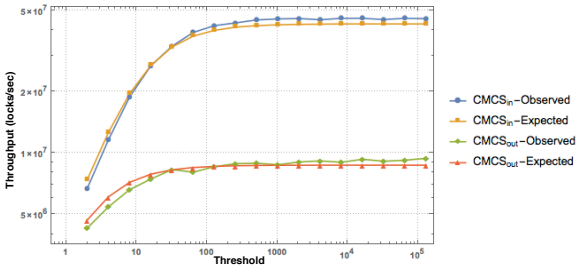



**Figure 12.** Expected vs. observed throughput of the C-MCS$_{in}$ and C-MCS$_{out}$ locks on IBM Power 755 with 128 threads.

**Table 2.** Difference in expected vs. observed throughput on IBM Power 755.

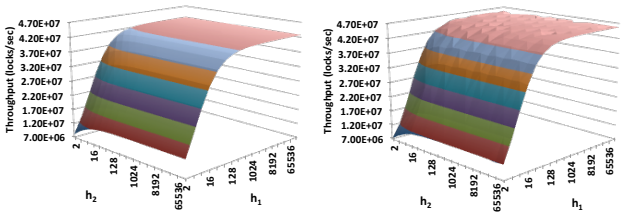
| Lock | Median Difference | Maximum Difference |
|---|---|---|
| HMCS⟨3⟩ | 6.3% | 15% |
| C-MCS$_{in}$ | 5.6% | 10% |
| C-MCS$_{out}$ | 4.7% | 11% |

### 6.1 Evaluation on IBM Power 755

The IBM Power 755, a 3-level system, used for our experiments has the specifications shown in Table 1. This system provides a total of 128 hardware threads. SMTs sharing an L1 and L2 cache form level-1 of the NUMA hierarchy. All cores on the same socket form level-2 of the NUMA hierarchy. Four sockets sharing the primary memory form level-3 of the NUMA hierarchy.

***Accuracy of analytical models:*** We inspected the compiler-generated assembly instructions appearing on the critical path and statically computed the number of CPU cycles. By knowing the passing time at each level of the hierarchy, we computed the expected throughput values at various passing thresholds. To assess the accuracy of our analytical models, we compared our model-derived throughput with the empirically observed throughput at 128 threads. Figure 12 plots the graph of expected vs. observed throughput of C-MCS locks. Figure 13(a) and 13(b) respectively plot the expected vs. empirically observed throughput of the HMCS⟨3⟩ lock. The difference in expected vs. observed throughput is small (see Table 2). It is clear that our analytical models accurately predict the performance at each value of lock passing threshold. As expected, the peak throughputs of the HMCS⟨3⟩ and C-MCS$_{in}$ locks are same ($4.58E$+07 acquisitions / second) and $4.8\times$ higher than the C-MCS$_{out}$ lock. Beyond a certain point, further increases in threshold yield no significant increase in the throughput of a lock. Significant performance gains happen in the HMCS⟨3⟩ lock by increasing the $h_1$ threshold. Table 3 shows the difference in unfairness of the C-MCS$_{in}$ lock and the HMCS⟨3⟩ lock to reach a given target throughput. The HMCS⟨3⟩ lock delivers up to 7x higher fairness than the C-MCS$_{in}$ lock. At 99.9% of the peak throughput, the HMCS⟨3⟩ lock delivers $2\times$ higher fairness than the C-MCS$_{in}$ lock.

***Empty CS:*** Figure 14 demonstrates the scalability of various MCS lock variants with empty critical sections. We chose the passing thresholds to deliver 99.9% of the peak throughput for C-MCS and HMCS⟨3⟩ locks. Under no contention, the HMCS⟨3⟩ lock has $2.9\times$ lower throughput than the MCS lock, but under high contention, the HMCS⟨3⟩ lock has $11.7\times$ higher throughput. The throughput of the MCS lock drops each time a new NUMA level is introduced. The C-MCS$_{in}$ lock follows similar high-throughput trend as the HMCS⟨3⟩ lock. The throughput of the C-MCS$_{out}$ drops between 4-8 threads when the SMT-threads diverged into multiple cores. The HMCS⟨3⟩ lock delivers $5.22\times$ higher throughput than the C-MCS$_{out}$ lock. At two processors, the MCS lock's throughput drops. This anomaly is explained in the original paper describing the MCS lock [7].




(a) Expected throughput (b) Observed throughput

**Figure 13.** Expected vs. observed throughput of HMCS⟨3⟩ lock on IBM Power 755 with 128 threads.

**Table 3.** Improvement in fairness of the HMCS⟨3⟩ lock over the C-MCS$_{in}$ lock.

| Percent peak throughput | Unfairness HMCS⟨3⟩ | Unfairness C-MCS$_{in}$ | (HMCS improvement) C-MCS$_{in}$ / HMCS⟨3⟩ |
|---|---|---|---|
| 50% | 24 | 173 | 7.14× |
| 70% | 222 | 568 | 2.56× |
| 90% | 1209 | 2545 | 2.10× |
| 99% | 14543 | 29236 | 2.01× |
| 99.9% | 147880 | 296142 | 2.00× |

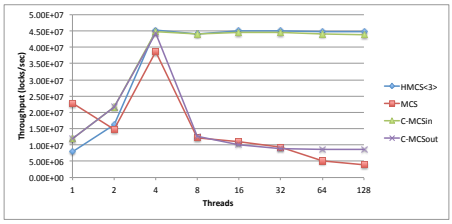



**Figure 14.** Lock scaling with empty CS on IBM Power 755.

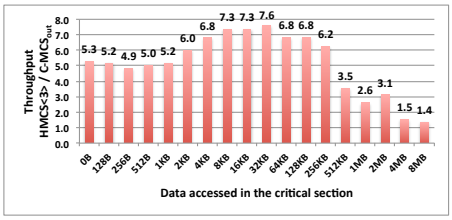



**Figure 15.** Throughput improvement of HMCS⟨3⟩ over C-MCS$_{out}$ with varying size of data accessed in the critical section on IBM Power 755 with 128 threads.

***Non-empty CS:*** We varied the data accessed in the CS from 0 to 8MB, while keeping the number of threads fixed at 128. We chose the passing thresholds to deliver 99.9% of the peak throughput. We compared the throughput of the HMCS⟨3⟩ lock to the C-MCS$_{out}$ lock (Figure 15). The throughput of the HMCS⟨3⟩ lock increases from $5.3\times$ at 0 bytes to $7.6\times$ at 32KB—the L1 cache size.The increase in relative throughput is due to the locality benefits enjoyed by the data accessed in the CS. Beyond the L1-cache size, the ratio decreases, yet continues to be significantly superior (more than $6\times$) until 256KB—the L2 cache size. Beyond

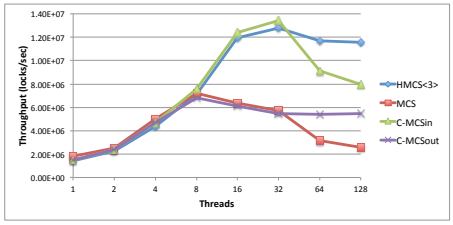



**Figure 16.** Lock scaling at lower contention on IBM Power 755.

the L2-cache size, the benefits of the HMCS⟨3⟩ lock steadily drop from 6.2× to 1.5× at 4MB—the L3 cache size.

***Non-empty CS with lower contention:*** Figure 16 demonstrates the scalability of various MCS lock variants with non-empty critical sections under *lower contention*. Our benchmark touches two cache lines inside the CS and spends a random 0-1.58 $\mu s$ outside the CS to mimic the test setup by Dice et al. [4]. We chose the passing thresholds to deliver 99.9% of the peak throughput for all cohort locks. In this case, the C-MCS$_{in}$ lock's throughput starts to degrade once the lock passing starts to go outside of the socket due to a lack of cohort formation. The HMCS⟨3⟩ lock maintains its high throughput and remains unaffected by increased NUMA levels, whereas throughput of all other locks degrades. At 128 threads, the HMCS⟨3⟩ lock delivers 1.46×, 2.13×, and 4.5× higher throughput than C-MCS$_{in}$, C-MCS$_{out}$, and MCS locks respectively.

***MineBench K-means code:*** K-means is an OpenMP clustering code from the MineBench v.3.0.1 suite—a benchmark suite with full-fledged implementations for data mining workloads [8]. "K-means represents a cluster by the mean value of all objects contained in it. The initial, user provided, $k$ cluster centers are randomly chosen from the database. Then, each object is assigned a nearest cluster center based on a similarity function. Once the new assignments are completed, new centers are found by finding the mean of all the objects in each cluster. This process is repeated until some convergence criteria is met [8]."

Our experiments used an input file with 65K objects and 32 attributes in each point. We set the convergence threshold to $10^{-5}$, and the minimum and maximum number of initial clusters to 2 and 15 respectively. The K-means code uses OpenMP `atomic` directives to update the new clusters centers as shown in Listing 2. Data ping-pongs indiscriminately between remote NUMA domains due to *true sharing* (threads in different NUMA domains modify the same location) and *false sharing* (threads in different NUMA domains modify different locations that share the same cache line). The result is poor scalability—increasing the number of threads increases the running time—as shown in Table 4 column #2.

To address the indiscriminate data movement, we replaced the atomic directives with a coarse-grained lock to protect the cluster centers (see Listing 3). While there exist other parallelization techniques that can deliver higher scalability, our single-lock solution serves to demonstrate the utility of the HMCS lock under high contention on a nontrivial use case. We employed the MCS, C-MCS$_{in}$, C-MCS$_{out}$, and HMCS⟨3⟩ locks as coarse-grained lock implementations. Respective running times are shown in columns #3-#6 in Table 4. For all cohort locks, we used a very high passing threshold since fairness was immaterial. Columns #7-#10 show the improvements of each of the locks, at the same thread settings, when compared to the K-means that used atomic directives.

The NUMA-agnostic coarse-grained MCS lock improved the performance of K-means by up to 14.9× over the atomic operations (Table 4, column #7). Even when there is no false sharing (1 thread), using a coarse-grained lock is superior to using multiple atomic add instructions, since each atomic add on POWER7 turns into a sequence of instructions that includes a *load reserve* and

```
1 /* update new cluster centers : sum of objects located within */
2 #pragma omp atomic
3 new_centers_len[index]++;
4 for (j=0; j<nfeatures; j++)
5 #pragma omp atomic
6   new_centers[index][j] += feature[i][j];
```

**Listing 2.** K-means atomic updates (poor performance).

```
1 /* update new cluster centers : sum of objects located within */
2 Acquire(lock, me);
3 new_centers_len[index]++;
4 for (j=0; j<nfeatures; j++)
5   new_centers[index][j] += feature[i][j];
6 Release(lock, me);
```

**Listing 3.** K-means coarse-grained locking (better performance).

a *store conditional* resulting in higher instruction count. When contention rises, a thread performing a *load reserve* frequently loses reservation to some other thread, resulting in high overhead.

The HMCS⟨3⟩ lock delivers 22.9× higher performance at 32 threads compared to the original execution that used atomic operations. This is 9.2× speedup compared to the original serial execution. Furthermore, beyond 16 threads, the HMCS⟨3⟩ lock's performance is better than all other locks in the same thread settings.

We compare the performance of the HMCS⟨3⟩ lock with the C-MCS$_{in}$, C-MCS$_{out}$ locks in columns #11-#12 respectively. The C-MCS$_{in}$ and HMCS⟨3⟩ locks start to show their superior performance at 8 threads, since they have SMT-level lock passing. While the MCS and C-MCS$_{out}$ locks degrade in performance beyond 8-threads, the C-MCS$_{in}$ and HMCS⟨3⟩ locks continue to improve untill 32 threads. At 32 threads, the C-MCS$_{in}$ lock degrades steeply since it does not pass locks among cores of the same socket. Due to lock passing at both SMT and core levels, the HMCS⟨3⟩ lock demonstrates superior performance over all other locks as the contention rises between 16-128 threads. The performance of the HMCS⟨3⟩ lock degrades at 64 and 128 threads compared to its own performance at 32 threads, which is due to the nature of the program itself. From column #12, we notice that the HMCS⟨3⟩ lock improves the performance by up to 1.55× compared to the C-MCS$_{out}$ lock. At lower contention, the HMCS⟨3⟩ lock incurs up to 9% slowdown compared to the two-level locks. This slowdown is expected due to the additional locking overhead of the third level.

### 6.2 Evaluation on SGI UV 1000

The SGI UV 1000 [9] used for our experiments consists of 256 blades. Each blade has two Intel Xeon X7560 processors processors, with the specifications shown in Table 1. Pairs of blades are connected with QuickPath interconnect. All nodes are connected via NUMAlink [9] technology. We had access to 4096 hardware threads out of the total 8192 on the system. We organized the SGI UV 1000 into a 5-level hierarchy:

| Level | Participants | Total |
|---|---|---|
| 1 | SMTs | 2 |
| 2 | Cores | 8 |
| 3 | 4 QPI connected sockets | 64 |
| 4 | 8-pairs of nodes on the same rack | 512 |
| 5 | 8 racks | 4096 |

For the C-MCS locks, we formed the cohorts at level-1 (C-MCS$_{in}$), level-2 (C-MCS$_{mid}$), and level-4 (C-MCS$_{out}$). We compared the C-MCS locks with a 5-level HMCS lock. In all locks we set infinite threshold at each level but fixed number of iteration for each thread, so that the lock may not be needed again once relinquished to the parent. This provided the highest possible throughput for any lock. We tested these locks both with empty and non-empty CS. The non-empty CS case touched two cache lines inside the CS and spent 0-2.5 microseconds outside the CS to mimic the test setup by Dice et al. [4]. Table 5(a) shows that the HMCS lock outperforms the peak throughput of all other locks in both modes. The HMCS lock outperforms even the C-MCS$_{in}$ because the threshold

**Table 4.** Comparison of different synchronization strategies for K-means on IBM Power 755.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Running time in micro seconds | | | | | Improvement over Atomic ops | | | | HMCS⟨3⟩ improvement over | |
| Num Threads | Atomic ops | MCS lock | C-MCS$_{in}$ lock | C-MCS$_{out}$ lock | HMCS⟨3⟩ lock | MCS (#2/#3) | C-MCS$_{in}$ (#2/#4) | C-MCS$_{out}$ (#2/#5) | HMCS⟨3⟩ (#2/#6) | C-MCS$_{in}$ (#4/#6) | C-MCS$_{out}$ (#5/#6) |
| 1 | 1.61E+08 | 4.34E+07 | 4.77E+07 | 4.78E+07 | 5.23E+07 | 3.70× | 3.37× | 3.36× | 3.07× | 0.91× | 0.91× |
| 2 | 1.29E+08 | 3.83E+07 | 3.70E+07 | 3.79E+07 | 4.03E+07 | 3.37× | 3.49× | 3.41× | 3.21× | 0.92× | 0.94× |
| 4 | 9.09E+07 | 2.95E+07 | 3.03E+07 | 3.03E+07 | 3.22E+07 | 3.08× | 3.01× | 3.01× | 2.83× | 0.94× | 0.94× |
| 8 | 1.28E+08 | 1.92E+07 | 1.82E+07 | 2.08E+07 | 1.90E+07 | 6.69× | 7.06× | 6.17× | 6.75× | 0.96× | 1.09× |
| 16 | 2.75E+08 | 2.58E+07 | 1.97E+07 | 2.67E+07 | 1.92E+07 | 10.7× | 14.0× | 10.3× | 14.4× | 1.03× | 1.39× |
| 32 | 4.02E+08 | 2.70E+07 | 1.77E+07 | 2.72E+07 | 1.76E+07 | 14.9× | 22.7× | 14.8× | 22.9× | 1.01× | 1.55× |
| 64 | 4.82E+08 | 5.62E+07 | 3.24E+07 | 3.56E+07 | 2.43E+07 | 8.58× | 14.9× | 13.5× | 19.8× | 1.33× | 1.46× |
| 128 | 6.49E+08 | 7.81E+07 | 5.67E+07 | 5.45E+07 | 3.93E+07 | 8.31× | 11.4× | 11.9× | 16.5× | 1.44× | 1.39× |

**Table 5.** Throughput (locks/sec) improvement of HMCS vs. C-MCS locks on a 4096-thread SGI UV 1000.

| | (a) Throughput mode | | | | (b) Fairness mode | | | |
|---|---|---|---|---|---|---|---|---|
| | HMCS⟨4⟩ | C-MCS$_{in}$ | C-MCS$_{mid}$ | C-MCS$_{out}$ | HMCS⟨4⟩ | C-MCS$_{in}$ | C-MCS$_{mid}$ | C-MCS$_{out}$ |
| **Empty CS** | 3.09E+07 | 1.26E+06 | 9.76E+06 | 4.32E+05 | 2.43E+06 | 5.02E+05 | 2.28E+06 | 4.11E+05 |
| (HMCS improvement) | - | (24.6x) | (3.17x) | (71.5x) | - | (4.84x) | (1.07x) | (5.92x) |
| **Non-empty CS** | 5.63E+06 | 5.27E+05 | 4.34E+06 | 2.61E+05 | 5.63E+06 | 3.78E+05 | 1.71E+06 | 3.96E+05 |
| (HMCS improvement) | - | (10.7x) | (1.3x) | (21.5x) | - | (14.9x) | (3.29x) | (14.2x) |

$c_{in}$ needed to amortize the latency of deep NUMA hierarchy is larger than the number of lock acquisitions needed by each thread.

We also compared the C-MCS locks with a 4-level HMCS lock, where the level-1 and level-2 were collapsed into a single domain. In this case, we set the threshold at each level to the number of participants at that level. This provided the highest fairness for any lock. As before, we tested these locks both with empty CS and non-empty CS. Table 5(b) shows that the HMCS lock outperforms the peak throughput of all other locks in both modes.

## 7. Conclusions and Future Work

This paper introduces the Hierarchical MCS (HMCS) lock for systems with NUMA hierarchies. We present analytical models for throughput and fairness of both HMCS and two-level Cohort MCS (C-MCS) locks that provide insight into properties of these designs. Experiments confirm the accuracy of our models. On systems with more than two levels of NUMA hierarchy, HMCS locks deliver high throughput with significantly less unfairness than C-MCS locks. Differences in access latencies between NUMA domains at different levels of a hierarchy determine what levels of the hierarchy are worth exploiting. Given a measure of the passing time at each level of a hierarchy, our models show how to pick cohorting thresholds that deliver a desired fraction of the maximum throughput. Experimental results confirm that the only way a C-MCS lock can match the throughput of an HMCS lock is under extreme contention with the inner level of cohorting configured for threads sharing a core, which was not the design point that Dice et al. intended for C-MCS locks. Furthermore, matching the throughput of HMCS locks with inner-level C-MCS locks requires cohorting thresholds that yield much greater unfairness.

While our experiments to date were conducted on shared memory platforms, our work was motivated by distributed-memory systems that use Remote Direct Memory Access (RDMA) primitives to implement system-wide queueing locks. The latency of RDMA operations between nodes is much higher than the latency of loads and stores within nodes. We believe that HMCS locks are the best way to achieve high throughput on such systems. Implementing HMCS locks for such systems remains future work.

## Acknowledgments

This work was supported in part by the DOE Office of Science through cooperative agreement DE-SC0010473 and Lawrence Berkeley National Laboratory subcontract 7106218. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. Specifically, it used the Blacklight system at the Pittsburgh Supercomputing Center (PSC). The IBM Power 755 used for experiments was acquired with support from NIH award NCRR S10RR02950 and an IBM Shared University Research (SUR) Award in partnership with CISCO, Qlogic and Adaptive Computing. We thank Karthik Murthy for suggesting the K-means application.

## References

[1] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proc. Linux Symposium*, 2012.

[2] P. Buhr, D. Dice, and W. Hesselink. High-performance n-thread software solutions for mutual exclusion. In *Concurrency and Computation: Practice and Experience, Early View*, 2014.

[3] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA Locks. In *Proc. of the Twenty-third Annual ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, 2011.

[4] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, 2012.

[5] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proc. of the 12th Intl. Conf. on Parallel Processing*, Euro-Par'06, pages 801–810, 2006.

[6] P. S. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proc. of the 8th Intl. Symp. on Parallel Processing*, pages 165–171, 1994.

[7] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[8] R. Narayanan, B. Azisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *IEEE International Symposium on Workload Characterization*, pages 182–188, 2006.

[9] SGI. SGI Altix UV 1000 System User's Guide. http://techpubs.sgi.com/library/manuals/5000/007-5663-003/pdf/007-5663-003.pdf.

[10] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 533–544, New York, NY, USA, 1998. ACM.