Software Combining to Mitigate Multithreaded MPI Contention

Abdelhalim Amer
Argonne National Laboratory
aamer@anl.gov

Charles Archer
Intel Corporation
charlesarcher@gmail.com

Michael Blocksome
Intel Corporation
michael.blocksome@intel.com

Chongxiao Cao
Intel Corporation
chongxiao.cao@intel.com

Michael Chuvelev
Intel Corporation
michael.chuvelev@intel.com

Hajime Fujita
Intel Corporation
hajime.fujita@intel.com

Maria Garzaran
Intel Corporation
maria.garzaran@intel.com

Yanfei Guo
Argonne National Laboratory
yguo@anl.gov

Jeff R. Hammond
Intel Corporation
jeff.r.hammond@intel.com

Shintaro Iwasaki
The University of Tokyo
iwasaki@eidos.ic.i.u-tokyo.ac.jp

Kenneth J. Raffenetti
Argonne National Laboratory
rafenet@mcs.anl.gov

Mikhail Shiryaev
Intel Corporation
mikhail.shiryaev@intel.com

Min Si
Argonne National Laboratory
msi@anl.gov

Kenjiro Taura
The University of Tokyo
tau@eidos.ic.i.u-tokyo.ac.jp

Sagar Thapaliya
Intel Corporation
sagar.thapaliya@intel.com

Pavan Balaji
Argonne National Laboratory
balaji@anl.gov

Abstract

Efforts to mitigate lock contention from concurrent threaded accesses to MPI have reduced contention through fine-grained locking, avoided locking altogether by offloading communication to dedicated threads, or alleviated negative side effects from contention by using better lock management protocols. The blocking nature of lock-based methods, however, wastes the asynchrony benefits of nonblocking MPI operations, and the offloading model sacrifices CPU resources and incurs unnecessary software offloading overheads under low contention.

We propose new thread safety models, CSync and LockQ, based on *software combining*, a form of software offloading without the requirement for dedicated threads; a thread holding the lock *combines* work of threads that failed their lock acquisitions. We demonstrate that CSync, a direct application of software combining, improves scalability but suffers from lack of asynchrony and incurs unnecessary offloading. LockQ alleviates these shortcomings by leveraging MPI semantics to relax synchronization and reduce offloading requirements. We present the implementation, analysis, and evaluation of these models on a modern network fabric and show that LockQ outperforms most existing thread safety models in low- and high-contention regimes.

CCS Concepts

• **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

ACM Reference Format:
Abdelhalim Amer, Charles Archer, Michael Blocksome, Chongxiao Cao, Michael Chuvelev, Hajime Fujita, Maria Garzaran, Yanfei Guo, Jeff R. Hammond, Shintaro Iwasaki, Kenneth J. Raffenetti, Mikhail Shiryaev, Min Si, Kenjiro Taura, Sagar Thapaliya, and Pavan Balaji. 2019. Software Combining to Mitigate Multithreaded MPI Contention. In *Proceedings of ACM International Conference on Supercomputing 2019 (ICS'19).* ACM, New York, NY, USA, 13 pages.

1 Introduction

The Message Passing Interface (MPI) remains the predominant programming system on distributed-memory high-performance computing (HPC) platforms. Because of the inadequacy of the message-passing model to program shared-memory parallel systems, however, MPI users are moving to hybrid models, MPI+X, that leverage MPI for distributed-memory programming and another programming system, X, such as OpenMP, suitable for shared memory [17]. In MPI+threads programming, concurrent thread accesses to the MPI stack are allowed and supported by most production MPI libraries. Although these libraries satisfy the functional requirement of multithreaded MPI, however, most, if not all, suffer from contention, which hinders application performance.

Among the major sources of contention is competition for lock acquisition, since locks are the primary mechanism used by these libraries to protect shared state. Several methods have been developed to mitigate lock contention issues and can be grouped into three orthogonal approaches. The first relies on *contention reduction* through fine-grained critical sections and atomic operations where needed (e.g., reference counting) [5, 9, 16, 19]. This approach,

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330378

however, reduces but does not eliminate contention and provides no remedy when it takes place. Blocking lock acquisitions also introduces nondeterministic and often long delays for nonblocking MPI operations, thus wasting their asynchrony benefits. The second approach focuses on lock *contention avoidance* by eliminating lock acquisitions on the critical path [10, 21, 28]. This is achieved by using an *offload model* where application threads offload communication operations to dedicated communication threads. This model has two shortcomings: it requires dedicated communication threads that compete with application threads for resources, and the software offloading system incurs compulsory overheads even when contention is nonexistent. The third approach aims at better *contention management*; when contention takes place, it attempts to reduce negative side effects by passing lock ownership to threads with productive work [1, 3, 8]. This is achieved by leveraging MPI internal knowledge to drive an adaptive locking protocol. This approach has also the asynchrony issue on nonblocking operations, however, in addition to being less practical because it requires hardware-aware custom lock implementations.

We propose new thread safety models based on *software combining* [11, 14, 24], a form of software offloading that does not require dedicated threads; lock holders become the *combiner* threads that execute work on behalf of threads that failed their lock acquisitions, thus behaving similarly to dedicated communication threads of software offloading methods. We first present CSync, an application of the state-of-the-art DSM-Synch [11] software combining technique to communication operations, which required API extensions for ease of integration. We demonstrate that CSync indeed significantly reduces scalability degradation but suffers from the same asynchrony and offloading shortcomings of the lock-based methods and software offloading, respectively. We then discuss the limits of software combining techniques that rely on a *coupled lock-list* data structure, and we propose an alternative model, LockQ, that decouples lock and the corresponding list data structures and leverages MPI semantics to (1) preserve asynchrony of MPI nonblocking operations under practical assumptions and (2) avoid unnecessary offloading. LockQ eliminates the unbounded synchronization delays on performing nonblocking MPI operations; to our knowledge, it is the first model that provides this progress property without relying on dedicated communication threads.

We present the implementation, analysis, and comparative evaluation of six thread safety models—CSync, LockQ, and four other models found in the literature—on a modern network fabric. Our experimental method includes communication-intensive benchmarks, graph traversal and particle transport proxy applications, and an integration of all the models in the same production MPI library. Our results show that LockQ outperforms CSync and the offload model under low contention. At high contention, LockQ performs competitively against CSync and the offload model and significantly outperforms the other methods while being hardware agnostic.

2 Background and Related Work

We describe in this section the subtle interaction between application threads and MPI. We also define concepts and terminology used throughout the paper, and we discuss the various contention-mitigating thread safety models found in the literature.

2.1 MPI+Threads Interaction

MPI is a library specification that allows but does not require concurrent multithreaded accesses from the MPI user. Given the increasing number of hybrid MPI+threads applications, demand for concurrent accesses to MPI has risen accordingly, resulting in most production MPI libraries supporting this mode of access. MPI defines two basic rules for concurrent accesses: *thread safety* and *progress guarantee*. Libraries satisfying the latter rule guarantee that a thread blocked within the MPI library does not obstruct the progress of other threads. Other rules exist and derive from single-threaded requirements but are less pertinent for the remainder of this paper. For instance, collective operations on the same communicator must be issued in the same order by all processes; thus, application threads involved in different collective operations on the same communicator must synchronize outside MPI in order to guarantee such ordering.

2.2 Terminology and Baseline

It is impractical to attempt a thorough review of the differences between MPI libraries and the methods developed in the past two decades to tackle thread safety issues in MPI. Instead, we group the various approaches found in practice and in the literature into what we call *thread safety models* and then illustrate how they are implemented through simplified code snippets. Our algorithmic descriptions will rely on *low-level network* (LLN) and *thread safety* building blocks to implement the internals of an MPI library. Our goals are simplicity and expressiveness to capture necessary details while leaving out noncritical information. The following are the assumptions and terminology used throughout the rest of the paper.

Low-Level Network. We assume that the MPI library calls internally a low-level network interface that is not thread safe; MPI, as a result, has to protect such calls in order to avoid corruption from multithreaded accesses to this API. We use `LLN_` to prefix such calls. In practice, this API can be seen as wrapping network fabric calls, which are closer to the hardware, such as the OpenFabrics Interfaces (OFI) [13] and Unified Communication X (UCX) [27], and doing the necessary translation from MPI-level information to the target low-level interface.

Critical Sections. To implement critical sections, we assume a locking API that supports lock *acquire* and *release* operations as well as a *lock acquisition attempt* operation. These operations are represented with the `lock_acquire`, `lock_release`, and `lock_tryacq` calls, respectively. For instance, these could map to the POSIX API calls `pthread_mutex_lock`, `pthread_mutex_unlock`, and `pthread_mutex_trylock` calls, respectively.

Baseline Model. Here we consider the most basic model, yet often adopted in practice, described using the above terminology and assumptions as illustrated in Figure 1a. This model relies on a *global coarse-grained lock* to protect every MPI call that accesses a shared state. The code snippets throughout the paper will use a nonblocking communication operation (`MPI_Isend`) and a blocking progress operation (`MPI_Wait`) to drive the discussion. The lock (`L`) ensures thread safety by requiring each MPI routine to acquire and release the lock at the entry and exit of the routine, respectively. `MPI_Isend` is required to create a user-visible request object (`req` at line 3) and to translate the call to the network fabric (`LLN_isend`). This model assumes the absence of asynchronous progress either because the LLN does not support it or because the application does not want to sacrifice CPU resources to enable it. As a result, application threads have to drive progress manually when in the MPI library (line 10). To respect MPI's progress semantics, blocking operations have to release the lock if the operations they are waiting for have not completed, in order to allow other threads to progress (line 12). While waiting for completion, the MPI library can optionally `pause` or `yield` the CPU to improve resource utilization.

As can be seen in this example, if two threads call `MPI_Isend` and `MPI_Wait` in parallel, they will compete for the lock `L`. If `MPI_Wait` holds the lock while there is no operation to complete, `MPI_Isend` will block waiting for lock acquisition unnecessarily. We consider `MPI_Wait` in this case as incurring unproductive lock acquisitions and thus wasting the opportunity for communication operations, such as `MPI_Isend`, to secure a productive lock acquisition in a timely manner. Moreover, waiting on a lock acquisition in `MPI_Isend` might block for an *unbounded* number of steps, thus wasting the asynchrony benefits of an MPI nonblocking call.

2.3 Survey of Existing Thread Safety Models

The most efficient way to describe existing contention-mitigating efforts is to depart from the coarse-grained locking model and show how improvements are made over the baseline implementation of `MPI_Isend` and `MPI_Wait`. The code snippets for these models are illustrated in Figures 1b through 1d; the key differences with respect to the coarse-grained model are highlighted with a colored background.

Fine-Grained Locking. This approach aims at reducing contention by shrinking the length of the critical sections. This is a large category in practice because numerous ways exist to protect shared states and objects (instances of this model can be found in [5, 9, 16, 19]). One way is to use a single lock, but acquiring and releasing the lock occur at fine-grained levels (e.g., just before touching a shared object). Another method is to use multiple locks where each lock protects a different object or a class of objects. In Figure 1b, we illustrate one way to reduce contention using fine-grained locks. The responsibility of the global lock is split into two locks, `req_L` and `LLN_L`, to independently protect request memory management operations and LLN operations, respectively. This model has several shortcomings, however. Contention can still take place in hotspots, such as when multiple threads actively wait and contend for the `LLN_L` lock. The overheads of this model in low-contention regimes also grow with the number of locking operations on the critical path; indeed, the locking operations incur overheads associated with function calls (if not inlined), memory barriers, and atomic operations that hurt instruction-level parallelism.

LLN Lockless Offloading. This model ensures contentionless access to the LLN (Figure 1c). The bulk of the work is performed by dedicated threads belonging to the LLN. Application threads simply post communication operations and either leave (nonblocking operation) or busy wait on a flag (blocking operation). The application thread first creates a work descriptor (line 6) and uses an LLN routine to post the descriptor (software offloading; usually a lock-free enqueue operation). A communication thread then pulls the descriptor and executes the operation on behalf of the calling thread. Waiting for completion simply involves checking the status of the request; no progress calls are required (loop at line 10). Lock contention is completely avoided in this case since it is lockless (note that contention for a descriptor queue might still occur), and the approach reduces interference between issuing operations and waiting for their completion since they are no longer coupled through lock acquisitions. This model, however, sacrifices CPU resources for the sake of the communication threads that might interfere with application threads when competing for on-node resources. Furthermore, software offloading incurs overheads in the absence of contention because of unnecessary offloading; in this case, a thread can directly issue the operation instead of creating, posting, and relying on the dedicated thread to dequeue and execute a work descriptor. MPICH2 over PAMI (Parallel Active Message Interface) on Blue Gene/Q systems [21], the work by Vaidyanathan et al. that offloads MPI communication to a dedicated thread [28], and the work by Wataru et al. that offloads Infiniband communication operations to user-level threads [10] are examples that follow this model.

Lock Contention Management. This model does not alter lock granularity. Instead, contention is managed in a way that reduces overheads outside serialization. For instance, as shown by Amer et al. [1, 3], in order to reduce the waste from unnecessary lock acquisitions in the progress loop (lline 14 in Figure 1a), the locking protocol can prioritize lock acquisitions with higher productive potential. Figure 1d shows an example of a locking API that exposes a routine `lock_acquire_low` that gives lower priority to the calling thread, thus prioritizing threads calling `lock_acquire`. To further reduce contention when waiting for completion, Dang at al. showed how the progress loop can be managed in a server-client model where one of the application threads (server) calls network progress while the others (clients) wait on a local flag [8]. The server wakes up threads that have their pending operations completed.

In the lock-based examples, locks have been used as the unique form of synchronization. Protecting LLN calls in this way is common and important for correctness and performance portability. Protecting the request object pool (line 2 in Figures 1b and 1c) is done only for illustration and can be implemented in a lockless manner. With the exception of the offload model, all the above models impose blocking lock acquisition on nonblocking MPI operations (`MPI_Isend` in the example). Depending on the performance and fairness of the lock implementation, the caller might wait for an unbounded number of steps to acquire the lock before issuing an operation, thus reducing the asynchrony benefits of MPI nonblocking operations. The offload model effectively eliminates this issue as long as the underlying queuing system guarantees a bounded number of steps to enqueue a work descriptor, which is a *wait-freedom* requirement. Since wait-free queues have been demonstrated in practice [20], this requirement can be satisfied, and the offload model can retain the asynchronous property of nonblocking calls. This model, however, imposes (1) offloading overheads (even in the absence of contention) and (2) communication threads, which sacrifice CPU resources and potentially cause interference with application threads.

3 Combining-Based Thread Safety Models

Ideally, the thread safety model would retain the asynchrony benefits of the offload model without requiring dedicated communication threads. In the following, we present two new thread safety

```
MPI_Isend (ARGS,*req) {
  lock_acquire(L);
  request_create(ARGS,req);
  LLN_isend(ARGS,req);
  lock_release(L);
}
MPI_Wait (ARGS,*req) {
  lock_acquire(L);
  while (!complete(req)) {
    LLN_progress_all();
    if (!complete(req)) {
       lock_release(L);
       /*[pause or yield]*/;
       lock_acquire(L);
    }
  }
  free(req);
  req = REQUEST_NULL;
  lock_release(L);
}
```

(a) Coarse-grained global locking

```
MPI_Isend (ARGS,*req) {
  lock_acquire(req_L);
  request_create(ARGS,req);
  lock_release(req_L);
  lock_acquire(LLN_L)
  LLN_isend(ARGS,req);
  lock_release(LLN_L);
}
MPI_Wait (ARGS,*req) {
  while (!complete(req)) {
    /*[pause or yield]*/;
    lock_acquire(LLN_L);
    LLN_progress_all();
    lock_release(LLN_L);
  }
  lock_acquire(req_L);
  free(req);
  lock_release(req_L);
  req = REQUEST_NULL;
}
```

(b) Fine-grained locking

```
MPI_Isend (ARGS,*req) {
  lock_acquire(req_L);
  request_create(ARGS,req);
  lock_release(req_L);
  /*create work descriptor*/
  descr_create(ARGS,req,&d);
  LLN_post(d);
}
MPI_Wait (ARGS,*req) {
  while (!complete(req)) {
   /* progress done by
   communication threads*/
   /*[pause or yield]*/;
  }
  lock_acquire(req_L);
  free(req);
  lock_release(req_L);
  req = REQUEST_NULL;
}
```

(c) LLN lockless offloading

```
MPI_Isend (ARGS,*req) {
  lock_acquire(L);
  request_create(ARGS,req);
  LLN_isend(req);
  lock_release(L);
}
MPI_Wait (ARGS,*req) {
  lock_acquire(L);
  while (!complete(req)) {
    LLN_progress_all();
    if (!complete(req)) {
       lock_release(L);
       /*[pause or yield]*/;
       lock_acquire_low(L);
    }
  }
  free(req);
  req = REQUEST_NULL;
  lock_release(L);
}
```

(d) Priority locking

Figure 1: Simplified description of various thread safety models. `LLN_progress_all` **progresses all network resources (global progress);** `complete(req)` **returns** `TRUE` **if** `req` **has completed;** `ARGS` **captures function arguments that are unnecessary to mention individually and would otherwise clutter the code snippets.**

models for multithreaded MPI based on software combining. The first model, CSync, is mostly a direct application of DSM-Synch [11], which is a scalable implementation of the combining principle. This model borrows from the offload model the concept of handing over work to another thread (i.e., combiner) but without requiring dedicated threads. This approach improves scalability by reducing remote memory references but carries over blocking synchronization as done in lock-based models. The second model, LockQ, addresses the shortcomings of CSync by relaxing synchronization on the critical path of nonblocking MPI operations by exploiting MPI knowledge. In the following, we describe the step-by-step process of implementing these models while discussing their costs on the critical path in the absence and presence of contention.

3.1 Software Combining in Practice

Combining is an old technique that was used in hardware [25] and software [30] to mitigate memory and network contention by combining requests from the same memory location. Software combining has also been used to implement concurrent data structures by exploiting the fact that sequentially combining multiple requests by the same thread (or processor) reduces the overall memory and network traffic. These benefits have long been questioned, however, because of high synchronization overheads.

Recently, software combining has become more popular thanks to more efficient implementations. The most notable works that made them more practical are *flat-combining* by Kendler et al. [14] and CC-Synch and DSM-Synch by Fatourou et al. [11]. These techniques share the general idea that critical section work is protected by a lock and represented by a *request* object. A thread first announces its request by pushing it to an *announcement list* and then proceeds to compete for the lock. We refer to this thread as an *announcer*. The thread that succeeds in acquiring the lock becomes the *combiner* that will not only execute its own request but also execute requests found in the announcement list. On lock acquisition failure, the thread busy waits until either its request has completed or the lock has been released. We refer to this step as *synchronizing* between an announcer thread and a combiner one.

To our knowledge, CC-Synch and DSM-Synch are the most scalable software combining techniques and draw their efficiency from the lock algorithms they are derived from: CC-Synch is based on CLH [22], and DSM-Synch is based on MCS [23]. Given that these lock algorithms build implicit queues of waiting threads, the corresponding software combining techniques reuse the queues to function as announcement lists. In addition to supporting lock ownership passing, these queues allow explicit traversal by the lock holder and execute the requests found in the queue nodes. CC-Synch has been proven to be slightly superior to DSM-Synch in practice, but both outperform all prior software combining techniques. We choose hereafter to use DSM-Synch instead of CC-Sync because it is based on MCS; this allows us to better isolate the performance differences with respect to the following Per-VNI and LockQ models that use MCS-related primitives.

To avoid confusion with MPI request objects, in the following we refer to *requests* being posted on the announcement list as *work descriptors*, and we use the terminology *work queue*, *combining queue*, and *announcement list* interchangeably.

3.2 Critical Section Scope

Software combining can be applied to any critical section, but it should be limited to offloading nonblocking operations because blocking operations will penalize the combining thread for unbounded periods of time, resulting in load imbalance and, worse, possibly leading to deadlocks in the context of MPI due to data dependencies between threads. Consequently, this paper limits software combining to only nonblocking LLN operations, which are often subject to significant contention hotspots. To expose internal parallelism in the MPI library while reducing the complexity of maintaining a large number of critical sections, we chose to implement software combining at the level of the *virtual network interface* (VNI). This thread safety granularity is a subcategory of the previously described *fine-grained* locking model. A VNI is an abstract object that encapsulates an independent network resource and is thread unsafe; it requires external thread safety mechanisms to ensure single-threaded access. For instance, it could be mapped to an OFI *endpoint*. This level of locking granularity has been exploited by Cray MPT [19] and Open MPI [16] to encapsulate network contexts, for instance. Any shared state outside a VNI, such as a request, is not protected by VNI locks and requires separate protection.

Figure 3a illustrates how the VNI-level granularity operates. We label the resulting thread safety model as Per-VNI, which is an intermediate model before applying software combining. The figure also highlights the changes over the fine-grained locking model in Figure 1b. We observe that the main difference between the two models is that Per-VNI allows more fine-grained locking by protecting independent VNIs separately as opposed to protecting all LLN operations with the same lock. The mapping between MPI-level information to the target VNI is done with the hash() function, developed with the goals of minimizing thread contention and maximizing network resource usage. MPI semantics, however, require some form of global progress in case the LLN does not support all of MPI in a hardware-native way. That is, if a single MPI operation is not supported natively by the network fabric, the MPI library has to emulate the operation with active messages.[1] As a result, global progress is necessary (line 22), but its overhead on the critical path can be controlled by some MPI library-specific policy.[2]

Costs of Per-VNI **over Global**. The major extra costs are atomic reference counting of request objects (to address races, such as when two threads access the same request while one of them is outside critical sections) and additional lock acquisitions on the critical path (two locks in the example figure).

3.3 The CSync Thread Safety Model

Applications of software combining have been demonstrated mostly to implement basic data structures, such as queues and stacks. Our goal in this work is to execute complex communication operations, such as network calls that traverse several layers of the software stack. One particular challenge we encountered was the programmability constraint of software combining; the user-facing API is often a single synchronization function that merges three operations in one API routine: acquiring the lock, combining, and releasing the lock. This forces the user to implement every critical section that accesses an object as a function (often called apply()) that operates on a descriptor. This can be unnecessary programming complexity especially for noncritical operations, which can be satisfactorily implemented with traditional mutual exclusion. In addition, indiscriminately announcing every operation regardless of productivity aspects pollutes the work queue with unproductive operations (e.g., `LLN_progress` can be unproductive). In the following, we describe DSM-Synch and its API extension to support traditional mutual exclusion and alleviate the above shortcomings.

3.3.1 DSM-Synch *and API Extension.*

The new DSM-Synch API is composed of `dsm_synch` (the original function); `dsm_acquire`, which performs lock acquisition and combining; and `dsm_release`, which releases the lock. For `dsm_acquire` to work while sharing the same queue as `dsm_synch`, a thread enqueues a special descriptor to announce an empty operation (we chose NULL as the special value, but any other carefully chosen constant can be used). A combiner thread then halts combining on seeing such a descriptor and passes ownership of the lock. Because frequently halting combining hurts scalability, we allow at most one empty descriptor in the queue, specifically by blocking `dsm_acquire` callers with an MCS lock and letting at most one thread proceed to compete for the combining queue. Details follow.

We first reproduce the original DSM-Synch algorithm from Fatourou et al. [11] in Figure 2a. We highlight the major algorithmic differences with respect to the original MCS algorithm it was derived from, as well as a bug fix at line 63. We notice that `dsm_synch` performs lock acquisition, combining, and lock release in the same API routine. This approach assumes that every access to the original critical section must go though the `dsm_synch` operation. In practice, however, we found that this is constraining on the programmer, involves a performance penalty of announcing a request when contention does not take place, and potentially pollutes the work queue with unproductive operations. As a result, we concluded that supporting traditional mutual exclusion along with software combining gives more flexibility to the programmer and provides means to avoid the performance overheads.

To develop a more expressive API, we first decoupled the three operations being performed in `dsm_synch` to extract reusable components (Figure 2b). The only minor change needed for this step is to keep track of the head of the queue when combining, which is highlighted in the code snippet. The changes allow the new release routine to infer whether a lock release operation is necessary (i.e., I am the combiner so I have to pass ownership of the lock). Next, we show our extensions to DSM-Synch to allow traditional mutual exclusion (Figure 2c). Let us first ignore the new MCS lock at lines 3, 17, and 28. The new API routine dsm_acquire performs a lock acquisition by enqueuing a NULL request (line 19), which mostly follows the original MCS algorithm. The calling thread has to wait until it acquires the lock and implies that it has to become a combiner thread. Thus, it follows with a call to combine (line 21). The routine dsm_release simply calls release (line 26). For this to work correctly, the most important change lies in the combine routine. The combiner thread must avoid executing a NULL request (lines 42–47) and break out of the loop when the next thread in-line called dsm_acquire (i.e., its request is NULL; line 54). In other words, combining is halted, and a lock ownership passing is enforced.

These changes allow correct behavior but exhibit a serious performance flaw under contention. In the presence of a large number of NULL requests, combining will be halted frequently, thus wasting its benefit. We tackled this issue by allowing at most one NULL request in the combining queue using a two-step lock acquisition algorithm. On calling dsm_acquire, only the lock holder of D->lock (line 17) proceeds to acquire the combining lock (line 19), effectively filtering the surplus of NULL requests.

3.3.2 *The* CSync *Model.*

Here we rely on the DSM-Synch API. In Figure 3b we illustrate how it is applied over the Per-VNI model. DSM-Synch assumes a global apply() function that operates on a work descriptor (lines 1–6). Our implementation uses operation codes to distinguish between the LLN operations to execute. For instance, `ISEND` corresponds to `LLN_isend`. Then, we replace the critical sections around nonblocking operations by creating a work descriptor followed by calling `dsm_synch`. At this stage, the thread either acquires the lock and combines operations or waits for another combiner thread to execute the operation on its behalf. For noncritical LLN operations (e.g., `LLN_progress`), mutual exclusion

[1] For instance, we know of no network fabric that supports natively the MAXLOC and MINLOC operations (Section 5.9.4 of the MPI-3.1 Standard). Doing so requires that MPI processes listen to potential active message requests to be serviced, which might arrive on any VNI.

[2] E.g., the branch at line 21 could be taken infrequently.

```
 1 typedef struct qnode {
 2   void *req;
 3   unsigned status;
 4   struct qnode *next;
 5 } qnode_t;
 6 /* thread private node */
 7 typedef struct tnode {
 8   qnode_t qnodes[2];
 9   int toggle;
10 } tnode_t;
11 thread_local tnode_t tnode;
12 typedef struct dsm {
13   qnode_t *tail;
14 } dsm_t;
15 void dsm_synch(dsm_t *D, void *req){
16   qnode_t *tmp, *local, *pred;
17   /* prepare my local node */
18   tnode->toggle = 1 - tnode->toggle;
19   local = &tnode->qnodes[tnode->toggle];
20   local->status = WAIT;
21   local->next = NULL;
22   local->req = req;
23
24   /* swap with global lock (queue tail)
25    * this announces my request "req" */
26   pred = SWAP(D->tail, local);
27   /* lock owned by other thread (combiner)
28    * update next and wait on my status */
29   if (pred != NULL) {
30     pred->next = local;
31     while(local->status == WAIT) /*NOP*/;
32     /* return if request completed */
33     if(local->status == COMPLETE)
34       return;
35   }
36
37   /* am combiner and req is pending */
38   tmp = local;
39   int counter = 0;
40   while (1) {
41     apply(tmp->req);
42     tmp->status = COMPLETE;
43     if (tmp->next == NULL ||
44         tmp->next->next == NULL ||
45         counter > MAX_COMBINE)
46       break;
47     tmp = tmp->next;
48     counter++;
49   }
50
51   /* release the lock */
52   if (tmp->next == NULL) {
53     if(CAS(D->tail, tmp, NULL))
54       return;
55     /* wait pending enq to update next */
56     while (tmp->next == NULL) /*NOP*/;
57   }
58
59   /* reached maximum combining operations
60    * or false-positive empty queue.
61    * elect next thread as the combiner.*/
62   tmp->next->status = UNLOCKED;
63   tmp->next = NULL; /* omitted for correctness */
64 }
```

(a) **Original** DSM-Synch[11]

```
 1 typedef struct tnode {
 2   struct qnode qnodes[2];
 3   int toggle;
 4   qnode_t *head;
 5 } tnode_t;
 6 void dsm_synch(dsm_t *D, void *req) {
 7   /* (1) acquire lock or enq req */
 8   acq_enq(D, req);
 9   /* (2) combine requests if any */
10   combine(D);
11   /* (3) release lock if needed. */
12   release(D);
13 }
14 void acq_enq(dsm_t *D, void *req) {
15   qnode_t *local, *pred;
16   tnode->toggle = 1 - tnode->toggle;
17   local = &tnode->qnodes[tnode->toggle];
18   local->status = WAIT;
19   local->next = NULL;
20   local->req = req;
21   pred = SWAP(D->tail, local);
22   if (pred != NULL) {
23     pred->next = local;
24     while(local->status == WAIT) /*NOP*/;
25     if(local->status == COMPLETE)
26       return;
27   }
28 }
29 void combine(dsm_t *D) {
30   qnode_t *tmp, *local;
31   local = &tnode->qnodes[tnode->toggle];
32   if (local->status == COMPLETE) {
33   /* combine and release unnecessary */
34       tnode->head = NULL;
35       return;
36   }
37   tmp = local;
38   int counter = 0;
39   while (1) {
40     apply(tmp->req);
41     tmp->status = COMPLETE;
42     if (tmp->next == NULL ||
43         tmp->next->next == NULL ||
44         counter > MAX_COMBINE)
45       break;
46     tmp = tmp->next;
47     counter++;
48   }
49   tnode->head = tmp;
50 }
51 void release(dsm_t *D) {
52   qnode_t *tmp = tnode->head;
53   /* tmp = head or NULL if req completed
54    * NULL, no need to perform release. */
55   if (tmp == NULL)
56       return;
57   if (tmp->next == NULL) {
58     if(CAS(D->tail, tmp, NULL))
59       return;
60     while (tmp->next == NULL) /*NOP*/;
61   }
62   tmp->next->status = UNLOCKED;
63   tmp->next = NULL; /* omitted for correctness */
64 }
```

(b) **Logical Decomposition**

```
 1 typedef struct dsm {
 2   qnode_t *tail;
 3   mcs_t lock;
 4 } dsm_t;
 5
 6 void dsm_synch(dsm_t *D, void *req) {
 7   /* (1) acquire lock or enq req */
 8   acq_enq(D, req);
 9   /* (2) combine requests if any */
10   combine(D);
11   /* (3) release lock if needed. */
12   release(D);
13 }
14
15 void dsm_acquire(dsm_t *D) {
16   /* (1) acquire the MCS lock */
17   lock_acquire(D->lock);
18   /* (2) acquire the combining lock */
19   acq_enq(D, NULL);
20   /* (3) combine requests if any */
21   combine(D);
22 }
23
24 void dsm_release(dsm_t *D) {
25   /* (1) release the combining lock */
26   release(D);
27   /* (2) release the MCS lock */
28   lock_release(D->lock);
29 }
30
31 void combine(dsm_t *D) {
32   qnode_t *tmp, *local;
33   local = &tnode->qnodes[tnode->toggle];
34   if (local->status == COMPLETE) {
35   /* combine and release unnecessary */
36       tnode->head = NULL;
37       return;
38   }
39   tmp = local;
40   int counter = 0;
41   while (1) {
42     if (tmp->req == NULL) {
43         /* invariants:
44            (1) I am the combiner thread
45            (2) I called "DSM-Acquire"
46            (3) first loop iteration */
47         assert(counter == 0);
48     } else {
49         apply(tmp->req);
50         tmp->status = COMPLETE;
51     }
52     if (tmp->next == NULL ||
53         tmp->next->next == NULL ||
54         tmp->next->req == NULL ||
55         counter > MAX_COMBINE)
56       break;
57     tmp = tmp->next;
58     counter++;
59   }
60   tnode->head = tmp;
61 }
62
63
64
```

(c) **API Extension**

Figure 2: DSM-Synch **description (a), logical component breakdown and restructuring (b), and API extensions (c).** WAIT, UNLOCKED, **and** COMPLETE **denote compile-time constants for the state of a node, and** MAX_COMBINE **is the threshold for the number of requests a combiner thread is allowed to execute.** SWAP **and** CAS **represent atomic swap and compare-and-swap operations, respectively. The code highlighted in (a) indicates the major changes with respect to the original** MCS **algorithm (note that the original algorithm had a bug at line 63, which we fixed by simply avoiding that unnecessary and erroneous store operation); those in (b) indicate the algorithmic changes compared with the original** DSM-Synch **to break it into separate routines; and the highlights in (c) indicate the changes over (b) to support traditional mutual exclusion in addition to software combining.**

is used by protecting the call with `dsm_acquire` and `dsm_release`; `dsm_acquire` not only ensures lock acquisition but also does combining. To avoid a thread falling victim to heavy combining for the rest of the threads, a threshold on combining is used and set to 1K by default (`MAX_COMBINE` constant in Figure 2) to circulate the combining responsibility.

Costs of CSync **over** Per-VNI **with** MCS. The synchronization part of `dsm_synch` is identical to MCS. In the absence of contention, CSync's heaviest extra cost is setting the work descriptor, which involves one to two cache lines of load/store operations. If combining is performed, unpacking the arguments adds a similar overhead. Synchronization is reduced for noncombining threads since they only read the status of a descriptor. CSync incurs the same extra costs as Per-VNI with respect to atomic operations in addition to having one extra lock acquisition in `dsm_acquire`.

```
 1 MPI_Isend (ARGS,*req) {
 2   lock_acquire(req_L);
 3   request_create(ARGS,req);
 4   lock_release(req_L);
 5   idx = hash(req);
 6   lock_acquire((L[idx])
 7   LLN_isend(VNI[idx],req);
 8   lock_release(L[idx]);
 9 }
10
11
12
13
14
15 MPI_Wait (ARGS,&req) {
16   idx = hash(req);
17   while (!complete(req)) {
18     lock_acquire(L[idx]);
19     LLN_progress(VNI[idx]);
20     lock_release(L[idx]);
21     if(cond)
22       LLN_progress_global();
23   }
24   lock_acquire(req_L);
25   free(req);
26   lock_release(req_L);
27   req = REQUEST_NULL;
28 }
29
```

(a) Per-VNI **model**

```
 1 void apply(*d) {
 2   switch(d->op) {
 3   case ISEND: LLN_isend(d->ARGS,d->req);
 4     ...
 5   }
 6 }
 7 MPI_Isend (ARGS,*req) {
 8   lock_acquire(req_L);
 9   request_create(ARGS,req);
10   lock_release(req_L);
11   idx = hash(req);
12   descr_create(ARGS,req,&d);
13   dsm_synch((L[idx], d);
14 }
15 MPI_Wait (ARGS,&req) {
16   idx = hash(req);
17   while (!complete(req)) {
18     dsm_acquire(L[idx]);
19     LLN_progress(VNI[idx]);
20     dsm_release(L[idx]);
21     if(cond)
22       LLN_progress_global();
23   }
24   lock_acquire(req_L);
25   free(req);
26   lock_release(req_L);
27   req = REQUEST_NULL;
28 }
29
```

(b) CSync **model**

```
 1 MPI_Isend (ARGS,*req) {
 2   lock_acquire(req_L);
 3   request_create(ARGS,req);
 4   lock_release(req_L);
 5   idx = hash(req);
 6   if (lock_tryacq(L[idx])) {
 7     combine(Q[idx]);
 8     LLN_isend(ARGS,req);
 9     lock_release(L[idx]);
10   } else {
11     decsr_create(&d);
12     post(Q[idx],d);
13   }
14 }
15 MPI_Wait (ARGS,*req) {
16   idx = hash(req);
17   while (!complete(req)) {
18     lock_acquire(L[idx]);
19     combine(Q[idx]);
20     LLN_progress(VNI[idx]);
21     lock_release(L[idx]);
22     if(cond)
23       LLN_progress_global();
24   }
25   lock_acquire(req_L);
26   free(req);
27   lock_release(req_L);
28   req = REQUEST_NULL;
29 }
```

(c) LockQ **model**

Figure 3: Per-VNI, CSync, **and** LockQ **thread safety models.**

3.4 The LockQ Thread Safety Model

CSync improves on MCS in most cases, as will be shown in the Evaluation section, but it falls short in many ways: (1) the compulsory announcement incurs overheads of creating the work descriptor particularity in the absence of contention; (2) the `dsm_synch` routine is blocking, which wastes the asynchrony of MPI nonblocking operations; and (3) the effectiveness of combining is limited by shallow queue depth (because of threads waiting instead of pushing more work and also because of interruptions from lock acquisitions with empty descriptors). We observe that the coupled lock-list data structure of CSync is at the heart of the issue. Because threads use the lock data structure to announce operations, they cannot leave unless their operations have completed or the lock has been passed to them. If a thread `T` leaves after announcing its operation, the system might hang for a long time or even indefinitely because `T` failed at its ownership passing and combining responsibilities.

LockQ addresses these shortcomings by decoupling lock and announcement list data structures and by leveraging MPI semantics to relax synchronization, preserve asynchrony, and feed combining threads with more work than CSync would. LockQ, similarly to CSync, is implemented with the Per-VNI model as its baseline; Figure 3c highlights the changes over the Per-VNI model. Each VNI `v` is associated with not only a lock `L[v]` but also a work queue `Q[v]`. By decoupling lock and data structures, a thread can enqueue a descriptor and leave without causing trouble to threads competing for the lock. Any successful lock acquisition on a VNI must be followed by combining the operations in the corresponding queue.

In this decoupled model, however, posting and combining an operation are racy operations. Suppose `T1` is about to release the lock of work queue `Q[v]` (i.e., it is done being the combiner of that queue) and `T2` fails its lock acquisition and posts its operation on `Q[v]`. `T1` will not execute the operation since it already finished combining while `T2` is leaving. If no other thread acquires `L[v]`, the operation will never get executed.

This is where MPI semantics come into play. In MPI, a nonblocking operation must be followed by a synchronization (or progress) MPI call to ensure progress and check (or wait) for the completion of the target operation. For instance, the `MPI_Wait` and `MPI_Test` calls are used to check or wait for the completion of various requests, such as those associated with nonblocking point-to-point operations. `MPI_Win_flush` is an example of a synchronization call for remote memory access (RMA) operations. LockQ relies on these calls as the last resort to combine pending operations on a VNI. Since these calls must be issued by the user eventually, correctness of execution is ensured, and deadlocks are avoided. From a different perspective, after a thread announces the `MPI_Isend` operation, busy waiting for its execution is redundant since it can wait for its completion when calling `MPI_Wait`; thus, MPI semantics allowed relaxing the synchronization.

Let us look at how LockQ is implemented in practice. When executing a nonblocking operation, instead of waiting for the lock, a thread performs a nonblocking lock acquisition attempt (line 6). On success, the thread becomes the *combiner* for the target VNI and combines any pending operation in its work queue (line 7) before executing its own operation (line 8). Doing so is critical for the correctness of ordered communication (e.g., point-to-point or one-sided accumulate operations). On lock acquisition failure, the thread creates a work descriptor for the operations and announces (posts or enqueues at line 12) it on the target work queue. The posted operation may or may not be combined by a thread executing another nonblocking operation. The user, however, has to follow the `MPI_Isend` operation with a synchronization call, such as `MPI_Wait`. In this case, the user waits on the lock acquisition (line 18) and combines the operation (line 19).

The `combine()` routine is similar to the one used by DSM-Synch. It traverses the work queue, calls the same `apply()` routine as DSM-Synch, and has the same `MAX_COMBINE` threshold on combining. If the threshold is reached, the thread enqueues its operation and releases the lock without executing its operation. This threshold part of the algorithm is omitted from Figure 3c for brevity (line 8 should not be executed if the threshold is reached). The final technical detail is with respect to ordering. Some MPI operations require total ordering, such as point-to-point and RMA accumulate operations. This implies that if two user threads synchronize to establish some ordered communication, the order must be preserved by the MPI library. To respect this constraint, the work queue must be totally ordered.

3.4.1 Totally Ordered Concurrent Queue. Communication that does not require ordering allows the LockQ model to leverage relaxed and potentially faster queues than does a totally ordered queue that is essential only for ordered communication. A totally ordered queue is more practical, however, because the MPI library programmer does not need to worry about ordering issues, regardless of the communication characteristic of the application. Here we describe a practical totally ordered queue that was used for LockQ. We note that in LockQ, at most one thread dequeues from a work queue. This design allows the dequeue routine to relax memory consistency and focus only on the consistency between enqueue operations and on possible races between an enqueue and a dequeue operation.

Figure 4 shows the implementation of SWP, a multiproducer single-consumer queue. This queue was inspired by the MCS lock [23], which maintains an implicit queue of waiting threads. The lock owner behaves as the only thread allowed to dequeue (wake up) the waiting thread next in line; SWP simply makes the queue explicit and removes unnecessary waiting loops. In SWP, `head` and `tail` point to a dummy node to decouple enqueue and dequeue operations. To enqueue an element, a thread prepares its node (lines 9–11), enqueues the node by first swapping it against the `tail` (line 12), and then updates the `next` pointer of the previous tail (line 13). During a dequeue operation, looking at the `next` field of the `head` of the queue is sufficient to detect an empty queue (line 18). If the queue is not empty, `next` points to the node that holds the data and will become the new `head` (becomes a dummy node) of the queue (line 22). The old `head` node gets freed (line 24).

Linearizability. SWP is not *linearizable* because the `enqueue` method does not admit a *linearization point* [15]; its effect is not visible to all methods at one point. The effects are globally visible at two points on lines 12 and 13. This makes the queue fragile since a thread that gets interrupted after executing line 12 and before executing line 13 renders the queue unusable until the thread is back. This implies that the queue is not robust against failures.

Wait-Freedom. Active enqueue/dequeue methods are wait-free since they finish executing in a bounded number of steps (even if an infinite delay at some thread is introduced between lines 12 and 13). In this case, however, the dequeue will always return empty because of the linearizability issue and renders the queue unusable.

This is not a setback for LockQ, however, since it admits any concurrent queue implementation. We have attempted to use the fast fetch-and-add wait-free queue by Yang et al. [29] (using the reference implementation by the authors), but its integration in LockQ and in our MPI library was not successful and requires further investigation (even with just two multithreaded MPI processes, the execution resulted in crashes and deadlocks). While SWP might seem inferior because of its lack of linearizability, we deployed this queue along with LockQ on production systems at large scale without any practical issues.

```
 1 init(Q) {
 2   node = alloc_node();          // Dummy node
 3   node->data = NULL;            // to avoid contention
 4   node->next = NULL;            // for the head and tail
 5   Q->head = node;               // from an enqueue and
 6   Q->tail = node;               // a dequeue
 7 }
 8 enqueue(Q, void *data) {
 9   node = alloc_node();
10   node->data = data;            // new tail node
11   node->next = NULL;            // nobody behind me yet
12   pred = SWAP(&Q->tail, node); // tail update visible
13   pred->next = node;            // link pred to my node
14 }
15 dequeue(Q, void **data) {
16   *data = NULL;
17   head = Q->head;
18   if (head->next == NULL) {     // queue empty
19     return;                     // return on empty queue
20   } else {                      // queue not empty, no race
21      next = head->next;         // between enqs and a deq
22      Q->head = next;            // head update visible
23      *data = next->data;
24      free_node(head);
25      return;
26   }
27 }
```

Figure 4: SWP: a multiproducer single-consumer concurrent queue. SWAP denotes an atomic swap operation. We highlight the lines that break linearizability.

3.4.2 Implementation of LockQ. The implementation in this paper relies on the SWP queue, which is efficient in practice because it ensures fast wait-free enqueue operations and total ordering. The other motivation for this queue is for a fair comparison with CSync and Per-VNI with MCS and to isolate the performance differences between them, since they are all based on MCS and rely on atomic swap operations.

LockQ preserves the theoretical property of asynchrony in MPI nonblocking calls. In practice, it allows better communication overlapping since the thread can return to the application to execute computational work. We also allow a thread to announce more than one request at a time, in order to exploit request pipelining and thus improve the latency hiding of the system. To avoid the overhead of announcing requests under no contention, a thread announces a request only on lock acquisition failure.

Costs of LockQ over CSync. In the absence of contention, LockQ is less costly and incurs similar costs as Per-VNI. Under contention, LockQ is slightly more costly than CSync: (1) it requires at least two atomic operations (one of `lock_tryacq` and one for enqueueing the work descriptor); and (2) since the same thread can pipeline multiple operations multiple times, descriptors require dynamic memory management instead of reusing the same queue node.

3.5 Nonblocking Progress Management

Thus far, we considered *communication initiation*-type of nonblocking MPI calls, such as `MPI_Isend`. There exists another type of nonblocking calls aimed for *communication progress*, such as `MPI_Test`, that allows testing for the completion of an operation. From an implementation perspective, `MPI_Test` could map to `MPI_Wait` (Figure 3c) but executing only a single iteration of the loop at line 17. This renders the call blocking on lock acquisition and wastes asynchrony. Moreover, by the time the waiting thread acquires the lock, the operation it is waiting for might have been completed by another thread, thus rendering the lock acquisition unnecessary. It also exacerbates contention because threads issuing operations on the same VNI as threads waiting on it compete for the same lock. An alternative approach would be to simply check for the request completion without progressing the LLN. This approach, however, violates the MPI progress requirement in the absence of asynchronous progress, as pointed out in Section 3.2.

We tackled this issue by adopting a nonblocking progress management not only for nonblocking MPI progress calls but also for managing internally progress on blocking calls, such as `MPI_Wait`. We use the progress loop in `MPI_Wait` to drive the discussion. A thread does a nonblocking lock acquisition attempt (instead of a blocking one at line 17 of any code snippet in Figure 3) and busy waits for the request to be completed on lock acquisition failure (`MPI_Test` would return to the user, but the same progress tight loop could occur in the user code as well). The winner of the lock progresses the VNI and might complete requests for the other threads, for which competing for lock acquisition at this level becomes unnecessary. Busy waiting on a lock acquisition attempt often translates into high cache traffic from threads competing for the same cache line, however, which outweighs the benefit of the nonblocking progress in practice. We thus reduce the pressure on the lock acquisition attempt using an exponential backoff, a known optimization in the context of locking. Here we apply it for MPI progress calls. For nonblocking progress calls, the current backoff value is memorized across successive calls in order to detect contention and use the backoff mechanism to reduce the frequency of lock acquisition attempts. This particular case occurs in the Graph500 benchmark evaluated in Section 4.4.

3.6 Progress and Asynchrony in LockQ

In this section, we informally reason about the progress properties of LockQ. First, we assume that the queue implementation is wait-free (see [29]) and that the LLN calls are nonblocking (needed for nonblocking MPI calls to be standard compliant). We distinguish three cases with respect to the type of nonblocking call being performed and the success of the lock acquisition. In the case of a progress call (e.g., `MPI_Test`), a lock acquisition failure returns immediately, thus preserving nonblocking behavior. In the case of a lock acquisition failure on a communication operation, the thread posts its work descriptor and returns immediately. Since the enqueue method is wait-free, this step is guaranteed to be nonblocking. In the case of lock acquisition success, the thread becomes a combiner and combines operations in the work queue. Executing individual work descriptors is nonblocking since the dequeue method is wait-free and the LLN calls are nonblocking. Since the number of operations executed by the combiner is bounded by `MAX_COMBINE`, it follows that the combiner returns in a bounded number of steps. This implies that LockQ can guarantee asynchrony for nonblocking MPI calls with practical assumptions (wait-free queue and nonblocking LLN routines).

3.7 Implementation

The new thread safety models have been integrated in the production-level MPICH implementation (version 3.3a2) based on the highly optimized CH4 device software layer [26]. Given that work descriptors and queue nodes are dynamically allocated on the critical path, we link all binaries against Tcmalloc [12] for scalable memory management.

Table 1: Platform specifications.

Microarchitecture	Skylake	Broadwell
Processor	Xeon Platinum 8180	Xeon 2695v4
Clock frequency	2.5 GHz	2.1 GHz
Sockets / NUMA nodes	2 / 2	2 / 2
Cores per NUMA node	28	18
HW threads per core	2	1
L2/L3 Cache size	1 MB/38.5 MB	256 KB/45 MB
Interconnect	Intel Omni-Path	Intel Omni-Path
Compilers	Intel 17.0.4	Intel 17.0.4
Linux kernel	3.10.0-693	3.10.0-693
Libpsm2/Libfabric	2.1/1.5.0	2.1/1.5.0

4 Evaluation

We describe in this section the performance evaluation of various thread safety models in terms of handling communication-intensive benchmarks as well as graph traversal and particle transport proxy applications.

4.1 Evaluation Platforms

Our evaluation was conducted on two commodity multicore clusters, as detailed in Table 1. The first platform is based on the Skylake microarchitecture featuring two 28-way core processors totaling 56 cores (112 hardware threads) per node. Intel Turbo Boost has been disabled to avoid dynamic frequency scaling from interfering with the experiments. The second platform is based on the Broadwell microarchitecture featuring two 18-way core processors totaling 36 cores per node. Both clusters interconnect nodes with the Intel Omni-Path fabric. MPICH on both clusters was built with the CH4 device over the libfabric network module and the provider based on the Intel Performance Scaled Messaging 2 (PSM2) library.

Our prototype implementation and evaluation method assumes a single network resource (i.e., one VNI), which is mapped to a libfabric endpoint. While using multiple VNIs would have been a valuable setting, the intricacies of how threads and communication patterns are mapped to VNIs are complex enough to warrant a separate study.

4.2 Experimental Method

We evaluate the implementations of six thread safety models, as briefly summarized below.

Global. Implements the model in Figure 1a.

Per-VNI. Implements the model in Section 3.2.

Offload. Implements the offload model in Figure 1c. It uses an identical queuing system as LockQ and spawns a dedicated communication thread running on CPU 0.

HMCS<N>USC. Implements one of the best lock management protocols for multithreaded MPI found in the literature [8] using HMCS [6, 7] as the high-throughput lock. N indicates the number of levels in this hierarchical lock, N=2 has one lock per NUMA node and one root lock for the entire machine, and N=3 adds a core-level lock for hardware threads sharing the same core. (N=3 will be evaluated only on the Skylake system since the Broadwell cluster supports only one hardware thread per core). USC stands for *user space condition variable* and captures the fact that this method wakes up threads with completed work directly ($O(1)$) instead of circulating the lock in $O(N)$, as demonstrated in [8]

CSync. Implements the model in Section 3.3.

LockQ. Implements the model in Section 3.4.

For a fair comparison, all these methods have been integrated in the same MPICH library, share the same critical paths except for the subtle differences in the way they manage thread safety, have been built with the same compiler toolchain, and are linked against the same libraries at runtime. Global, Per-VNI, and LockQ use MCS as the underlying locking algorithm (labeled LockQ-MCS). To showcase the practicality of LockQ over CSync, we also include results with Pthread mutex (labeled LockQ-MTX) as the underlying lock for this model. The MPI request object pool is protected by the global lock in the Global and HMCS<N>USC models and by a separate lock in the other models. In both cases, we maintain a per-thread local request cache to reduce contention for the lock, as done by Balaji et al. [5]. None of the methods use CPU 0 for the application threads. Thus, it is either used only by the communication thread in the offload model or unused.

Unless specified otherwise, the data points in the figure plots that follow are sample means of 10 runs augmented with lower and upper Gaussian confidence limits at 95% (error bars) based on the t-distribution, which were computed with the `smean.cl.normal` statistical function of the R `Hmisc v4.1-1` package.

4.3 Communication-Intensive Benchmarks

Here, we consider a simple two-node setting with one MPI process per node, various combinations of communication models (nonblocking two-sided, blocking two-sided, and one-sided communication) and degrees of concurrency. The experiments were conducted on the Skylake cluster to emphasize thread contention rather than node scaling.

Two-Sided Nonblocking. This model allows assessment of point-to-point message rate capabilities. One of the processes is the source of messages (calling `MPI_Isend`); the other is the sink (calling `MPI_Irecv`). We consider two variations depending on the degree of concurrency on each process, in other words, which communication operation is being concurrently executed.

One-Sided. This model focuses on RMA message rate capabilities using the `MPI_Put` operation. We use RMA passive communication by opening an epoch with `MPI_Win_lock`, issuing concurrently a large window of `MPI_Put` operations with multiple threads, synchronizing with the master thread, then using `MPI_Win_flush` to complete all the pending operations, and closing the epoch with `MPI_Win_unlock`.

Two-Sided Blocking. This model is a ping-pong test that targets point-to-point latency measurement. A single-threaded server receives messages from a multithreaded client and sends back acknowledgments. This ping-pong benchmark is implemented by using `MPI_Send` and `MPI_Recv` calls.

Figure 5 shows the message rate and latency with respect to the message size using 55 threads per MPI processes (one thread per core). In the message rate case, we observe that Offload dominates across the board, followed by LockQ, regardless of message sizes; but the gap between the methods is more pronounced for messages below 16 KB. We also notice that the code path that does not require waiting for a completion event (`MPI_Isend` with messages below 64 B is injected and completed immediately) shows higher absolute performance across methods and that LockQ competes with Offload. On the latency side, most methods perform well except Global. Here, the bottleneck lies in the progress management; except for Global, all the methods have ways to reduce progress management overheads and thus exhibit good latency. For example, Per-VNI, CSync, and LockQ use a nonblocking progress with exponential backoff, and the HMCS-based methods use an efficient $O(1)$ mechanism; Offload has the lowest overhead in this respect since application threads only busy wait on completion without any contention. We also observe that CSync underperforms at this scale especially with the latency benchmark.

More insight is brought to light when varying the degree of concurrency (Figure 6).

Sequential. This case has no contention. Global shows the best performance since it has the lowest overhead (only one atomic swap operation to acquire an MCS lock). Next are the HMCS-based models, followed by the Per-VNI and LockQ that incur around 10% or less overhead due to mostly performing more atomic operations. Offload and CSync are the worst here because of the overhead of unnecessary descriptor announcement. Offload is worse than CSync since the application thread passes the descriptor to the dedicated thread instead of executing it by itself.

Concurrency within a socket. The number of threads here is within 28 and shows that the most scalable methods maintain mostly flat performance, indicating little performance degradation (Offload and LockQ). CSync has the least clear trend: (1) it does not perform well when the degree of concurrency is low because the combining queue is shallow and reduces the effectiveness of combining; and (2) for `MPI_Isend` and `MPI_Put`, performance grows with the number of threads, thanks to better combining and no combining interruptions (no threads in the progress loop calling `dsm_acquire`). With `MPI_Irecv` and the latency benchmark, frequent combining interruptions happen and degrade its performance.

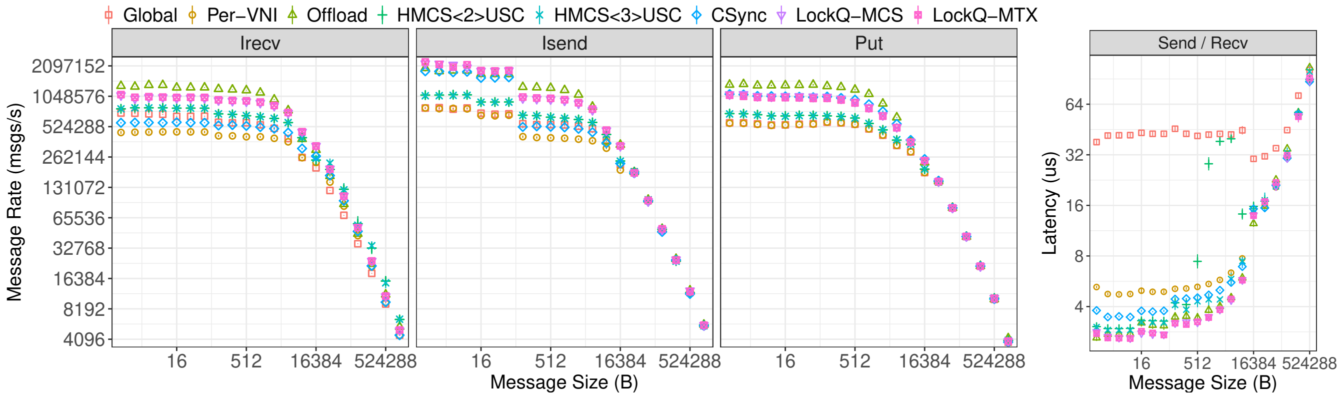
Concurrency across sockets. This corresponds to thread counts between 28 and 56. Offload is the only one that remains scalable (thanks to reduced remote memory references); the others degrade. The LockQ methods remain the next best performing.

Concurrency within cores. This corresponds to thread counts above 56, in which case two hardware threads can run on the same core. Offload remains scalable, whereas HMCS<3>USC improves scalability by exploiting core-level locality. At 110 thread count, the LockQ methods are still close to HMCS<3>USC despite being hardware agnostic.

We also observe overall that both MCS- and Pthread mutex-based LockQ methods perform similarly despite using completely different locking algorithms. This performance indicates the practicality of LockQ, which can be easily integrated to other systems, unlike CSync, which is reliant on the coupled lock-list data structure.

4.4 Breadth-First Search

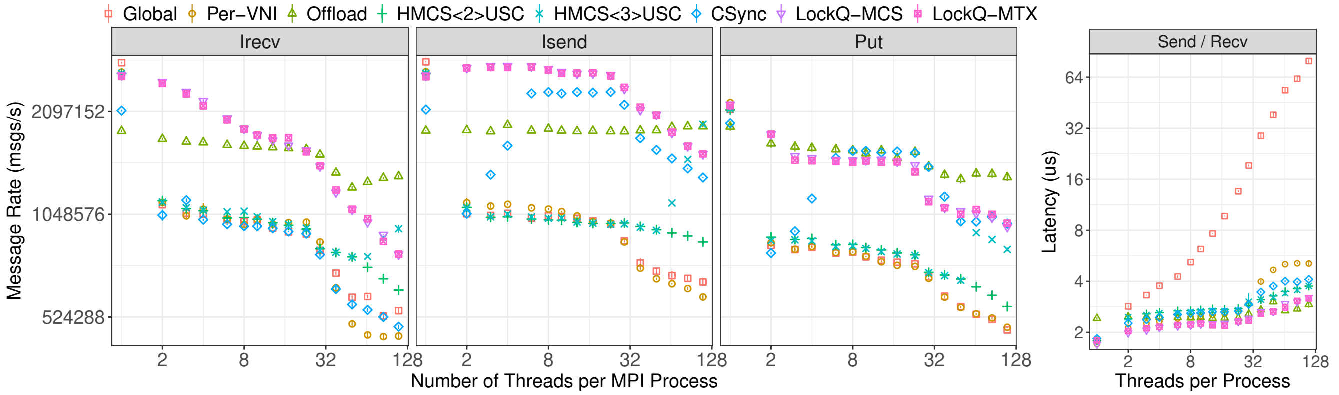
Parallel breadth-first search (BFS) implementations are characterized by irregular, dynamic, and sparse data exchanges (DSDEs). Here, a process communicates with a small neighborhood of processes that dynamically reshapes over time. $\mathcal{NBX}$ [18] has been demonstrated to be an efficient implementation of DSDEs and has been exploited by Amer et al. to implement a hybrid MPI+OpenMP BFS algorithm using the Graph500 benchmark as a baseline [2].



(a) Message rate

(b) Latency

Figure 5: Message rate and latency results with respect to the message size. The message rate results are grouped with respect to the MPI operation being performed concurrently by 55 threads (the title of each group is the operation that is performed with multiple threads).



(a) Message rate

(b) Latency

Figure 6: Message rate (with 64 B message) and latency results with respect to thread concurrency. Threads are bound to hardware threads in a way to prioritize filling cores close to each other. Only one hardware thread per core is used except when running with 110 threads. The results are grouped with respect to the MPI operation being performed concurrently by multiple threads (the title of each group is the operation that is performed with multiple threads).

This implementation handles computation and communication concurrently by multiple threads. The communication is implemented with nonblocking point-to-point calls, and the wait for their completion is performed with nonblocking progress calls (`MPI_Test`). These progress calls are issued when outgoing buffers are needed for reuse or when there is no local computation to be performed. In the Per-VNI, CSync, LockQ models, `MPI_Test` is implemented following the backoff-based nonblocking progress method described in Section 3.5. That is, if the backoff threshold has not been reached or the lock acquisition fails, the call behaves as if the communication has not yet completed. Since MPI requires that repeated calls to `MPI_Test` eventually succeed for completed operations, our thread safety models can satisfy this semantic by bounding the tolerated number of lock acquisition failures.

This section focuses on heavy concurrency; the goal of the new methods in this case is to perform as closely as possible to Offload without scarifying resources. Strong-scaling results with a graph scale of 32 (i.e., 2^{32} vertices) are shown in Figure 7. The performance is measured as the harmonic mean of the number of traversed edges per second. Offload performs the best, followed by the LockQ instantiations. CSync does improve on Per-VNI but remains inferior to LockQ, which performs up to 8% better and indicates the benefits of additional asynchrony and reduced overheads. LockQ significantly underperforms compared with Offload (up to 30%). Our analysis showed that the major bottleneck in these runs is waiting for completion, which significantly favors the Offload model (threads only wait on a local flag, which is optimal). Threads in the other methods are required to not only wait on a flag but also perform expensive lock acquisitions and network progress. For confirmation, by forcing threads in the Offload model to make progress (Offload-P in Figure 7) performance is brought down to the same level as LockQ. Synchronization counters [8] have the potential to reduce these overheads by electing one of the threads as a server with the others simply wait on a local flag. Unfortunately, this method is practical only for blocking calls and is unfit for nonblocking progress calls such as `MPI_Test`. Investigation is need to develop thread synchronization methods more suitable for nonblocking MPI progress calls.

4.5 SNAP: Particle Transport

SNAP[3] (SN Application Proxy) is a proxy application that emulates the MPI-based discrete ordinates neutral particle transport

[3] https://github.com/losalamos/snap

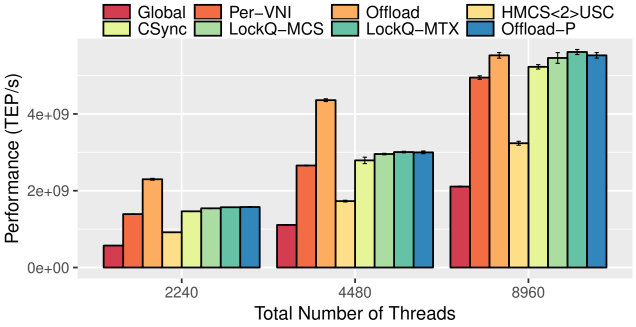
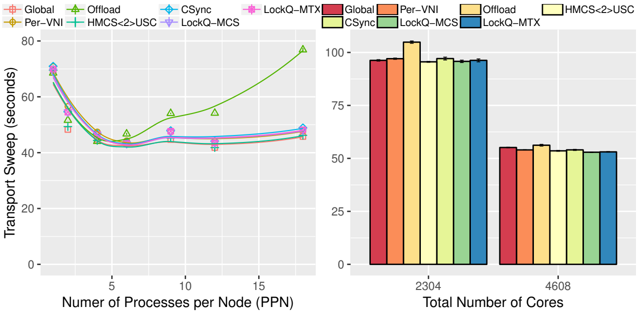


Figure 7: Graph500 strong-scaling results on the Broadwell cluster with 35 threads per MPI process with respect to the total number of threads.

Table 2: Input parameters for the SNAP runs. The parameters *npey* and *npez* depend on the number of MPI processes, and *nthreads* equals the number of threads per process, which depends on the number of PPN and whether a dedicated a thread is used (*).

`nthreads`	**variable**	`lx`	0.08	`src_opt`	3
`npey`	**variable**	`lz`	0.08	`timedep`	1
`npez`	**variable**	`ly`	0.08	`it_det`	0
`ndimen`	3	`nmom`	4	`tf`	1.0
`nx`	128	`nang`	32	`nsteps`	10
`ny`	**variable**	`ng`	72	`oitm`	40
`nz`	**variable**	`mat_opt`	1	`fluxp`	0
`iitm`	5	`scatp`	0	`angcpy`	1
`epsi`	0.0001	`fixup`	0	`ichunk`	16



(a) PPN Tuning **(b) Strong Scaling with $PPN = 6$**

Figure 8: Performance with the "Transport Sweep" stage of SNAP on the Broadwell cluster: (a) results with problem size $\{nx, ny, nz\} = \{128, 72, 64\}$ with respect to PPN; (b) strong scaling with the problem size $\{nx, ny, nz\} = \{128, 192, 192\}$ from 64 to 128 nodes. *npey* and *npez* have been computed to be as close to each other as possible.

application PARTISN. Although PARTISN solves the linear Boltzmann transport equation on multidimensional grids, SNAP does no actual physics but instead mimics the computational intensity, memory footprint, and communication patterns of PARTISN. The core of SNAP is characterized by an outer iterative loop that solves the flux over the energy domain using, typically, tens to hundreds of energy groups that are exploited for OpenMP thread-level parallelism. Parallelism is also exploited at the other spatial and angular dimensions. MPI-level data decomposition is performed along the spatial domain and traversed through sweeps along the discrete direction of the angular domain following the parallel Koch-Baker-Alcouffe wavefront method [4]. The traversal incurs data exchanges between MPI ranks using mostly two-sided point-to-point MPI communication. We also built SNAP with OpenMP to perform both communication and computation concurrently by multiple threads.

The baseline input problems used for the following experiments originate from one of regression tests that come with SNAP: `mms_src`. The following experiments study the various thread safety models under low thread contention while striking a balance between communication and computation. Under these circumstances, computation and coarse-grained communication (large data transfers) render fine-grained thread synchronization methods prone to overheads, and the simpler global locking models (Global and HMCS<2>USC) act as an upper bound on performance. Results are presented as time to solution of the "Transport Sweep" stage, which is the most time-consuming part of the application.

Because of the increasingly deep memory hierarchies in cluster nodes and difficulty in implementing NUMA-awareness in applications and dependent software layers, users often rely on spawning multiple processes per node (PPN) to reduce the diameter of the cache coherency traffic, which often improves overall parallel efficiency. Figure 8.a shows the result of tuning SNAP on 16 Broadwell nodes with a medium problem size $\{nx, ny, nz\} = \{128, 72, 64\}$. We observe that most methods achieve peak performance at PPN=6, then mostly stagnate, except Offload, which suffers significant degradation.

The performance improvement is a combination of better cache performance, from reducing remote memory references and cache coherency traffic that comes from accessing shared data among threads, and also by driving the network using multiple processes. The optimal PPN value represents a saturation point. PPN also represents the number of cores Offload sacrifices. Sacrificing up to four cores is tolerable in this case since the benefits outweigh the losses. Beyond four, however, it is counter productive and results in up to 70% performance degradation at $PPN = 18$.

Because of the neighborhood communication pattern in SNAP, the optimal PPN can be preserved regardless of the number of nodes provided that the problem size per process does not change significantly. For instance, by weakly scaling the previous small problem to $\{nx, ny, nz\} = \{128, 192, 192\}$ on 64 nodes, the behavior (Figure 8.b, 2,304 cores) is similar to that of the smaller-scale experiment (Figure 8.a, PPN=6). In a strong-scaling case, however, the optimal PPN is not portable because the system shifts to a more communication-intensive regime and reduces the degradation suffered in the Offload model (Figure 8.b, 4,608 cores).

These experiments with SNAP confirmed that the software combining methods show no significant degradation under low contention and equip the user with the same flexibility as the global locking methods without negative side effects. Offload, on the other hand, is constraining and is justified only when the application can spare CPU resources, a situation that is not always possible, especially with compute-intensive codes.

5 Concluding Remarks

We proposed new thread safety models for multithreaded MPI, CSync and LockQ, that leverage software combining to mitigate lock contention. These models were designed as protection mechanisms around independent VNIs, abstract objects meant to capture independent network resources. CSync shows significant scalability improvements over traditional locking but suffers from lack of asynchrony and overheads in the absence of contention. LockQ is a significant step forward in both performance and practicality; it preserves asynchrony of MPI nonblocking calls, eliminates unnecessary offloading operations, and is more flexible and easier to integrate in MPI libraries than CSync is. The comparative evaluation of these models over an Omni-Path fabric shows significant improvement over existing contention-reducing and contention management methods. LockQ also shows competitive performance against software offloading but without sacrificing computational resources for dedicated communication threads while avoiding undesirable software offloading overheads at low contention.

The paper focused only on the highly serialized case, in order to stress the combining synchronization aspect of the thread safety model and evaluate its contention mitigation capabilities. We are currently widening the scope of the study to multiple VNIs and focusing on contention avoidance. This work was also performed under the *totally ordered communication* constraint, thus requiring a totally ordered queue for the LockQ model. This constraint can be relaxed in favor of more parallelism at the announcing threads by posting operations at different queues. Such a capability could be leveraged for hardware awareness (e.g., NUMA-aware) or simply for contention reduction.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Laboratory Computing Resource Center (LCRC) and by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

References

[1] Abdelhalim Amer, Huiwei Lu, Pavan Balaji, Milind Chabbi, Yanjie Wei, Jeff Hammond, and Satoshi Matsuoka. 2019. Lock Contention Management in Multithreaded MPI. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 12.

[2] Abdelhalim Amer, Huiwei Lu, Pavan Balaji, and Satoshi Matsuoka. 2015. Characterizing MPI and Hybrid MPI+Threads Applications at Scale: Case Study with BFS. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1075–1083.

[3] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015. MPI+ Threads: Runtime Contention and Remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. 239–248.

[4] Randal S Baker and Kenneth R Koch. 1998. An S_n Algorithm for the Massively Parallel CM-200 Computer. *Nuclear Science and Engineering* 128, 3 (1998), 312–320.

[5] Pavan Balaji, Darius Buntinas, D. Goodell, W. D. Gropp, and Rajeev Thakur. 2010. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *International Journal of High Performance Computing Applications (IJHPCA)* 24 (2010), 49–57.

[6] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-Level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. 215–226.

[7] Milind Chabbi and John Mellor-Crummey. 2016. Contention-Conscious, Locality-Preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. 22:1–22:14.

[8] Hoang-Vu Dang, Sangmin Seo, Abdelhalim Amer, and Pavan Balaji. 2017. Advanced Thread Synchronization for Multithreaded MPI Implementations. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 314–324.

[9] Gábor Dózsa, Sameer Kumar, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Joe Ratterman, and Rajeev Thakur. 2010. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'10)*. Springer-Verlag, Berlin, Heidelberg, 11–20.

[10] Wataru Endo and Kenjiro Taura. 2018. Parallelized Software Offloading of Low-Level Communication with User-Level Threads. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. ACM, 289–298.

[11] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. 257–266.

[12] Sanjay Ghemawat and Paul Menage. 2009. Tcmalloc: Thread-Caching Malloc.

[13] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D Russell, Howard Pritchard, and Jeffrey M Squyres. 2015. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI'15)*. 34–39.

[14] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 355–364.

[15] Maurice Herlihy and Nir Shavit. 2011. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[16] Nathan Hjelm, Matthew GF Dosanjh, Ryan E Grant, Taylor Groves, Patrick Bridges, and Dorian Arnold. 2018. Improving MPI Multi-Threaded RMA Communication Performance. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 58.

[17] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. 2013. MPI+MPI: A New Hybrid Approach to Parallel Programming with MPI plus Shared Memory. *Computing* 95, 12 (2013), 1121–1136.

[18] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. 2010. 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)*. 159–168.

[19] Krishna Kandalla, Peter Mendygral, Nick Radcliffe, Bob Cernohous, David Knaak, Kim McMahon, and Mark Pagel. 2016. Optimizing Cray MPI and SHMEM Software Stacks for Cray-XC Supercomputers based on Intel KNL Processors. *Cray User Group* (2016).

[20] Alex Kogan and Erez Petrank. 2011. Wait-Free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. 223–234.

[21] Sameer Kumar, Amith R Mamidala, Daniel A Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, et al. 2012. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. 763–773.

[22] Peter Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue Locks on Cache Coherent Multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*. IEEE, 165–171.

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

[24] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. 1999. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Vol. 16. Citeseer.

[25] GF Pfister, WC Brantley, DA George, SL Harvey, WJ Kleinfelder, KP McAuliffe, EA Melton, VA Norton, and J Weiss. 1985. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing: August 20–23, 1985*. IEEE Computer Society Press, Washington, DC.

[26] Ken Raffenetti, Abdelhalim Amer, Lena Oden, Charles Archer, Wesley Bland, Hajime Fujita, Yanfei Guo, Tomislav Janjusic, Dmitry Durnov, Michael Blocksome, Min Si, Sangmin Seo, Akhil Langer, Gengbin Zheng, Masamichi Takagi, Paul Coffman, Jithin Jose, Sayantan Sur, Alexander Sannikov, Sergey Oblomov, Michael Chuvelev, Masayuki Hatanaka, Xin Zhao, Paul Fischer, Thilina Rathnayake, Matt Otten, Misun Min, and Pavan Balaji. 2017. Why is MPI So Slow?: Analyzing the Fundamental Limits in Implementing MPI-3.1. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. 62:1–62:12.

[27] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI'15)*. 40–43.

[28] Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, Jeff R. Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint Joó. 2015. Improving Concurrency and Asynchrony in Multithreaded MPI Applications Using Software Offloading. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. 30:1–30:12.

[29] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-Free Queue as Aast as Fetch-and-Add. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 16.

[30] Pen-Chung Yew, Nian-Feng Tzeng, et al. 1987. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Trans. Comput.* 100, 4 (1987), 388–395.