Lock Contention Management in Multithreaded MPI

ABDELHALIM AMER, HUIWEI LU, and PAVAN BALAJI, Argonne National Laboratory, USA
MILIND CHABBI, Hewlett-Packard Labs, USA
YANJIE WEI, Shenzhen Institute of Advanced Technologies, Chinese Academy of Sciences, China
JEFF HAMMOND, Intel, USA
SATOSHI MATSUOKA, Tokyo Institute of Technology, Japan

In this article, we investigate *contention management* in lock-based thread-safe MPI libraries. Specifically, we make two assumptions: (1) locks are the only form of synchronization when protecting communication paths; and (2) contention occurs, and thus serialization is unavoidable. Our work distinguishes between lock acquisitions with respect to work being performed inside a critical section; *productive* vs. *unproductive*. Waiting for message reception without doing anything else inside a critical section is an example of unproductive lock acquisition. We show that the high-throughput nature of modern scalable locking protocols translates into better communication progress for throughput-intensive MPI communication but negatively impacts latency-sensitive communication because of overzealous unproductive lock acquisition. To reduce unproductive lock acquisitions, we devised a method that promotes threads with productive work using a generic two-level priority locking protocol. Our results show that using a high-throughput protocol for productive work and a fair protocol for less productive code paths ensures the best tradeoff for fine-grained communication, whereas a fair protocol is sufficient for more coarse-grained communication. Although these efforts have been rewarding, scalability degradation remains significant. We discuss techniques that diverge from the pure locking model and offer the potential to further improve scalability.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; **Concurrent algorithms**; • **Software and its engineering** → **Parallel programming languages**; **Distributed programming languages**; **Concurrent programming languages**;

Additional Key Words and Phrases: MPI, threads, runtime contention, critical section

ACM Reference format:
Abdelhalim Amer, Huiwei Lu, Pavan Balaji, Milind Chabbi, Yanjie Wei, Jeff Hammond, and Satoshi Matsuoka. 2019. Lock Contention Management in Multithreaded MPI. *ACM Trans. Parallel Comput.* 5, 3, Article 12 (January 2019), 21 pages.
https://doi.org/10.1145/3275443

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, by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357, by the JSPS KAKENHI Grant No. 23220003, and by the Science Technology and Innovation Committee of Shenzhen Municipality under Grants No. JCYJ20160331190123578 and No. GJHZ20170314154722613.

Authors' addresses: A. Amer and P. Balaji, Argonne National Laboratory; emails: halim.amer@acm.org, balaji@anl.gov; H. Lu, Tencent; email: lvhuiwei@gmail.com; M. Chabbi, Scalable Machines Research; email: milind@ScalableMachines.org; Y. Wei, Shenzhen Institute of Advanced Technologies, Chinese Academy of Sciences; email: yj.wei@siat.ac.cn; J. Hammond, Intel; email: jeff_hammond@acm.org; S. Matsuoka, Tokyo Institute of Technology; email: matsu@is.titech.ac.jp.

2329-4949/2019/01-ART12 $15.00
https://doi.org/10.1145/3275443

1 INTRODUCTION

The Message Passing Interface (MPI) [27] has been the de facto standard for programming high-performance computing (HPC) systems for more than two decades. With the advent of multicore and many-core systems, however, relying purely on MPI to handle both internode and intranode parallelism is being questioned [3, 24]. Indeed, per core resources (e.g., memory capacity and network endpoints)[1] are becoming scarcer, implying that programming systems that encourage collaboration within the same node to efficiently share those resources are more desirable than systems that are inherently competitive. Unfortunately, MPI's default private address-space model, where processes view resources as dedicated, falls into the latter category. As a result, applications, libraries, and languages centered on MPI are moving to hybrid models that exploit MPI for coarse-grained parallelism and another programming system more suitable for shared memory. Among these models, MPI+threads is predominant, with MPI+OpenMP being the most widely used hybrid programming model.

In order to support applications that require concurrent multithreaded accesses to MPI, thread safety is necessary but not sufficient. The library must also guarantee that a thread in a blocking call (e.g., waiting for message reception) does not obstruct the progress of other threads. Because of the complex MPI progress semantics and the stateful nature of MPI processes, however, most of these libraries satisfy thread compliance by using locks for simplicity. Specifically, this study makes two assumptions. The first assumption is that locks are the only form of synchronization to manage critical sections. This is a reasonable assumption because locks have been the prominent form of synchronization for implementing thread-safe programs and MPI libraries are no exception. In particular, most MPI libraries follow a coarse-grained locking approach; that is, long code paths (e.g., operations on the same network endpoint) are protected with the same lock. Indeed, such simple thread-safety models avoid the intricacies associated with fine-grained locks and lock-free data structures[2]; programmers work on long code paths within critical sections, thereby simplifying the task of leaving the state of an MPI process consistent and ensuring deadlock freedom across lock acquisitions.

The second assumption in this study is that contention takes place, and thus serialization is unavoidable. The simplicity of this thread-safety model, however, potentially incurs significant performance penalties in the form of serialization and contention management overheads. Significant effort has been invested during the past decade to improve threaded accesses to MPI libraries and overcome contention, which remains a significant obstacle in achieving scalable performance. Most work has focused on *contention-avoidance* techniques, such as fine-grained locking [7] and exploiting independent communication paths [16]; and more investigations are under way. While these directions are significant, however, they do not guarantee contention freedom. These techniques offer no remedy when contention is unavoidable, a prominent issue in current MPI libraries.

In this article, we investigate the practical limits of reducing overheads when managing contention to the same *communication path* from multiple threads. We consider a communication path to be a code path operating on a single network resource, such as a communication context or endpoint. Given our assumption that a communication path is protected by a lock, our study focuses exclusively on the relation between the locking protocol and contention management on the communication path; that is, other forms of synchronization, such as condition variables, are left out of this study. Given the large number of locking protocols in the literature, we focus on a carefully selected set that offers various tradeoffs between fairness, latency, and throughput: Pthread

[1] An endpoint is a system resource, such as a network socket, within a node that is addressable and used for sending and receiving data. A network interface can expose multiple endpoints.

[2] For example, deadlocks stemming from lock acquisition ordering issues when using fine-grained per object locks.

mutex [19] (with its unique protocol that combines a user space test-and-set operation with OS-blocking), MCS [26] (representative of fair first-in, first-out, or FIFO, protocols), and HMCS [11, 12] (representative of modern high-throughput protocols). We analyzed the relationship between the locking protocols and progress on a communication path, using microbenchmarks as well as a real-world code.

One of our primary findings was the important distinction between lock acquisitions with respect to work being performed on a communication path. We distinguish *productive* lock acquisitions as those that execute an operation within a bounded number of steps and *unproductive* as those that execute an operation that involves waiting for an unbounded number of steps without doing anything else. Our analysis showed that the high-throughput nature of modern scalable locking protocols translates into better communication progress for throughput-intensive MPI communication but negatively impacts latency-sensitive communication, which stems from overzealous unproductive lock acquisition.

Based on these initial findings, we developed a new method that prioritizes productive work over potentially unproductive lock acquisitions in order to improve the overall progress. This method leverages a new protocol that has two priority levels and takes arbitrary locking subprotocols for each level; that is, the lock has a generic interface that allows it to select existing protocols to be used at each priority level. With this method we were able to readily evaluate various protocol combinations. Our results showed that using a high-throughput protocol for less productive code paths significantly degrades latency regardless of the message sizes. Furthermore, adopting a high-throughput protocol for productive work and a fair protocol for less productive code paths exhibits the best tradeoff for fine-grained communication, whereas a fair protocol is sufficient for more coarse-grained communication. Nevertheless, despite these improvements to communication progress, our investigation indicated that relying exclusively on the locking protocol has scalability limits that cannot be overcome with this simple thread-safety model. We briefly discuss more elaborate models that combine locks with other mechanisms that have the potential to overcome these limits.

2 BACKGROUND

We begin this section with a discussion of how threads interoperate with MPI. Next, we present the basic thread-safety model being investigated. Then we describe the set of locking protocols studied throughout the rest of the article.

2.1 MPI and Interoperability with Threads

MPI is a library specification for moving data between address spaces (processes). Application threads can access the library sequentially or concurrently if the library implementation supports this capability. Given the performance and correctness implications of allowing concurrent multithreaded access, MPI defines the following rules. First, concurrent access is optional; that is, an MPI library is free to allow only single-threaded access while being compliant with the specification. Second, multiple levels of threading support are defined and represented as monotonically increasing integer values; lower levels restrict the degree of interoperability of application threads with MPI, while the highest level (i.e., MPI_THREAD_MULTIPLE) allows threads to concurrently access MPI. Applications can query the MPI library at initialization time for support of the desired level. This approach allows applications to avoid unnecessary overheads from supporting higher levels. The MPI_THREAD_MULTIPLE level, which this work focuses on, imposes thread safety as one of the main requirements for concurrent threading compliance. The other main requirement is aimed at ensuring progress: a thread in a blocking call must not block other threads from making progress.

(a) `MPI_Isend` path (b) `MPI_Wait` path decomposition

Fig. 1. Code path analysis of nonblocking and blocking MPI call representatives. Operations between brackets ("[ ]") can be omitted depending on the MPI library and network hardware. The numbers (1), (2), and (3) in circular shapes indicate creation, completion, and freeing operations on request objects.

2.2 Thread-Safe Single-Communication-Path MPI

Since this study assumes that contention takes place, it is sufficient to consider a single communication path taken by all threads. The simplest lock-based thread-safety model in this case would protect any API routine with a global lock while yielding within blocking calls to respect the progress requirements. Figure 1 shows how a representative nonblocking MPI call (`MPI_Isend`) and a representative blocking call (`MPI_Wait`) would be implemented in this model. `MPI_Isend` issues a send operation, returns immediately (nonblocking), and updates `req` with a request handle to the user to track the progress of the operation. `MPI_Wait` blocks the caller and waits for the completion of the operation corresponding to the request handle the user has passed (`req`). `L` is a lock that protects a communication path; since we assume a single path for simplicity, all MPI calls are protected by `L`. Thread safety for nonblocking calls is straightforward; we need only to acquire and release `L` at the entry and exit of the call, respectively (Figure 1(a), lines 2 and 8). For blocking calls, a thread waiting within the loop at line 4 of Figure 1(b) must relinquish `L` eventually in order to allow other threads to execute their critical sections (lines 10–12); failing to do so might result in deadlock.

In Figure 1(b) we also highlight two operations that are important but that can be optional depending on the target system. The first, `network_progress` (line 6), pokes the network hardware to make progress on outstanding operations, including the operation corresponding to `req`. This manual progress model is predominant in practice and compulsory on most network hardware, which does not support asynchronous progress. In either case, once the operation is executed, the `network_progress` call or the asynchronous progress mechanism will mark `req` as completed, in order to notify the MPI library. The second operation politely *yields* (line 11) to allow other threads to run on the same CPU. This operation is often omitted, however, because it incurs expensive system calls (e.g., `sched_yield` on Linux systems) on the critical path and because oversubscribing threads on the same CPU is a rare occurrence on HPC systems.

We consider this coarse-grained thread-safety model robust against common thread safety issues, such as deadlocks and data corruption. This design, however, is prone to high serialization and contention overheads.

2.3 Description of the Locking Protocols

We focus on three protocols that offer different tradeoffs between fairness, latency, and throughput: Pthread mutex, MCS, and HMCS. We also include a description of ticket for its relevance in Section 4.1 as a two-way concurrency protocol. *Latency* here refers to the latency of passing ownership of a lock, while *throughput* refers to the number of lock acquisitions per time unit. Although several hierarchical locking protocols have been proposed [1, 9, 12, 14, 15, 25] and are worth including in our study, we believe HMCS is a sufficient and solid representative of their high-throughput nature.

Pthread Mutex. Several software packages, including many MPI libraries, rely on the system-provided Pthread mutex to ensure thread safety. Because our platforms are Linux-based clusters, the Pthread mutex API is provided by the Native POSIX Thread Library (NPTL) [19]. In the remainder of this article, we will refer to the NPTL mutex protocol as *Pthread mutex* (or Mutex). Pthread mutex is blocking (a lock acquisition attempt might block the calling thread in the OS kernel when the lock is held by another thread), and its protocol follows a two-step process. First, in the user space, a thread tries to acquire the lock by using hardware atomics, usually with a test-and-set operation. Second, in the kernel space, if the acquisition fails, the thread goes to sleep in the kernel using the `FUTEX_WAIT` operation of the `futex` (fast user space mutex) system call [18, 20]. When the lock holder leaves the critical section, it wakes up blocked threads, usually at most one, with the `FUTEX_WAKE` operation. The awakened threads return to the user space and compete again for the lock by using hardware atomics. Although some variations might exist across hardware, Linux kernels, and NPTL versions, this two-phase implementation holds for most platforms. As we demonstrated in our prior work [4, 5], lock acquisition is heavily dictated by the user space atomic operation and results in unfair lock ownership passing.

MCS. The MCS protocol (named after its inventors Mellor-Crummey and Scott) builds a queue out of waiting threads and ensures a FIFO lock-passing ordering. Each participant brings a local node to be atomically swapped with the node previously in line, thus executing fully in user space. A participant busy waits on its private node for the owner to pass ownership of the lock. These properties ensure fairness and avoid competition for the same cache line from all waiters that would cause significant cache traffic and long lock-passing latencies. The strict FIFO protocol has the unfortunate side effect, however, of forcing lock ownership circulation across all participants with no regard to locality of reference, which results in overall low performance on deep-memory hierarchies.

HMCS. The Hierarchical MCS protocol aims at overcoming the performance shortcomings of the original flat MCS lock. Participants in HMCS prioritize ownership passing between neighbors in the memory hierarchy before passing the lock to remote participants. This approach improves locality of reference, effectively amortizes expensive ownership passing between remote participants, and achieves an overall low latency and high throughput when tuned appropriately. HMCS offers knobs for controlling thresholds of local passing to achieve the desired performance vs. fairness tradeoff.

Ticket. The ticket protocol uses two shared counters. The first counter is incremented atomically by each thread to get a unique ticket and wait its turn, and the second counter is incremented at release time to grant critical section access to the next thread. On cache-coherent systems, the ticket protocol is known to suffer from heavy cache-coherency traffic because of the shared-global state of the lock. For low degrees of concurrency, this protocol performs similarly to MCS. Because of its simplicity, we leverage ticket in Section 4.1 in a case that guarantees that at most two threads compete for the lock.

3 PROGRESS ASPECT IN MULTITHREADED MPI

To shed light on the intricacies of lock contention for a communication path, we first investigate the relationship between lock management and communication progress in the context of the locking protocols described above. We then present an alternative locking protocol that offers better progress properties, and we evaluate both its effectiveness and its shortcomings.

3.1 MPI Library Use Case: MPICH

We use MPICH v3.2 [2] on commodity clusters that use a single coarse-grained critical section. MPICH v3.2 exploits only a single communication context, or endpoint, per process. Hence, we consider this MPI library as having a single communication path for all threads. This path is protected by default with a Pthread mutex following the thread-safety model described in Section 2.2. As a result, all API calls are serialized, and the message rate is bounded by single-threaded performance. The goals of the thread-safety model in this case are not only to guarantee correctness of execution (thread safety and satisfy the progress requirements) but also to reduce or avoid any performance degradation below that of a single-threaded execution.

3.2 Communication Progress in Single-Threaded MPI

The term *progress* in the context of MPI has never been defined formally by the MPI community. Understanding the performance implications of lock management in MPI, however, requires a quantitative method in order to correlate software changes with progress efficiency. For instance, the rate at which `MPI_Isend` calls get translated into the network hardware interface[3] correlates with the rate at which messages get injected into the network and indicates how efficiently progress is being made on the sender side. In this article, we rely on the notion of *progress efficiency* when executing a code path inside the MPI library. Our quantitative metric for measuring progress on a given code path relies on counting the number of *productive* operations executed on that path. We consider *productive* any operation that changes the state of the communication system (composed mostly of the MPI library data and user buffers). Waiting or sleeping operations, as a result, are considered *unproductive*. For instance, busy waiting for message reception without doing anything else (e.g., without servicing requests in the meantime) does not change the state of the system and is thus unproductive. Tracking every operation being executed is impractical, however, given their large number. Instead, we track a few key operations in isolation and use well-understood benchmarks to infer the amount of progress that is being made.

The following sections present a progress analysis of carefully designed latency-bound and throughput-bound benchmarks. We found that tracking operations on requests (creation, completion, and destruction operations) on the code paths is simple and is sufficiently insightful. For instance, in the thread-safety model of Figure 1, the numbers 1, 2, and 3 in circular shapes indicate creation, completion, and freeing operations on request objects, respectively. Thus, our progress metrics rely on counting these operations.

3.3 Progress in Lock Management

How lock *acquire* and *release* protocols (which we refer to as *lock management*) are implemented influences the order and waiting time before a thread accesses the critical section. From these protocols, one can infer whether a lock allows all threads *fair* access to the critical section or whether lock operations are nonblocking (e.g., releasing a lock is nonblocking if it takes a bounded

[3]Each network hardware offers a low-level network interface. Since MPI is a portable specification, implementations have to express and map higher level MPI abstractions (e.g., `MPI_Isend`) into the corresponding low-level interface.

number of steps). As we demonstrate in the following section, progress in lock management has a strong impact on progress in MPI communication.

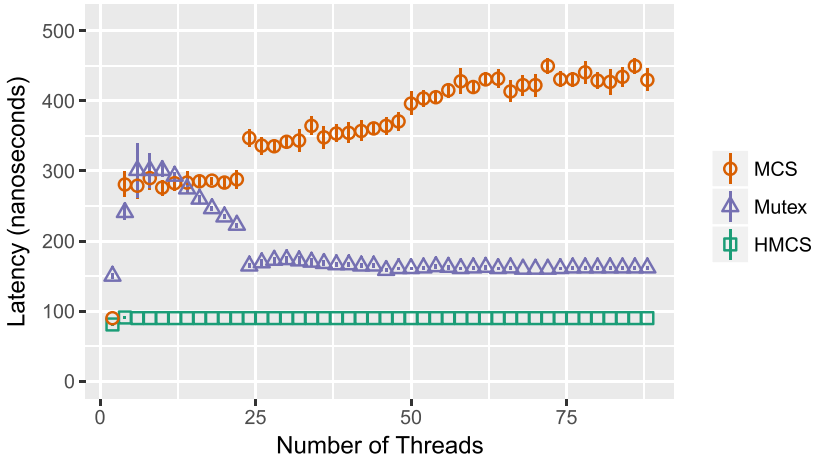
3.4 Effect of Lock Acquisition Progress on MPI Communication

In this section, by analyzing the progress on MPI communication of various lock implementations with diverse lock management protocols, we gain insight into this subtle interaction. This insight will allow us to build a more complex protocol that leverages the interaction to enhance communication progress.

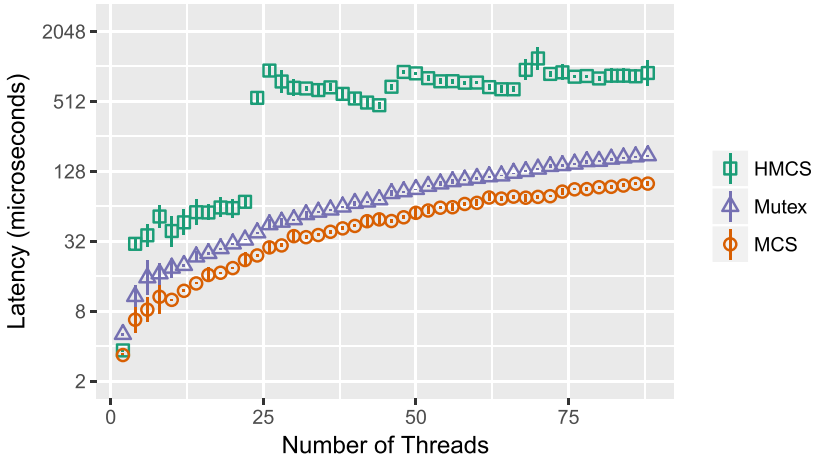
3.4.1 Motivating Example. We compared lock acquisition latency between two threads against MPI message latency between two multithreaded processes, in order to gauge the possibility of correlation between the two performance metrics. Our experiments were conducted on 88-way hardware-threaded Intel Broadwell nodes connected through an Intel Omni-Path network fabric (more architecture details are provided in Section 5.1). Figures 2(a) and (b) show the results of the two experiments with respect to the number of threads contending for a critical section. We observe that latencies vary significantly across locking protocols. Perhaps the most intriguing part is the performance results between the experiments: the lower the latency to acquire a lock, the higher the MPI latency achieved. This result is somewhat counterintuitive because one would expect that faster access to the critical section would allow better performance on the communication path. These raw performance results, unfortunately, do not provide hints as to the cause of such discrepancies, and hence a more thorough inspection is needed. To this end, we rely on the *path efficiency* metric described in Section 3.2. Specifically, we measure the amount of work being achieved on the main and the progress paths with respect to the lock acquisition being used. Furthermore, we perform our experiments in different regimes, *latency* and *throughput*, to cover the communication-intensive cases found in practice.

3.4.2 Progress in Latency-Oriented Regimes. In an extreme latency-oriented regime, a thread would wait for a single communication operation to get executed before issuing another operation. Pipelining multiple operations would allow hiding the latency of individual operations, but in a latency-oriented regime there is little to no latency hiding. To build a test case that emphasizes this behavior, we implemented a ping-pong benchmark between two MPI processes, a *server* and a *client*. The server is single-threaded and services the requests from the client one at a time. Each request is a message sent by the client and waits for an acknowledgment from the server; the message size for both operations is the same. Each thread on the client side sends a single message and waits for an acknowledgment message from the server. From a progress perspective, the server cannot listen to the next request until the current one is satisfied. On the client side, each thread has to wait for an acknowledgment before proceeding to the next request. This *strict* progress requirement translates into pressure on communication latency; for either party to proceed to the next request, it has to wait for a round-trip message to complete.

The results presented in Figure 2(b) were obtained with this benchmark when scaling the number of threads on the client side. We observe that the latency to service a single request grows with the number of threads involved at the client and reaches two orders of magnitudes higher than single-threaded latency at full concurrency across lock implementations. This result indicates significant scalability issues. We also observe that the lower the lock passing latency, the longer the corresponding request latency. To shed light on this issue, we investigated the path efficiency at the client side. More specifically, we instrumented the MPI library to record the number of times the main path and the progress path have been taken as well as the amount of work that has been produced. This benchmark uses `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` routines that allow us to track work being accomplished as a function of the operations on requests.



(a) Lock acquisition latency. At each iteration, a thread acquires a global lock, touches randomly 10 cache lines, and releases the lock. Each experiment runs 2^{22} iterations scheduled cooperatively among threads.



(b) Multithreaded MPI one-way latency for 64 B messages. The benchmark does a ping-pong communication between two processes; one multithreaded and the other single threaded. Each experiment runs 10K iterations scheduled cooperatively among threads.

Fig. 2. Lock acquisition and multithreaded MPI latency. Each experiment was run 10 times. The results tested positive for normality; thus we report the arithmetic mean (data points) with a confidence interval of 95% (error bars).

Figure 3 shows the efficiency of the main and progress paths (as defined in Section 3.2) with respect to the number of threads at the client side. The profiling overhead in this experiment was considerable for small concurrency (up to 100%) but acceptable at high concurrency (roughly 20%).[4] We observe that the efficiency of the main path is insensitive to the number of threads or the

[4]The overhead numbers were obtained by comparing performance numbers of running the same experiment with and without profiling enabled.

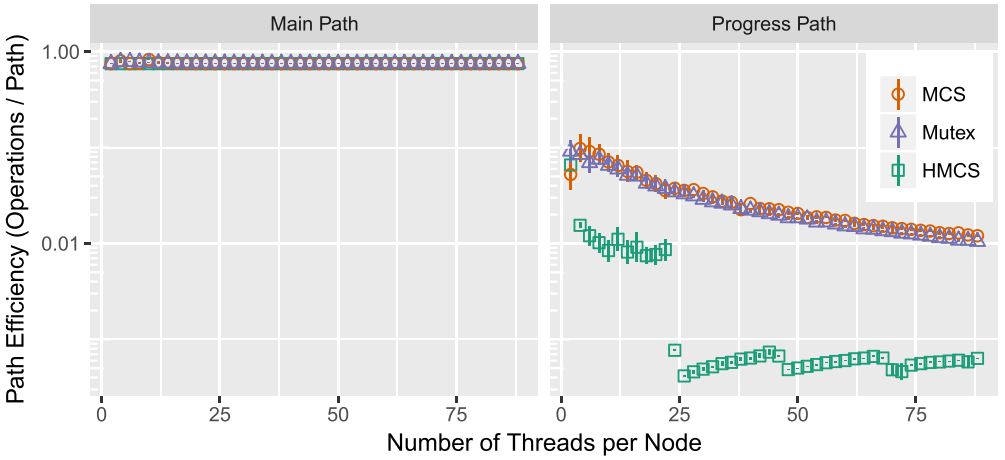


Fig. 3. Progress efficiency per execution path in a latency-oriented regime. Each data point is the arithmetic mean of 10 runs with a confidence interval of 95% (error bars).

locking protocol. This result indicates that productive operations are being executed for most lock acquisitions at this level. Looking at the progress path, however, we observe scalability issues that correlate with Figure 2(b). In particular, the latency with HMCS is significantly worse than with the other locking protocols. What sets HMCS apart from the others, especially MCS, is its ability to circulate the lock among neighbor threads. Unfortunately, the active neighborhood where the lock is being circulated might not be waiting for the operation that the server had issued. For instance, if the server has sent an acknowledgment, only threads waiting for a message would consume it; threads that are waiting for the request sent to be matched at the server would not consume such message. As a result, this neighborhood-passing protocol can result in a significant amount of unproductive work on the progress path. In particular, the step-like pattern when crossing NUMA boundaries (degree of concurrency is 24, 46, or 68) is an artifact of this neighborhood-passing protocol; the neighborhood of a newly explored NUMA domain starts small, then grows to fill that domain. When the neighborhood on a domain is small, the chances of productive work are lower than when the NUMA domain is fully exploited. This step-like pattern shows up in the remainder of this article for the same reasons mentioned here.

Before suggesting alternative protocols, we analyze how these existing protocols perform in throughput-oriented regimes.

3.4.3 Progress in Throughput-Oriented Regimes. In a throughput-oriented regime, sufficiently pipelined operations exist to allow latency hiding, in which case throughput is the bounding factor. We created a benchmark that stresses this aspect by maintaining two MPI processes: a *source* and a *sink*. Pipelined messages flow from the source process to the sink process. Both processes are multithreaded. Using the same approach as in the preceding section, we inspected the progress efficiency on both paths at the level of the sink process. The profiling overhead in this experiment was considerable for small concurrency (up to 200%) but acceptable at high concurrency (roughly 20%). As shown in Figure 4, the lock implementation that stands out is HMCS with the highest efficiency on both paths. This indicates that the high throughput of HMCS matches well this type of regime. We observe, however, that the efficiency still drops on both paths, indicating scalability issues as well.

We conclude from this path efficiency analysis that a locking protocol that manages contention for a communication path that suits all situations does not exist. In the following section, we

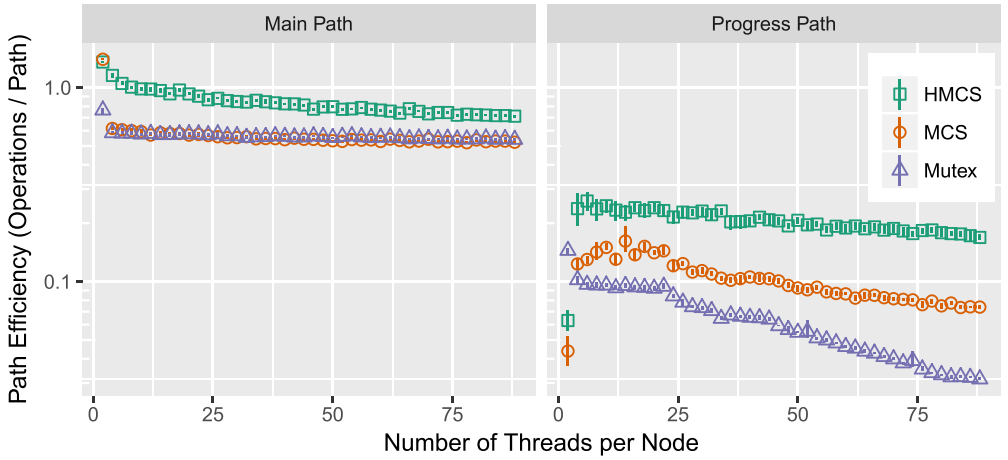


Fig. 4. Progress efficiency per execution path in a throughput-oriented benchmark. Each data point is the arithmetic mean of 10 runs with a confidence interval of 95% (error bars).

present a novel locking protocol that aims at providing better progress properties when managing contention for a communication path.

4 IMPROVING PROGRESS ON COMMUNICATION THROUGH ADAPTIVE LOCKING

Determining *a priori* which thread is going to make progress after getting the lock is not a trivial task. The situation depends in many cases on external events such as message reception. The previous analysis, however, suggested that zealously circulating the lock in the progress path is often unproductive. In this section, we exploit this information to propose a locking protocol that prioritizes the main path over the progress path.

4.1 Generic Construction of a Two-Level Priority Lock

Given the previous observations, we decided to lower the priority of progress waiters in favor of threads on the main path. To this end, we developed a two-level priority locking protocol called TLP. The principle behind this protocol is to use a *filter* to allow only high-priority threads when they exist. Each priority class can adopt any existing locking protocol to allow only one thread per class to proceed to the filter step. As a result, this protocol is generic in the sense that it does not impose which sublocking protocol to use at each priority class. Using this protocol, we were able to evaluate various locking protocol combinations.

Figure 5 illustrates how this protocol is implemented. We exposed two API routines for the two classes of threads. The regular interfaces are for normal usage that we consider as high priority (lines 12 and 20). The low-priority interface (prefixed with `_l`) is meant for low-priority execution paths (lines 28 and 34). The data structure of this lock (defined at lines 3–8) uses two arbitrary locks, `high_p` and `low_p`, corresponding to the two classes of threads. The `go_straight` flag, set by the first high-priority thread (line 16), informs subsequent high-priority threads that the filter is set. The second step in the protocol is executed only by the first high-priority thread (line 15) or any low-priority thread (line 30). It consists of acquiring `filter`, a ticket lock that blocks low-priority threads when high-priority threads are present. A ticket locking protocol is sufficient here since at most two threads (one from each priority class) will be competing for it and using other protocols, such as MCS, would not provide any additional benefits. At this point, the method reuses existing lock *acquire* and *release* operations and simple flag manipulation. The only extension required for

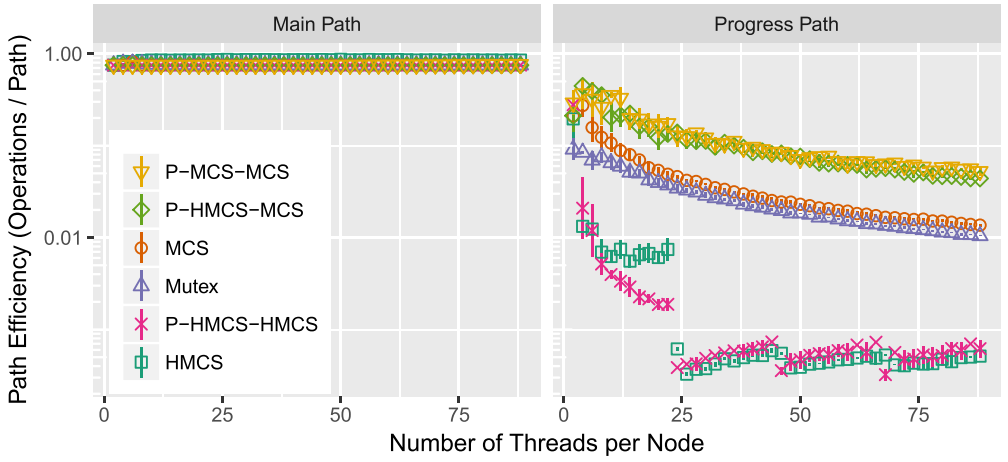
```
1
2   /* Lock Data-Structure*/
3   struct tlp_lock_t {
4       tlp_hlock_t high_p; // mutual exclusion between high-priority threads
5       bool go_straight;   // allow lockless second phase to high-priority threads
6       ticket_t filter;    // filter lock to block low-priority threads
7       tlp_llock_t lock_p; // mutual exclusion between low-priority threads
8   };
9
10
11  /* Regular/High-Priority Interfaces */
12  void acquire(tlp_lock_t *L) {
13      acquire_high_p(L);              // acquire the high-priority lock
14      if (!L->go_straight) {          // if I am the first high-priority thread
15          ticket_acquire(&L->filter); // acquire the filter lock
16          L->go_straight = 1;         // inform next priority threads to go straight
17      }
18  }
19
20  void release(tlp_lock_t *L) {
21      if (nowaiters_high_p(L)) {      // no high-priority thread is waiting
22          L->go_straight = 0;         // force high-priority threads to acquire filter
23          ticket_release(&L->filter); // allow low-priority threads to pass
24      }
25      release_high_p(L);
26  }
27
28  int acquire_l(tlp_lock_t *L) {
29      acquire_low_p(L);               // acquire the low-priority lock
30      ticket_acquire(&L->filter);     // unconditionally acquire the filter lock
31
32  }
33
34  void release_l(tlp_lock_t *L) {
35       ticket_release(&L->filter);    // unconditional release the filter lock
36       release_low_p(L);              // release the low priority lock
37  }
```

Fig. 5. Priority locking pseudo-algorithm with two levels of priority.

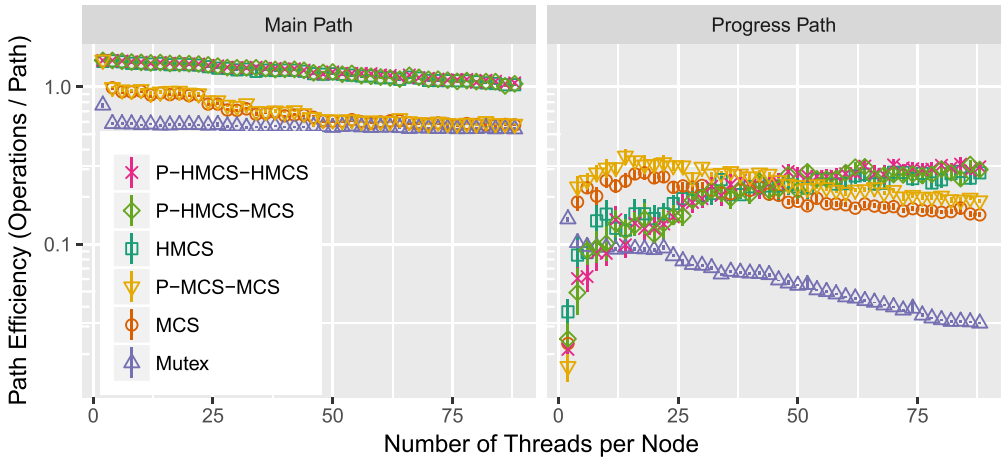
this method to work is to have an API routine that allows checking whether high-priority threads are waiting, namely, `nowaiters_high_p` (line 21). This check is necessary in order to reset the `go_straight` flag correctly. Failing to reset this flag might indefinitely block low-priority threads. This extra routine is allowed to return a true negative; that is, it would return *true* even if there were high-priority threads waiting. A false positive, however, is unacceptable because it will cause deadlock. This routine has been implemented for all locking protocols studied in this article.

4.2 Empirical Progress Analysis

Figures 6(a) and (b) compare the path efficiency of the traditional flat priority locks and our new two-level priority locking protocol variations with the latency and throughput benchmarks. In the following, we adopt the naming convention P-X-Y for the various TLP versions, where X is the protocol used for the high-priority class (i.e., main path) and Y is the protocol used for the low-priority class (i.e., progress path). The generic nature of TLP allowed us to readily evaluate various combinations for the high-priority and the low-priority subprotocols, but for brevity we show results only for the most insightful subset. In the latency-oriented regime, we observe that any lock that features HMCS for the progress path (low priority) performs the worst. On the other hand, locks that circulate ownership fairly at the level of the progress path ensure higher communication progress. In particular, the priority locking protocol improves upon the flat MCS lock, indicating that the prioritization is effective. In a throughput-oriented regime, we observe that the protocols that feature HMCS on the main path achieve the best efficiency and those that feature MCS for high priority perform the worst on the main path. The common protocol that



(a) Progress efficiency per execution path with a latency-oriented benchmark



(b) Progress efficiency per execution path with a message-rate-oriented benchmark

Fig. 6. Path efficiency comparison between flat and two-level priority locks. Each data point is the arithmetic mean of 10 runs with a confidence interval of 95% (error bars).

performs best in both regimes is P-HMCS-MCS. Thus, having high-throughput synchronization is important when issuing operations, but waking up threads on the progress path in a fair manner allows the best overall communication progress.

5 EVALUATION

In this section, we present performance results with the various locking protocols using microbenchmarks and a particle transport proxy application.

5.1 Testing Platforms

Our analysis and evaluation were conducted on a commodity multicore Intel Omni-Path cluster, as detailed in Table 1. The nodes are based on Intel dual-socket Broadwell processors interconnected

Table 1. Platform Specifications

Microarchitecture	Broadwell	L3 Size	28MB
Processor	Xeon 2699v4	L2 Size	256KB
Clock frequency	2.2GHz	Number of nodes	12
Number of sockets	2	Interconnect	Intel Omni-Path
Number of NUMA nodes	4*	Compilers	Intel 17.0.2
Cores per NUMA node	11	Linux kernel	3.10.0-693
HW threads per core	2	Glibc	2.17

*Two NUMA nodes per socket in this processor is a result of cluster-on-die mode, which is a BIOS option. The more common option is a single NUMA domain per socket.

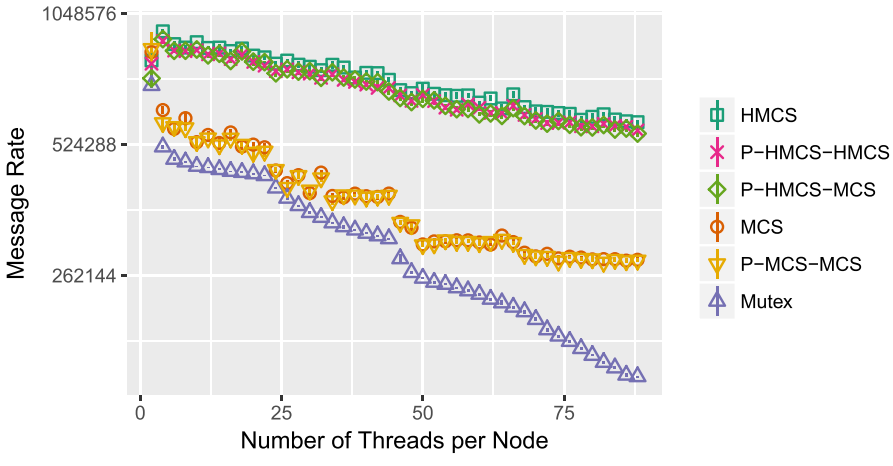
with an Intel Omni-Path fabric. The cluster is located at the Joint Laboratory for System Evaluation at Argonne National Laboratory. The nodes are 88-way hardware threaded and are characterized by a three-level memory hierarchy: board level (all threads), NUMA node level (threads sharing the same level 3 cache), and core level (threads sharing the L1 and L2 caches). HMCS has been built to match this architecture with three hierarchy levels, and the threshold for local passing has been tuned to the minimal value that achieves at least 95% of the peak throughput ($threshold = 256$ at all levels) when accessing 10 cache lines in the critical section. Intel Turbo Boost has been disabled to avoid dynamic frequency scaling inference with the experiments. All the locking protocols have been implemented as part of an open-source library called Izem, which is publicly available on a GitHub repository.[5] In particular, HMCS is portable across hardware platforms by leveraging the `hwloc` tool [8].

5.2 Microbenchmarks

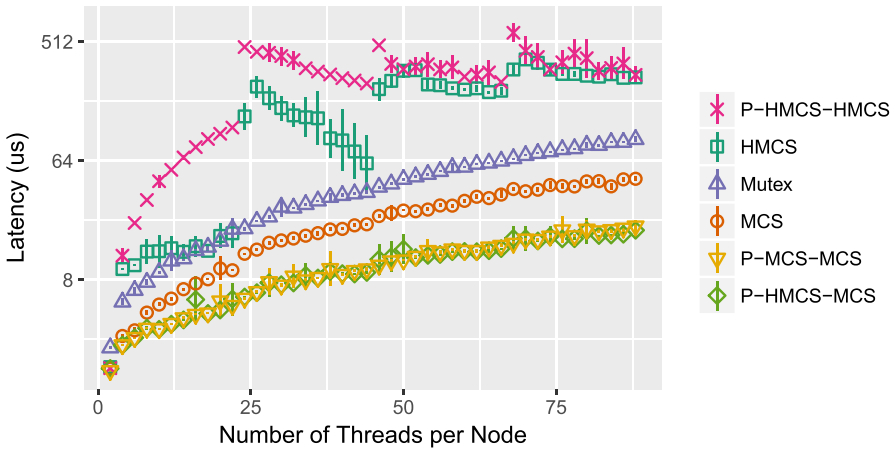
We begin by presenting results with multithreaded two-sided point-to-point throughput and latency benchmarks. Figures 7 and 8 summarize the results with various locking protocols while scaling the number of threads per MPI process or the message sizes, respectively. The benchmarks used here are the same as those described in Section 3. When the number of threads are scaled, we observe in Figure 7 that the performance results follow the same trend as the progress efficiency analysis in the preceding section. In a throughput-oriented regime, locks that feature HMCS as the locking protocol for the whole system or as the protocol for high-priority threads perform the best. In particular, HMCS-based protocols achieve roughly 2× speedup over P-MCS-MCS, which performs similarly to our previous record with the CLHP locking protocol [5]. In the latency-oriented regime, locks that feature MCS as the locking protocol for the progress path perform the best. When the message sizes are scaled, we observe in Figure 8 that the gap between the various protocols is more pronounced for message sizes below 8KB, after which only locks that feature HMCS on the progress path stand out negatively. The gap between MCS, P-MCS-MCS, and P-HMCS-MCS is not significant because the main time-consuming part is moving data across the network rather than managing contention. Combining all results, we conclude that P-HMCS-MCS is the protocol that offers the best scaling properties in both throughput- and latency-oriented regimes for medium and small messages since contention is more prominent.

We have also performed a similar evaluation with one-sided, or Remote Memory Access (RMA), interfaces. In this communication model, a process can access remote memory regions of other processes with little to no involvement from the target side. In the following, we summarize

[5] https://github.com/pmodels/izem.



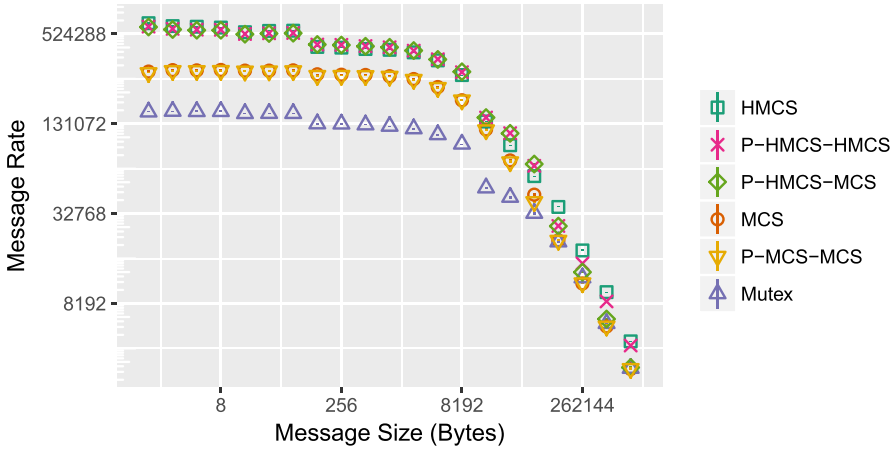
(a) Message Rate



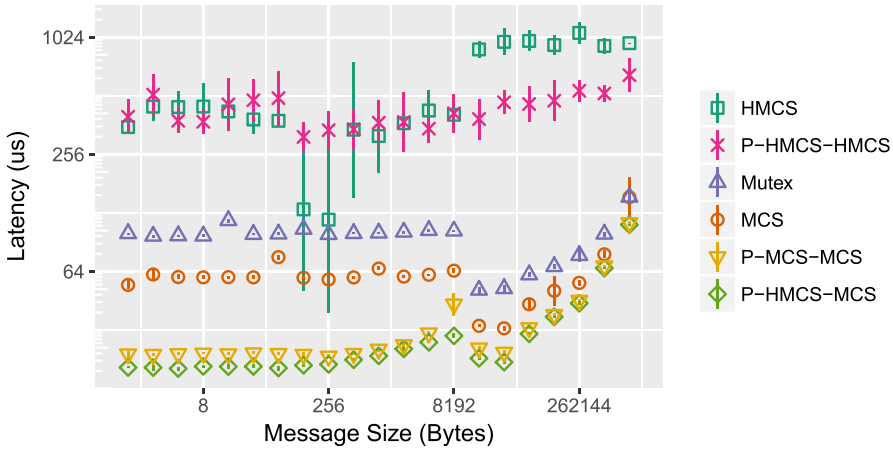
(b) Latency

Fig. 7. Strong scaling the number of threads per process in point-to-point message rate and latency benchmarks with respect to the locking protocol. Here, 64B messages have been used. Each data point is the arithmetic mean of 10 runs with a 95% confidence interval. Each run executes 4,000 and 1,000 iterations after a warm-up step for the message rate and the latency benchmarks, respectively.

our observations and leave out the details for brevity. In summary, when RMA operations (e.g., `MPI_Put` and `MPI_Get` operations) get issued concurrently by multiple threads but only a *single thread* waits for their completion with a synchronization call (typically using `MPI_Win_unlock` or `MPI_Win_fence`), results are similar to the throughput-oriented point-to-point regime discussed earlier. Since concurrency occurs only at issuing time, there is no interference from the progress path, and protocols that featured HMCS for the main path performed the best. What sets the RMA case significantly apart from the point-to-point case is when RMA synchronization is performed concurrently by multiple threads (e.g., with `MPI_Win_flush`). Unlike point-to-point completion calls, such as `MPI_Wait`, that wait for a specific set of operations to complete, RMA synchronization calls wait for the completion of all previously issued operations on the target window.



(a) Message Rate



(b) Latency

Fig. 8. Message rate and latency at full concurrency (88 threads per process) with respect to the locking protocol and message sizes. Each data point is the arithmetic mean of 10 runs with a 95% confidence interval. Each run executes 4,000 and 1,000 (200 for messages larger than 8KB) iterations after a warm-up step for the message rate and the latency benchmarks, respectively.

Most MPI implementations track pending RMA operations with a shared counter. As a result, if a thread T1 is in a synchronization call waiting for the counter to reach zero to return (i.e., all operations have completed), another thread T2 might increment that same counter (i.e., it issued new operations) thereby increasing the wait time of T1. This model effectively imposes unnecessary synchronization since T1 waits for the completion of operations that got issued *after* its synchronization call. In practice, this leads to a scalability collapse worse than with single-threaded synchronization calls or when using point-to-point communication. This issue, however, is orthogonal to lock contention management and requires an algorithmic fix suitable to multithreaded RMA synchronization.

Table 2. Input Parameters for the SNAP Runs

nthreads	88	lx	0.08	src_opt	3	iitm	5
npey	**variable**	lz	0.08	timedep	1	epsi	0.0001
npez	**variable**	ly	0.08	it_det	0	scatp	0
ndimen	3	nmom	4	tf	1.0	fixup	0
nx	128	nang	16	nsteps	10	angcpy	1
ny	72	ng	88	oitm	40	ichunk	4
nz	72	mat_opt	1	fluxp	0		

The parameters *npey* and *npez* vary according to the scale of the experiment.

5.3 SNAP: A Particle Transport Proxy Application

SNAP[6] is a proxy application, or miniapp, that models the computational characteristics of the PARTISN transport code. PARTISN is an MPI parallel application that solves the linear Boltzmann transport equation on multidimensional grids. SNAP does not solve actual physical problems; instead, it approximates the computational intensity, memory footprint, and communication patterns of PARTISN. For a time-dependent transport equation, SNAP exploits parallelism at the level of the spatial, angular, and energy dimensions. Computation along the energy domain is carried out typically through several energy groups, which are suitable for the thread-level parallelism of modern cluster nodes. Because of its prohibitive size, the spatial domain is partitioned across MPI ranks and traversed through sweeps along the discrete direction of the angular domain with the necessary data exchanges between ranks. The traversal follows the parallel Koch-Baker-Alcouffe wavefront method [6]. Most of the communication is performed by using point-to-point communication. Furthermore, threads are configured to perform both computation and MPI communication concurrently.

For our evaluation, we chose one of the regression tests that uses the method of manufactured solutions setting to generate a source for the SNAP computations in a three-dimensional spatial domain. This test was used as a base to generate scaling experiments. The input parameters used in this evaluation are listed in Table 2. We performed strong-scaling experiments by increasing the number of MPI processes along the Y and Z dimensions (npey and npez) while fixing the problem size (nx, ny, nz) = (128,72,72) to fit in the memory of a single node, and we varied the number of MPI processes by doubling the number of ranks per one of the Y or Z dimensions while alternating. Each MPI process runs on its dedicated node and spawns 88 threads to fully occupy the computational units.[7]

The scaling results with respect to the locking protocol in effect on the Broadwell cluster are shown in Figure 9. Scaling MPI processes goes from left to right in the figure and reaches 12 processes running 1,056 threads in total. This effectively shifts the application from a computation-intensive regime to a communication-intensive one. We observe that at the smallest scale (two MPI processes) all locking protocols perform similarly. This result indicates that lock management is not an issue at this stage. When the number of MPI processes is scaled, the time decreases, but the scalability varies across locking protocols and exhibits the highest variation when exceeding three MPI processes in either the Y or the Z dimension. In these cases, the protocols that negatively stand out are Pthread mutex and those that use HMCS for the progress path. SNAP performs best with MCS, followed by protocols that feature MCS for low priority. P-HMCS-MCS performs closely to

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

[7] Other variations in the input parameters and scaling methods showed either no contention for MPI communication or scalability bottlenecks that are not strongly related to locking issues.

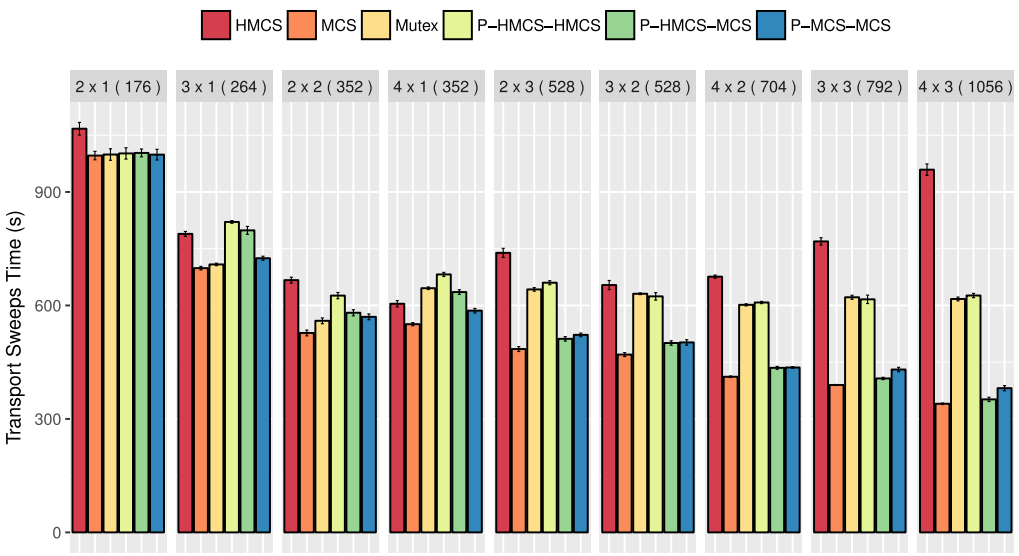


Fig. 9. SNAP strong-scaling results on the Broadwell cluster with 88 threads per MPI process. We show the time only for the *transport sweeps* step, which is the most time-consuming. The results are organized according to the number of processes on the *Y* and *Z* dimensions. The notation "y × z (t)" indicates y processes on the *Y* dimension (npey), z processes on the *Z* dimension (npez), and an aggregated total number of t threads. The results from left to right represent strong scaling of MPI processes on each dimension. Each bar is the arithmetic mean of five runs with a 95% confidence interval (error bars).

MCS and outperforms P-MCS-MCS at the highest-contention regimes. These results indicate that P-HMCS-MCS has negligible overhead over MCS and improves upon P-MCS-MCS since SNAP is slightly sensitive to the rate at which nonblocking operations are pipelined. At the largest scale, threads communicate messages whose sizes are larger than 8KB. Each thread pipelines only two operations at a time before waiting for their completion. This behavior is closer to the latency-oriented benchmark since there is little communication hiding, but it is not completely the same. Looking back at message rate and latency results for messages bigger than 8KB (Figure 8), we observe that the discrepancies between the locking protocols that feature MCS on the progress path are not significant. We conclude that the main reason for P-HMCS-MCS not outperforming MCS is that messages are big enough to make the influence of contention management differences between them less significant. Having input problems that generate even more fine-grained messages showed P-HMCS-MCS outperforming MCS, but the overall application does not scale well. Since this type of action is something a user would avoid doing, we did not include such results.

6 DISCUSSION: LIMITS OF CONTENTION MANAGEMENT THROUGH LOCKING

From the previous experiment we have seen that managing contention through the locking protocol by itself results in scalability drops despite using state-of-the-art protocols. We believe that this situation will be exacerbated when scaling the number of computational units and going deeper in the memory hierarchies and will require more elaborate methods that diverge from the pure locking thread-safety model. This section discusses a few directions along this path.

6.1 Progress Wakeup Complexity

Our performance and progress efficiency analysis shows that a strict FIFO locking protocol, such as that of MCS, allows the best overall productivity on the progress path. Unfortunately, such

progress management remains inefficient. Comparing the scalability loss (from two-way to 88-way concurrency per node) between the throughput and latency benchmarks (Figure 7(a) vs. (b)) with the best-performing protocol (P-HMCS-MCS), we observe that the scalability loss in the latency regime is much more significant: more than 8× scalability loss as opposed to less than 2× compared with the throughput case. The major source of this scalability loss is from the blind lookup for a thread to produce work on the progress path. Since each thread on the client side waits for the acknowledgment message from the server and only one of them can consume it at a time, the time to find the target thread is $O(N)$ proportional to the number of threads in the progress path—clearly not a scalable approach. We believe that this situation cannot be overcome with the current locking protocol semantics and will require either combining with other synchronization constructs, such as condition variables, or extending the flat acquire/release locking protocol to a more advanced model that incorporates information to achieve an $O(1)$ wakeup while ensuring traditional mutual exclusion.

We have explored the former method by combining locks with a condition variable-like construct (called *synchronization counter*) that counts the number of outstanding events a thread is waiting for in a critical section before waking up [13]. Results showed a significant reduction in scalability loss on latency-oriented regimes. Unfortunately, this approach adds significant software complexity to the MPI library compared with the simpler pure-locking methods. We are currently exploring the latter approach of extending traditional locking protocols with minimal new semantics that retain the simplicity of locking while achieving $O(1)$ wakeup on known events by the user for greater scalability.

6.2 Leveraging Abort Locking Protocols

Our thread-safety model description in Section 2.2 assumed that *all* API calls start executing the main path at the beginning and fall back to the progress path on waiting for completion. This assumption is sound in general, but exceptions exist. For instance, nonblocking progress calls, such as the `MPI_Test` family of calls, do not have a progress loop internally but are meant only for progress. As a result, a model that assumes these calls take the progress path right at the beginning are valid as well. In practice, which model would perform better is application dependent. We arbitrarily chose to give these calls high priority and leave it to the user to tune the frequency of polling for progress. Putting aside this semantic issue, these nonblocking progress routines offer optimization opportunities. The `MPI_Test` family of calls allow advanced control over overlapping communication with other operations, including computation and other communication operations. Instead of blocking waiting for completion, other work could be executed. Unfortunately, the blocking nature of a lock *acquire* operation translates an MPI nonblocking call into a blocking one and reduces opportunities for the aforementioned advantages.

To alleviate this issue, we explored the possibility of leveraging abort locking [10], which differs from the traditional locking protocol assumed in Section 2.2. In this model, nonblocking progress routines attempt a lock acquisition and return to the application on failure to acquire the lock after a given timeout period. Instead of waiting for the lock acquisition, useful work could be executed. Our results show that this model achieves significant scalability improvement with a multithreaded MPI implementation of the breadth-first-search (BFS) algorithm.

6.3 Combining Advanced Contention Management with Contention Avoidance

In this article, we assumed a *worst-case* scenario in that contention is assumed to be taking place. While this is an important aspect to the problem of thread safety in MPI, the orthogonal dimension of avoiding contention is at least as important. We believe that combining the two directions will have a synergistic effect on reducing the runtime contention. However, a cost-effectiveness

study of both methods on the same testbed is still warranted. Such a study can guide the development process of a thread-safe runtime. A possible model would be to start with a coarse-grained thread-safety model, explore effective contention management methods, reduce granularity if high contention persists, and repeat the process.

7 RELATED WORK

Tackling challenges related to the interoperability between MPI and threads is not a new topic. Researchers have explored the topic from different perspectives, ranging from algorithmic optimizations for specific aspects of the runtime to synchronization granularity optimizations and standard extensions.

Algorithmic optimizations target specific components of an MPI implementation. For instance, Gropp and Thakur provided an efficient algorithm for generating context ids in a multithreaded context [22]. Goodell et al. showed that concurrent accesses from multiple threads to MPI objects can be a bottleneck when using reference counts, and they proposed more scalable solutions [21].

Granularity optimizations, on the other hand, have been proposed to reduce the extent of thread synchronization. Gropp and Thakur presented an exhaustive thread-safety requirement analysis of MPI functions and their implementation issues [22]. Their study showed that not every routine requires locking. Most production MPI implementations ensure thread safety of the core functionalities through a coarse-grained critical section. The exception is MPICH, which exploits fine-grained locking on IBM Blue Gene systems. We investigated this fine-grained design in our prior work, where we compared different levels of critical-section granularity and their implications in terms of performance and implementation complexity [7, 17].

Over time, the MPI standard has been extended to support more efficient multithreaded MPI implementations. Hoefler et al. showed that `MPI_Probe` cannot be used by multiple application threads at the same time and that a conventional lock-based workaround is not scalable [23]; they proposed an efficient solution that goes beyond the implementation level and requires changing the MPI standard. In order to ensure contention-free multithreaded communication, close to that of multiprocess-driven communication, the concept of MPI endpoints has emerged as a potential extension that is being reviewed for inclusion in the next version of the MPI standard [16].

Most existing works have aimed at avoiding contention. To the best of our knowledge, we were the first to tackle this challenge from a contention management perspective in our prior publication [4]. The current article extends our prior work by providing substantially more analysis and bringing more insight into the interoperation between locking and MPI runtime progress.

8 CONCLUSION AND FUTURE WORK

In this article, we investigated the practicality of efficiently managing contention for a communication path in MPI from a purely locking protocol perspective. We found that locking protocol characteristics greatly influence progress on the communication path. From the shortcomings of existing protocols, we devised alternatives that offer better communication progress properties that leverage knowledge of the memory hierarchy for high throughput and MPI progress characteristics to reduce unproductive lock acquisitions. Nevertheless, scalability drops have been observed and showed limits of the pure-locking model to fully manage access to a communication path on contention. We therefore briefly discussed more elaborate contention management and contention avoidance methods that have the potential to further mitigate the scalability challenges. The primary recommendation for thread-safety management in MPI as well as other communication libraries is to reduce or eliminate interference between threads issuing operations and those waiting for completion. When contention is unavoidable, it is advised to move threads waiting for completion out of the critical path of threads issuing operations either through priority locking as

done in this article or by moving the former to a waiting queue, such as a condition variable, to avoid slowing issuing operations. Furthermore, productive nonblocking operations must be issued on a fast path characterized by high-throughput and low latency.

ACKNOWLEDGMENTS

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] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing* 1, 2 (2015), Article 13.

[2] Abdelhalim Amer, Pavan Balaji, Wesley Bland, William Gropp, Rob Latham, Huiwei Lu, Lena Oden, Antonio Pena, Ken Raffenetti, Sangmin Seo, et al. 2015. MPICH User's Guide.

[3] Abdelhalim Amer, Huiwei Lu, Pavan Balaji, and Satoshi Matsuoka. 2015. Characterizing MPI and hybrid MPI+threads applications at scale: Case study with BFS. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*. 1075–1083.

[4] 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.

[5] A. Amer, Huiwei Lu, Yanjie Wei, Jeff Hammond, Satoshi Matsuoka, and Pavan Balaji. 2016. *Locking Aspects in Multithreaded MPI Implementations*. Technical Report P6005-0516. Argonne National Lab.

[6] 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.

[7] 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.

[8] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'10)*. IEEE, 180–186.

[9] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*, Vol. 48. 157–166.

[10] Milind Chabbi, Abdelhalim Amer, Shasha Wen, and Xu Liu. 2017. An efficient abortable-locking protocol for multi-level NUMA systems. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 61–74.

[11] 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.

[12] 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.

[13] Hoang-Vu Dang, Sangmin Seo, Abdelhalim Amer, and Pavan Balaji. 2017. Advanced thread synchronization for multithreaded MPI implementations. In *Proceedings of the17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'17)*. IEEE, 314–324.

[14] Dave Dice. 2017. Malthusian locks. In *Proceedings of the 12th European Conference on Computer Systems*. ACM, 314–327.

[15] David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing* 1, 2 (2015), 13.

[16] James Dinan, Pavan Balaji, Dave Goodell, Doug Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI interoperability through flexible communication endpoints. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'13)*, 13–18.

[17] 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, 11–20.

[18] Ulrich Drepper. 2009. Futexes are tricky. Retrieved October 18, 2016 from https://www.akkadia.org/drepper/futex.pdf. *Red Hat Inc.* (2009).

[19] Ulrich Drepper and Ingo Molnar. 2005. The native POSIX thread library for Linux. Retrieved October 18, 2016 from https://www.akkadia.org/drepper/nptl-design.pdf. *White Paper, Red Hat Inc.* (2005).

[20] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. [n.d.]. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*.

[21] David Goodell, Pavan Balaji, Darius Buntinas, Gabor Dozsa, William Gropp, Sameer Kumar, Bronis R. de Supinski, and Rajeev Thakur. 2010. Minimizing MPI resource contention in multithreaded multicore environments. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'10)*, 1–8.

[22] William Gropp and Rajeev Thakur. 2007. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing* 33 (2007), 595–604.

[23] Torsten Hoefler, Greg Bronevetsky, Brian Barrett, Bronis R. de Supinski, and Andrew Lumsdaine. 2010. Efficient MPI support for advanced hybrid programming models. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI'10)*, Vol. 6305. Springer, 50–61.

[24] 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.

[25] Saurabh Kalikar and Rupesh Nasre. 2016. DomLock: A new multi-granularity locking technique for hierarchies. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. 23.

[26] 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.

[27] Message Passing Interface Forum. 2015. *MPI: A Message-Passing Interface Standard Version 3.1*. Technical Report.

Received May 2016; revised May 2018; accepted July 2018