MPI+Threads: Runtime Contention and Remedies

Abdelhalim Amer
Tokyo Institute of Technology, Japan
amer@matsulab.is.titech.ac.jp

Huiwei Lu
Argonne National Laboratory, USA
huiweilu@anl.gov

Yanjie Wei
Shenzhen Institute of Advanced Technologies,
Chinese Academy of Sciences, China
yj.wei@siat.ac.cn

Pavan Balaji
Argonne National Laboratory, USA
balaji@anl.gov

Satoshi Matsuoka
Tokyo Institute of Technology, Japan
matsu@is.titech.ac.jp

Abstract

Hybrid MPI+Threads programming has emerged as an alternative model to the "MPI everywhere" model to better handle the increasing core density in cluster nodes. While the MPI standard allows multithreaded concurrent communication, such flexibility comes with the cost of maintaining thread safety within the MPI implementation, typically implemented using critical sections. In contrast to previous works that studied the importance of critical-section granularity in MPI implementations, in this paper we investigate the implication of critical-section arbitration on communication performance. We first analyze the MPI runtime when multithreaded concurrent communication takes place on hierarchical memory systems. Our results indicate that the mutex-based approach that most MPI implementations use today can incur performance penalties due to unfair arbitration. We then present methods to mitigate these penalties with a first-come, first-served arbitration and a priority locking scheme that favors threads doing useful work. Through evaluations using several benchmarks and applications, we demonstrate up to 5-fold improvement in performance.

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

Keywords MPI, threads, runtime contention, critical section

1. Introduction

Most parallel applications running on high-performance computing (HPC) systems use the Message Passing Interface (MPI) [5] for interprocess communication. Given the increasing difference in growth of the number of cores and the remaining resources on the node (e.g., memory, cache, network endpoints), MPI alone might not always be the most efficient model for exploiting intranode resources. To alleviate such issues, application developers are looking for models that allow node resources to be shared. Recently, an increasing number of applications have been adopting an MPI+X hybrid model, where X often designates a threading model [7]. The goal is to use MPI for interprocess communication while a threading model such as OpenMP [2] or Intel Threading Building Blocks [25] handles shared-memory parallelism within the same address space. This model, however, comes with the synchronization and memory consistency costs related to data sharing between threads.

The MPI standard defines how such hybrid applications interoperate with an MPI library. In particular, MPI implementations that offer multithreading support must guarantee thread safety. Such thread-safety is guaranteed through a combination of processor atomic operations and critical sections in virtually every MPI implementation today. In order to maintain performance, however, two orthogonal dimensions of optimizations need to be investigated—(1) granularity of critical sections and (2) arbitration of critical sections. Most MPI implementations use coarse-grained critical sections due to its relative simplicity, though some implementations have investigated fine-grained critical sections as well [6, 12, 16]. However, all existing implementations ignore the orthogonal dimension of critical section arbitration.

In this paper, we investigate how critical section arbitration can affect communication performance. Specifically, we perform an in-depth analysis of the MPICH [1] runtime, which forms the basis of most MPI implementations in the world today. Our analysis shows that POSIX thread mutexes, that are used by most MPI implementations, is "biased" based on the memory hierarchy on the node. We demonstrate that the deep memory hierarchies on modern architectures promote threads closer to the previous owner of the critical section as the new owner. This results in lock monopolization causing threads having useful work to not get access to the critical section for long periods of time, thus slowing overall progress.

We propose two approaches to mitigate the issue. The first approach is a simple first-come-first-served (FCFS) model that removes any hardware-induced bias in the arbitration model. This allows all threads to make progress thus avoiding lock monopolization and starvation issues. The second approach is a custom critical section arbitration model that reintroduces bias into the arbitration process, but this time taking into account the execution paths inside the MPI runtime to favor threads with a higher probability of doing useful work. We evaluate our methods against the baseline mutex approach and show up to 5-fold improvement in performance with various microbenchmarks, computation kernels, and a genome assembly application.

The remainder of this paper is organized as follows. In Section 2 we discuss thread-safety challenges in MPI in general, and thread-safety measures in MPICH in particular. We briefly describe the testbed used for our experiments in Section 3. In Section 4 we present an in-depth analysis of the critical-section arbitration process and the design, implementation, and preliminary evaluation of our solutions in Section 5. In Section 6 we evaluate and compare all methods with microbenchmarks, 3D stencil and graph traversal kernels, and a genome assembly application. Additional discussion on the impact of the proposed work is presented in Section 7. In Section 8 we discuss related work, and present concluding remarks and ideas for future work in Section 9.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3205-7/15/02...$15.00
http://dx.doi.org/10.1145/2688500.2688522

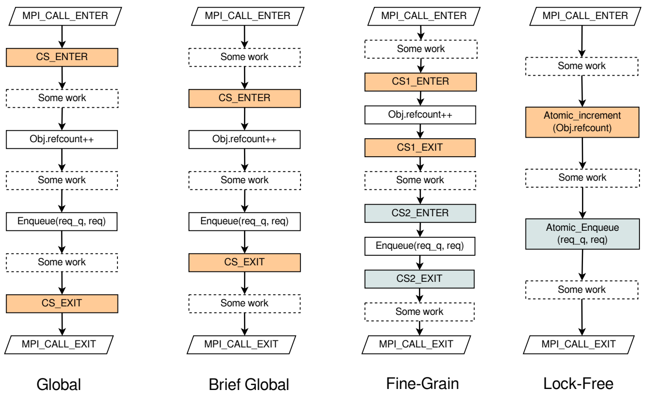


Figure 1. Critical-section granularity. `CS_ENTER` and `CS_EXIT` denote the protocol operations executed at the entry and exit of a critical section, respectively. `Atomic_OP` refers to a lock-free implementation of the operation `OP`.

2. Thread Safety in MPI

In this section we provide background information about thread safety and its relation to MPI, and we discuss thready-safety efforts in MPICH.

2.1 MPI Requirements for Thread Safety

The MPI standard defines four levels of thread safety: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. These levels are listed in increasing order of thread safety. Given the performance costs of thread safety, these levels allow adjusting the MPI implementation to the needs of the target application without incurring unnecessary overheads. In this work, we target the least restrictive level, `MPI_THREAD_MULTIPLE`, which allows multiple threads to call MPI routines concurrently.

Many aspects are considered when designing a thread-safe MPI library. Critical-section granularity is an important parameter that influences the degree of parallelism allowed for concurrent thread executions. That is, the longer a critical section is, the more it incurs serialization and thus hinders parallel performance. In prior work we investigated different levels of critical-section granularity (Figure 1) and their implications in terms of performance and implementation complexity [6, 12]. In Figure 1, the granularity becomes increasingly fine-grained from left (“Global”) to right (“Lock-free”). In practice, most MPI implementations use a combination of “Global” and “Lock-free”, with the latter applying to atomic updates of reference counters that are required for managing internal MPI objects.

Regardless of the granularity, however, contention to enter a critical section can still occur, and serialization is inevitable. Several mechanisms, such as mutexes and spinlocks, have been developed to implement mutual exclusion in order to protect a shared resource. Such mechanisms usually differ in the way they handle contention, the unnecessary lock/unlock overhead they incur in scenarios where there is no contention, reducing synchronization granularities, and time to transfer ownership of the lock. However, the serialization order of the critical section and its effect on an MPI runtime performance are less understood. To the best of our knowledge, no existing work has studied thread synchronization from this perspective.

2.2 Thread Safety in MPICH

In this section, we analyze thread-safety in MPI implementations in the context of MPICH, though the analysis is largely true for most other MPI implementations as well. On our platform, MPICH thread safety is ensured by using a Native POSIX Threading Library (NPTL) [22] global pthread mutex. Locking a default mutex in NPTL is implemented as follows. First, a thread tries to acquire the lock in the user space with an atomic compare-and-swap operation. If not successful, the thread goes to sleep in the kernel space using the `FUTEX_WAIT` operation of the futex (fast user-space mutex) system call [13, 14]. The mutex holder wakes up at most one thread in the futex queue, with the `FUTEX_WAKE` operation, when leaving the critical section. The thread that wakes up again competes to acquire the lock and the same process repeats if the lock acquisition fails.

From an arbitration perspective two distinct arbitration policies can be identified. In the kernel space the futex wake-up operations are performed according to the kernel-scheduling policy. In the user space, however, the policy obeys the fastest-thread-first rule. We note two major implications. First, the overall arbitration is dominated by the user-space policy since threads must try to acquire the lock every time they leave the kernel space. Second, the current user space arbitration model can yield to starvation since no mechanism to maintain fairness is used. In Section 4, we analyze how such starvation affects the MPICH runtime.

3. Testing Platform

All experiments reported in this paper were conducted on a commodity cluster with each node equipped with a dual-socket Intel Nehalem processor where each socket contains 4 cores (simultaneous multithreading (SMT) is disabled). In addition, the cluster nodes are interconnected with the Mellanox InfiniBand QDR fabric. The detailed specification is summarized in Table 1. We used MPICH 3.2a1, which is configured to run with the Mellanox Messaging Accelerator (MXM) interface.

Table 1. Target machine specification

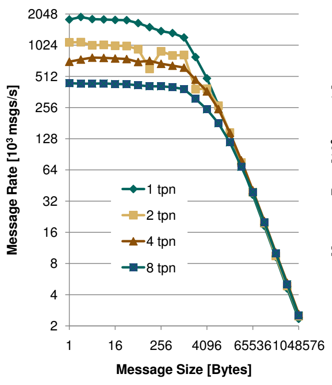
Architecture	Nehalem
Processor	Xeon E5540
Clock frequency	2.6 GHz
Number of sockets	2
Cores per socket	4
L3 Size	8192 KB
L2 Size	256 KB
Number of nodes	310
Interconnect	Mellanox QDR

4. MPI Runtime Analysis

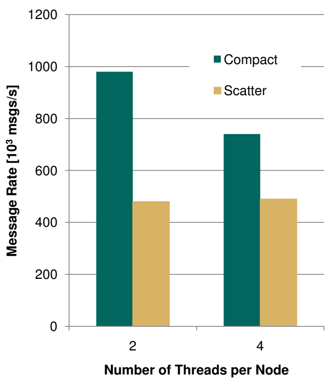
In this section we first consider MPICH as a black box and perform a simple performance evaluation and analysis in a multithreaded scenario. We then present a detailed profiling in order to shed light on the runtime inefficiencies. Unless specified otherwise, we bind the first four threads to cores on the first socket and the rest to cores on the second.

4.1 Initial Performance Evaluation

Here we present a simple evaluation that reflects the runtime contention experienced with MPICH. We measure the point-to-point throughput using a benchmark derived from `osu_bw` of the OSU microbenchmarks suite [3]. We modified the benchmark to fork a team of threads to perform the communication in parallel. Since threads are not doing computation and MPICH uses a global critical section, we expect to see serializaction in communication which would result in no speedup. Figure 2a shows the message rate as a function of message sizes and the number of threads per node. We observe performance degradation proportional to the number of threads reaching up to a four-fold reduction in



(a) Multithreaded throughput performance with varying message sizes; *tpn* stands for thread(s) per node



(b) Effect of thread concurrency and NUMA on probability performance

Figure 2. Preliminary evaluation of multithreaded communication performance

throughput. This experiment shows that there is contention in the MPI runtime although it does not expose the primary source of that contention. For large messages, network communication time dominates rendering runtime inefficiencies negligible.

4.2 Impact of NUMA on Critical Sections

Here we present the results of our experiments on throughput performance when threads are bound to cores on the same socket (`Compact`) vs. when they are bound to cores on different sockets (`Scatter`). The goal was to determine the effect of nonuniform memory access (NUMA) on the runtime contention. The results are shown in Figure 2b. We observe that throughput is 1.5 to 2-fold worse with a scatter binding. This suggests that the runtime contention is sensitive to NUMA. Given that mutex uses a compare-and-swap operation in user space, we point out two major drawbacks that are amplified by NUMA. First, the hand-off time[1] between threads is amplified by the intersocket latency. Second, since mutex does not guarantee fairness, ownership of the lock may not be passed fairly between threads. We speculate that NUMA biases the arbitration in favor of the threads on the same socket—more analysis on this speculation is presented in the Section 4.3.

4.3 Arbitration Fairness Analysis

The speculation of critical section arbitration being biased by NUMA architectures, as mentioned in Section 4.2, stems from the nonuniform proximity of threads to the memory hierarchy (caches and memory). Specifically, the thread that releases the lock dirties the cache line holding the lock, which makes it most favorable for other threads closest to this cache to acquire the lock.

To assess the degree of unfair arbitration, we manually instrumented MPICH to trace the lock acquisition, processed the collected data, and compared mutex to a fair arbitration to analyse to which extent mutex biases the arbitration. We further considered the role of the memory hierarchy by analysing two levels at which the arbitration may be biased. At the *core level*, given T threads waiting to enter the critical section, we estimate the probability P_c that the same thread reacquires the lock successively. At the *socket level*, given N sockets, T_i the number of threads on socket i waiting to enter the critical section, we estimate the probability P_s that the new owner of the lock runs on the same socket j as the previous owner.

[1] The elapsed time between when a lock holder marks the lock as free and when the next owner detects it

We statistically estimate these quantities for both mutex and a fair arbitration as follows:

$$P_c = (\sum_{l=1}^{L} X_l)/L$$
$$P_s = (\sum_{l=1}^{L} Y_l)/L$$

Where, for a mutex arbitration:

$$X_l = \begin{cases} 1 & \text{if same owner as } l-1 \\ 0 & \text{else} \end{cases}$$
$$Y_l = \begin{cases} 1 & \text{if same socket as } l-1 \\ 0 & \text{else} \end{cases}$$

and for a fair arbitration:

$$X_l = 1/T_l$$
$$Y_l = T_{j,l}/(\sum_{i=0}^{N-1} T_{i,l})$$

We discretized the execution at the lock acquisition level using the subscript "l" in our equations. L denotes the total number of lock acquisitions and X_l and Y_l are the probability of electing a thread running on respectively the same core and the same socket during lock acquisition l. T_l and $T_{i,l}$ represent respectively the total number of waiting threads and the number of waiting threads on socket i during lock acquisition l. We then analyzed the arbitration with the point-to-point throughput benchmark on our dual-socket system for each message size. We compute P_c and P_s using the previous equations and derive the ratio of the mutex probabilities over the fair arbitration probabilities that we refer as *Bias Factors*. Theses factors indicate to which degree mutex biases the arbitration where a fair arbitration would have a bias factor of 1. The results of our analysis are shown in Figure 3a. We notice that mutex biases the arbitration by 2x at the core level and 1.25x at the socket level on average across message sizes. These results confirm our assumption regarding the unfair arbitration and our belief that NUMA amplifies the problem.

In the next section we relate the unfair arbitration to the MPI implementation internals and thus explain the performance consequences.

4.4 Message Requests Analysis

Before we discuss our profiling method, we first describe how requests are handled inside the runtime. For simplification, we consider only nonblocking receive requests. The upper part of Figure 3b shows the state diagram of a receive request. When a user calls `MPI_Irecv`, a request is issued internally. If the corresponding message was already received in the *unexpected queue,*[2] then the request is marked as completed. Otherwise, the request is posted in the *posted queue*. Later, if a message arrives and matches the request, it will be marked as completed. `MPI_Wait` or `MPI_Test` or their derivatives will be called and block the caller until the request is found completed and then gets freed.

The point-to-point throughput benchmark performs two-sided nonblocking communication (`MPI_Isend` and `MPI_Irecv`) on a window of 64 requests, and `MPI_Waitall` is used to wait for all the requests in batches (bottom of Figure 3b). In our multithreaded version each thread manages its own window. In order to make progress on communication, all requests need to be detected as completed and then freed before moving to a new window. In a multithreaded scenario, threads can mark other threads' requests as completed inside the runtime, but only the thread that waits for the completion of a request can free it. In addition, we do not tag

[2] A queue for incoming message handlers without matching receive requests

messages so that threads can match any message from the same process and communicator.

Our profiling method relies on the notion of *dangling requests*, that is, requests that are completed but not yet freed. We track the number of these requests at sampling intervals corresponding to successive lock acquisitions. We then define our metric as the average number of dangling requests during the program execution. To make rapid progress on communication, threads should detect completed requests early, free them, and generate new requests to feed the runtime and the network. Thus, this metric should be kept low. In fact, with fair arbitration one would expect this metric to be low since requests will be issued in batches, communicated through the network, and detected by the threads evenly. The profiling results with the throughput benchmark are plotted in Figure 3c. We notice that the number of dangling requests is high. The reason is that the window of a starving thread will likely take longer to be filled with completed requests and that the requests inside incomplete windows are counted as dangling. A chain reaction will result by delaying the next requests to be issued and thus their network communication and the matching process. Consequently, the overall communication progress will be slowed. In the next section we propose alternatives to the original mutex-centered design with the goal of improving the progress of threads inside the MPICH runtime.

5. Reducing Contention

The analysis in the previous section confirmed that the threads have unfair access to the critical section. Here we explore two alternative locking methods that take into account fairness and communication progress in the arbitration policy.

5.1 First-In First-Out Arbitration

One alternative ensures fairness through a first-in first-out critical section arbitration. In Section 2 we pointed out that changing the scheduling policy in the kernel space does not affect the arbitration in the user space. To alter the lock arbitration, we rely on a locking method that is entirely implemented in the user-space through busy waiting. Several locking methods, for example, MCS [19] and ticket [18], establish a first-in first-out threads ordering in the user space. Ahe recent study by David et al. showed that the *ticket lock* performs well on most modern many-core architectures and in various contention scenarios [9]. The principle behind the ticket lock is simple. Each thread gets a ticket at the entry of a critical section and waits for its turn. Figure 4 shows a basic implementation. We note that only one atomic operation (`fetch_and_increment`) is needed. In addition, the busy-wait condition does not involve extensive cache traffic, unlike a `compare_and_swap`–based spinlock. Thus, our first step is to use a ticket lock to implement the MPICH global critical section.

After integration in the MPICH runtime, we compare the new design to the mutex-based runtime by analyzing dangling requests. The results are shown in Figure 5a. We observe that using ticket keeps the number of dangling requests very low and, according to our analysis, should have a positive effect on performance. We investigated the effect of the degree of concurrency and the NUMA effect by scaling the number of threads per core. The results are shown for compact and scatter bindings in Figure 5b. In a compact binding, the ticket method reduces contention compared with mutex and improves the performance by 68% with four threads. With scatter binding and two threads per node, the ticket method loses slightly to mutex, suggesting that the ticket method incurs more intersocket synchronization. Indeed, the mutex socket-level lock monopolization reduces the amount of intersocket synchronization compared with that of the ticket method. However, the benefit of fair arbitration grows with the degree of concurrency. We conclude that for a scatter binding and a low degree of concurrency, the overhead

```
int next_ticket;
int now_serving;

ticket_acquire_lock()
{
  my_ticket = fetch_and_incr(next_ticket);
  /* Wait for my turn */
  while(my_ticket!=now_serving) ;
}

ticket_release_lock()
{
  now_serving++;
}
```

Figure 4. Pseudo-algorithm for a ticket lock

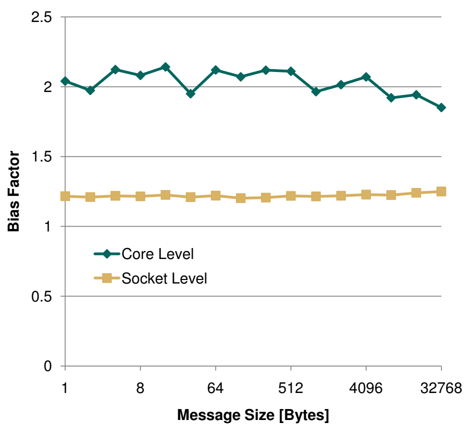
of intersocket thread synchronization may outweigh the benefit of fair arbitration. A common method for solving this issue is to spawn one process per socket and to use threads within a socket to avoid intersocket synchronization and data movements. In such cases, the ticket method will be more effective than mutex. Figure 5c shows throughput comparison between the two methods with respect to the message size. We notice that the ticket method outperforms mutex by 30% on average for messages below 4 KB. The gap between the methods decreases until reaching 32 KB, from which point performance differences become negligible.

5.2 Priority Locking Scheme

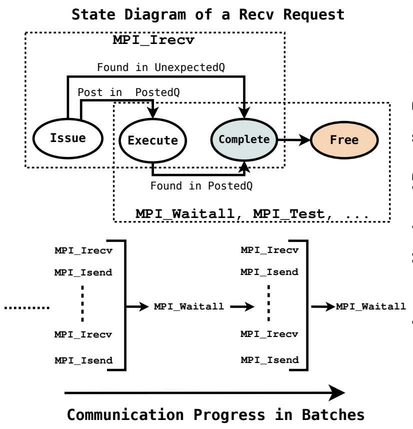
In this section, we describe our custom lock implementation, which takes into account MPICH internal details to prioritize threads with higher chances of making progress. We follow with an example benchmark where this method is superior to a flat fair arbitration.

The main consequence of unfair arbitration is the possibility that threads monopolizing the lock may not yield useful work while penalizing a starving thread that can make progress. It is not trivial to know a priori which thread is going make progress after getting the lock, since this situation depends in many cases on external events such as message reception. Nevertheless, we can identify unbalanced execution paths within the runtime that yield different amounts of work within the critical section. Our examination of the MPICH runtime exposed two coarse-grained execution paths as shown in the simplified flowchart of Figure 6a: (1) a *main path* that each thread-safe routine implements differently and (2) a *progress loop* that some MPI routines enter to poll for communication progress. For instance, `MPI_Irecv` may allocate resources and enqueue the request in a queue in the *main path*, while not going through the *progress loop*. On the other hand, a blocking call such as a `MPI_Wait` will enter the communication progress engine until the corresponding request completes. An important observation is that threads in the *main path* have more chances to yield useful work and thus are unlikely to waste a lock acquisition. There is an exception, however, for some MPI operations that may not make progress even in the main path such as `MPI_Test`. The ticket lock gives threads equal chances to acquire the lock independently from the path they take, for example, in cases where threads are blocked at the entry of the *main path* while waiting for threads in the *progress loop* to release the lock. Such cases may also yield to wasted lock acquisitions.

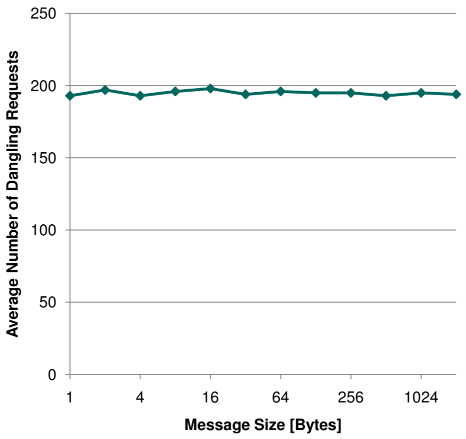
Given these observations, we propose a custom locking scheme that favors threads in the *main path*. This can be achieved by assigning a high priority to threads at the entry of MPI routines, then lowering their priorities if they enter the *progress loop*. To avoid lock monopolization, we ensure fair arbitration between threads of the same priority. To achieve this goal without using a heavyweight locking mechanism, we implement the lock acquisition and relinquish



(a) Mutex arbitration fairness analysis at the core and socket levels

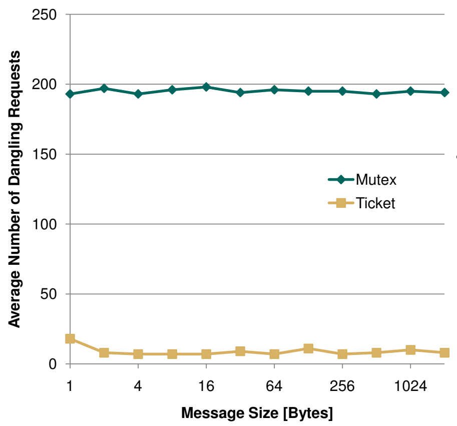


(b) Communication progress with MPI non-blocking two-sided communication

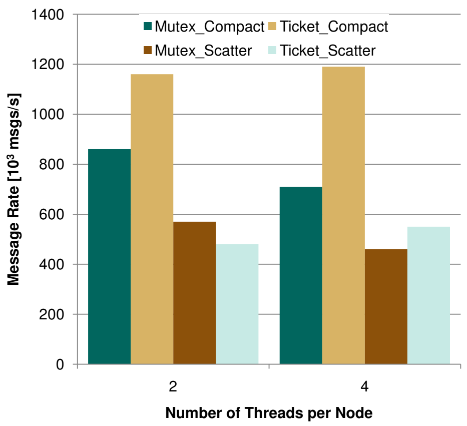


(c) Dangeling requests analysis

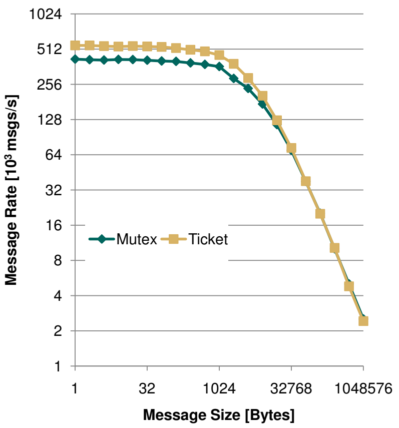
Figure 3. Analysis of unfair arbitration and its consequence on communication progress in the point-to-point throughput benchmark



(a) Dangling requests analysis



(b) Effect of degree of concurrency and thread binding on throughput performance with 1-byte messages



(c) Throughput performance with varying message sizes and 8 threads per node

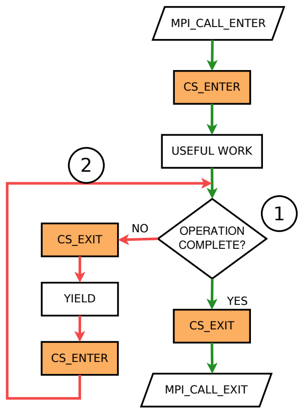
Figure 5. Analysis and performance comparison between using mutex or the ticket lock with the throughput benchmark

operations with ticket locks (Figure 7). More specifically, we use three ticket locks: `ticket_H` for the high-priority threads in the main path; `ticket_L` for the low-priority threads in the progress loop; and `ticket_B` for the high-priority threads to block low-priority ones. We emphasize that using a ticket lock for the high-priority threads to block the low-priority ones is necessary. Failing to do so may generate lock monopolization in favor of low-priority threads.

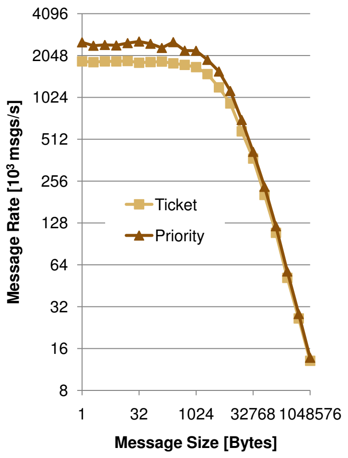
A consequence of prioritizing the main path is feeding the runtime with requests at a higher rate than with the ticket lock. Our design also maintains a low overhead in case the prioritization does not improve the communication progress. Thus, we expect to have some cases with performance improvements and others with performance similar to that of the flat ticket or with a slight overhead. In the following experiment we performed an all-to-all communication pattern named N2N. The benchmark is derived from the multithreaded throughput benchmark with the exception that each process sends/receives a continues stream of messages to and from all the other processes. The results in Figure 6b show that the priority lock improves the throughput of the N2N benchmark by 33% on average for messages below 32 KB. Here, executing the main path is more critical than with the point-to-point communication pattern. In the point-to-point throughput benchmark, threads can match any message since they all come from the same source, the same communicator, and all have the same tags. In the N2N benchmark, however, threads cannot match certain messages. For instance, if a thread is blocked at the entrance of the main path while attempting to post a receive for a certain process, another thread inside the runtime may process the incoming message and put it in the unexpected queue. As a result, the request matching is delayed, slowing the issue and matching of requests and thus the overall communication progress. Prioritizing the main path solves this issue by promoting request generation.

6. Evaluation

In this section we compare the performance of the original design with that of our methods using microbenchmarks, application kernels, and a genome assembly application.



(a) Thread safety in a MPI runtime



(b) N2N throughput performance with 4 processes

Figure 6. (a) Simplified flow diagram of a generic thread-safe MPI implementation with a global critical section. We distinguish two main paths: (1) a *main path* and (2) a *progress loop*. (b) Performance comparison of the ticket and priority locks with the N2N throughput benchmark.

6.1 Microbenchmarks

We begin by presenting results with multithreaded two-sided point-to-point throughput and latency benchmarks. We also evaluate MPI one-sided operations with asynchronous progress.

6.1.1 Two-Sided Communication

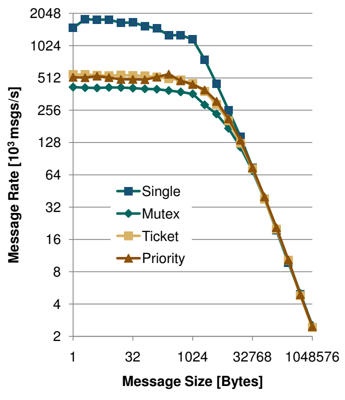
Figure 8 summarizes the results with throughput and latency benchmarks of all methods using 8 threads per node. In addition, we provide a comparison with single-threaded performance (i.e. `MPI_THREAD_SINGLE`). The latency benchmark was derived from `osu_latency` of the OSU microbenchmarks suite. We observe that the throughput achieved with the ticket and priority methods are similar (Figure 8a) and outperform mutex but are only 36% that of single-threaded performance. Figure 8b shows that the ticket method reduces latency by up to 3.5x over mutex. The priority method adds around 11% overhead over the ticket method for small messages but performs similarly with large messages. The latency with the ticket method is 1.66x that of the single-threaded approach with messages below 128 bytes. Surprisingly, however, multithreaded latency with the ticket and priority methods is up to 3.6x better than with a single thread for messages larger than 128 bytes. The reason is that the multithreaded communication issues several requests inside the runtime instead of an individual request, as in the single-threaded case, and helps feed the network resources. For small messages, thread synchronization and runtime contention hide this benefit.

6.1.2 Remote Memory Access with Asynchronous Progress

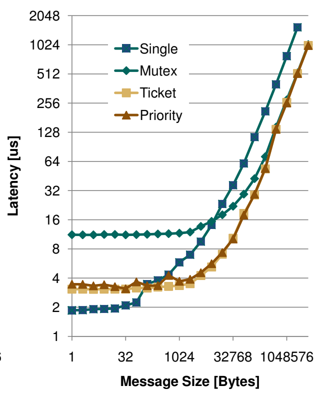
Remote memory access (RMA) provides a powerful model where processes can access memory outside their address space and even outside the physical node they are running on. This concept is at the heart of many programming models, such as Global Arrays (GA) [23], that offer a *shared-memory* view on top of distributed-memory systems. Here we evaluate the performance of ARMCI-MPI [10], an implementation of the Aggregate Remote Memory Copy Interface (ARMCI) [24], which uses MPI one-sided operations. ARMCI-MPI is an important interface that is used, for instance, by the chemistry package NWChem [28].

```
/* High/Low priority ticket locks*/
ticket_lock_t ticket_H;
ticket_lock_t ticket_L;
/* Lock to block lower priority  */
ticket_lock_t ticket_B;
/* Let high priority go through  */
int already_blocked;

high_priority_acquire_lock()
{
  ticket_acquire_lock(ticket_H);
  if(!already_blocked)
  {
    ticket_acquire_lock(ticket_B);
    already_blocked = true;
  }
}

low_priority_acquire_lock()
{
  ticket_acquire_lock(ticket_L);
  ticket_acquire_lock(ticket_B);
}

high_priority_release_lock()
{
  if(last high priority thread)
  {
     /* Let low priority pass */
     ticket_release_lock(ticket_B);
     already_blocked = false;
  }
  ticket_release_lock(ticket_H);
}

low_priority_release_lock()
{
  ticket_release_lock(ticket_B);
  ticket_release_lock(ticket_L);
}
```

Figure 7. Priority locking pseudo-algorithm with two levels of priority



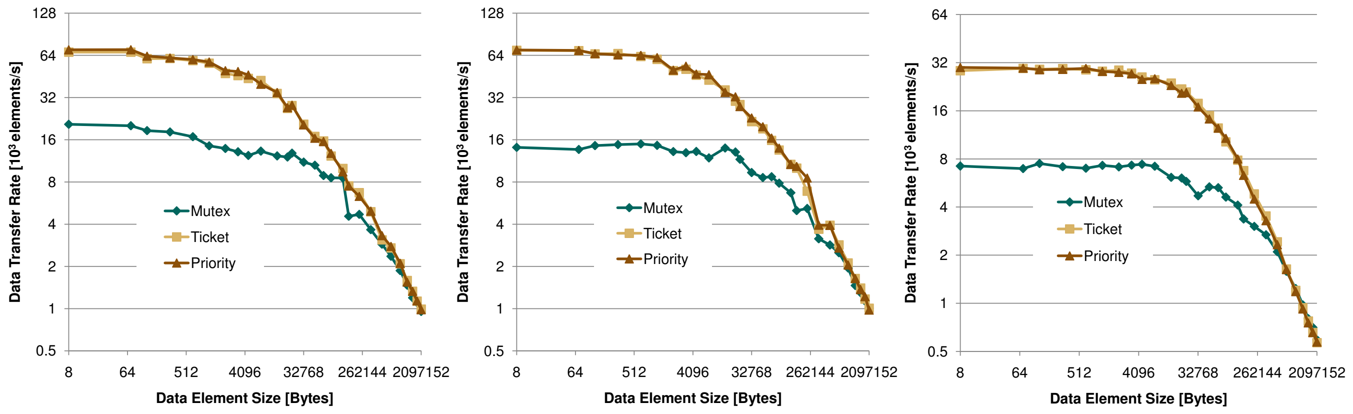
(a) Throughput



(b) Latency

Figure 8. Performance comparison between mutex, ticket, priority, and single-threaded execution with the two-sided point-to-point throughput and latency benchmarks

We conducted an experiment in which one process does RMA operations (put, get, and accumulate) to or from other processes



(a) Put (b) Get (c) Accumulate

Figure 9. Performance comparison of all methods when doing RMA contiguous data transfer using ARMCI-MPI with asynchronous progress

on contiguous data. This benchmark is single threaded; however, we enabled MPICH asynchronous progress that triggers progress on communication in the background by forking an additional thread. Thus, when asynchronous progress is enabled, MPICH uses internally `MPI_THREAD_MULTIPLE` since two threads are running concurrently inside the runtime. The results of this experiment are shown in Figure 9 when running with 8 processes. Although only two threads are running concurrently, we notice a substantial performance difference between mutex and our methods. The reason is that the progress thread, which is most of the time in the progress loop, heavily monopolizes the lock since it does not do useful work most of the time. Thus, enforcing fairness produces a tremendous speedup. We also note that our methods improve performance up to 5x over mutex. Similar to the two-sided communication results, the difference between ticket and priority is not perceptible.

6.2 Kernels

In this section we evaluate some computational kernels often encountered in real applications.

6.2.1 The Graph500 Benchmark BFS

The Graph500 benchmark is a communication-intensive code used for ranking large-scale systems in terms of graph processing capabilities [4]. It is composed of multiple kernels, but we consider only breadth-first search (BFS) in this work. More specifically, the baseline algorithm is an MPI-only level-synchronized BFS that relies on nonblocking point-to-point MPI communication for data exchanges. We do not discuss the details of the BFS reference implementation here; readers can refer to the work by Suzumura et al. [26] for a detailed description. Our hybrid MPI+OpenMP implementation extends the MPI-only design by allowing multiple threads to cooperate for computation and independently communicate with remote processes. Moreover, both computation and communication are lock-free and atomic-free, inspired by the single-node implementation of Chhugani et al. [8]. Each thread maintains outgoing buffers corresponding to each remote process and one buffer for incoming messages. Threads repeatedly check for completed requests using `MPI_Test` and eventually do computation and generate new outgoing requests.

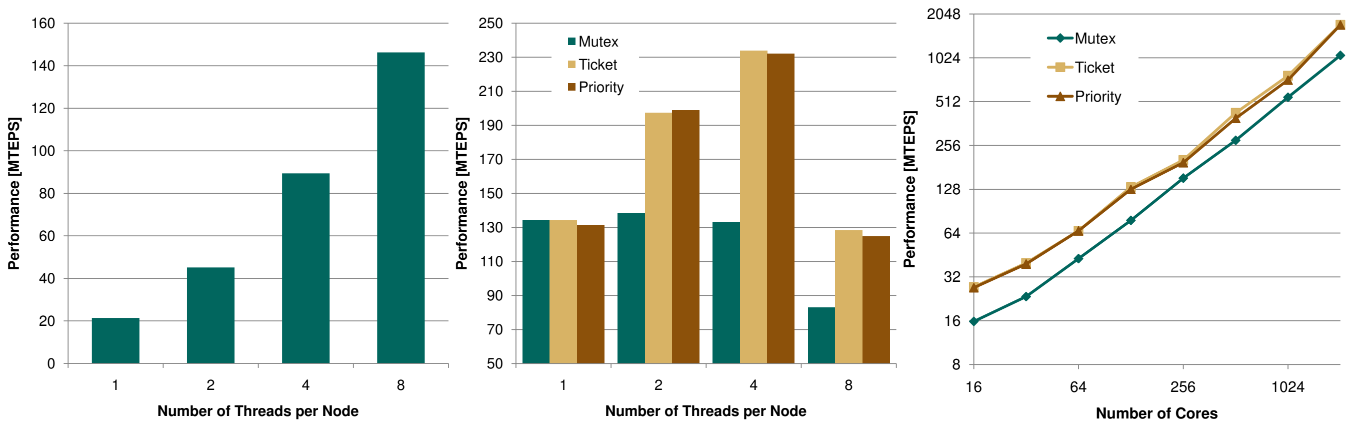
First, we evaluated the performance of our implementation on a single node (Figure 10a). The problem size is expressed by the scale of the graph in terms of number of vertices.[3] We notice that our implementation scales linearly up to 4 cores and loses 10% efficiency with 8 cores. The loss in efficiency is likely due to intersocket data movements since our implementation is not socket-aware. We conducted simple thread-scaling experiments similar to those of Section 5.1. Here, the difference with the throughput benchmark is that threads are doing computation in addition to MPI communication. Consequently, the time spent by one thread inside the MPI runtime may overlap with the computation of another thread. Combined with a parallelized computation, speedups over a single-threaded execution may occur. Indeed, the results in Figures 10b show that speedups do occur when threads are located on the same socket and a fair lock is used with up to 4 threads. With mutex no speedup is apparent, suggesting that the unfair arbitration generates contention and consequently wastes the speedup of the parallel computation. When both sockets are involved, thread synchronization across sockets is an obvious bottleneck; but unlike mutex, our methods avoid slowdowns compared with using a single-threaded method. Figure 10c shows the results of a weak-scaling performance comparison of all methods. We notice performance improvements close to 2x for our methods. The priority method does not show signs of superiority in this case. We explain this by the fact that threads do not busy wait in the progress loop since they use only immediate `MPI_Test` calls. That is, all threads always have the same high priority inside the runtime.

6.2.2 3D 7-Point Stencil Kernel

Stencil codes are a class of iterative methods found in many scientific and engineering applications. Here, the problem domain is iteratively updated by using the same computational pattern, called a *stencil*. We implemented a hybrid MPI+OpenMP 3D 7-point stencil code that simulates a heat equation. Our decomposition methodology tries to reduce the internode communication by dividing the domain along all dimensions, while we avoid splitting the process subdomain along the most strided dimensions for better cache performance. The basic communication method is to perform nonblocking send/receive operations at each iteration followed by `MPI_Waitall` to wait for all the requests. Common hybrid stencil codes typically require the `MPI_THREAD_FUNNELED` threading support, where the computation is done in parallel by a team of threads but only the main thread is driving the communication. In our implementation, all threads independently do computation and communication and synchronize only at the end of an iteration.

We conducted a strong-scaling experiment on 64 nodes, using all 8 threads per node, while increasing the problem size. The

[3] We use a logarithmic scale: $scale = \log_2(\#vertices)$

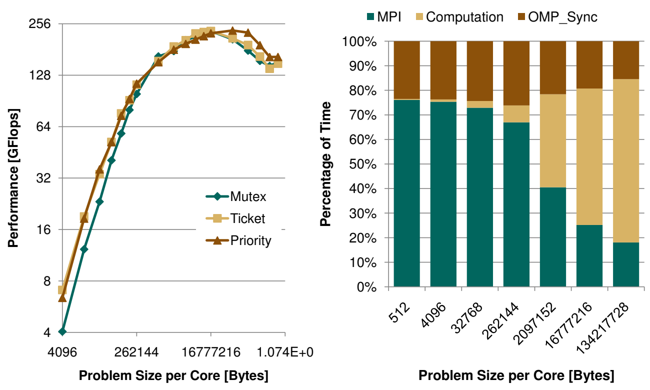


(a) Performance scalability on a single node with a problem size of 24

(b) Thread scaling performance with a compact binding, 16 processes, and a problem size of 28

(c) Weak-scaling performance with one process per node, 8 threads per process and problem sizes from 25 to 32

Figure 10. Performance comparison of all methods with the Graph500 BFS kernel. In (a) the single-node results do not involve MPI processes.



(a) 3D-stencil strong scaling

(b) 3D-stencil execution breakdown

Figure 11. Strong-scaling comparison of all methods with the 3D 7-point stencil kernel

results in Figures 11a and 11b show that our methods improve performance for relatively small problems (<= 1 MB per core). The reason is that, as demonstrated by our microbenchmarks, the runtime contention is more critical for relatively small messages; in addition, the computation takes more time than does the communication, as shown in Figure 11b. Arguably, the benefit of our methods is less pertinent for stencil applications on this typical platform, since bigger problems are often run in production. Nevertheless, given the increase in core counts and the trend of reduced memory per core, we expect that our solutions will play a more important role when running stencil applications on platforms with less memory per core, such as current systems equipped with many-core accelerators and future systems. We note that the priority locking does not improve performance over the ticket lock method. The reason is that since threads have few requests (8 receive and send operations), the rate at which the critical section is entered from the main path is negligible compared with polling for communication in the progress loop. That is, most threads will fall back to low priority, which is equivalent to the ticket method. As a result, giving priority to the main path has negligible effect.

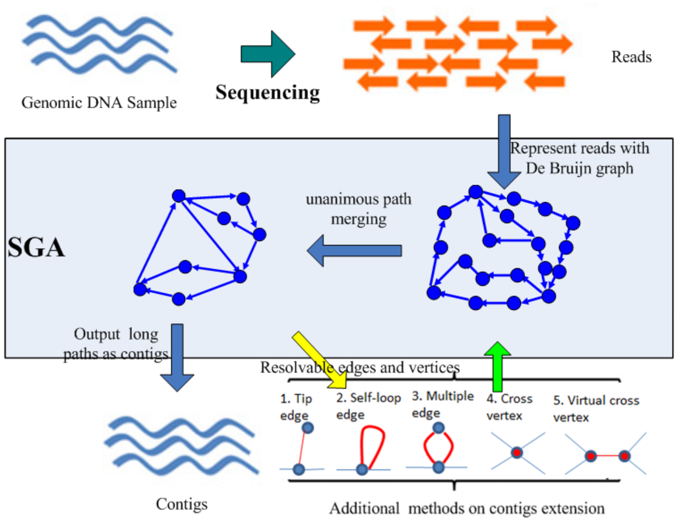
6.3 Genome Assembly Application

Genome assembly is an important process for many fields, such as biological research and virology. It refers to the process of reconstructing a long DNA sequence, such as the chromosome of an organism, from a set of *reads* (short DNA sequences) by aligning and merging them together. The reads originate from automated sequencing machines, which can generate billions of reads to be processed by assembly applications. As a result, these applications exploit high-performance computing systems and explore efficient parallel solutions in order to cope with the ever-increasing generated sequencing data.

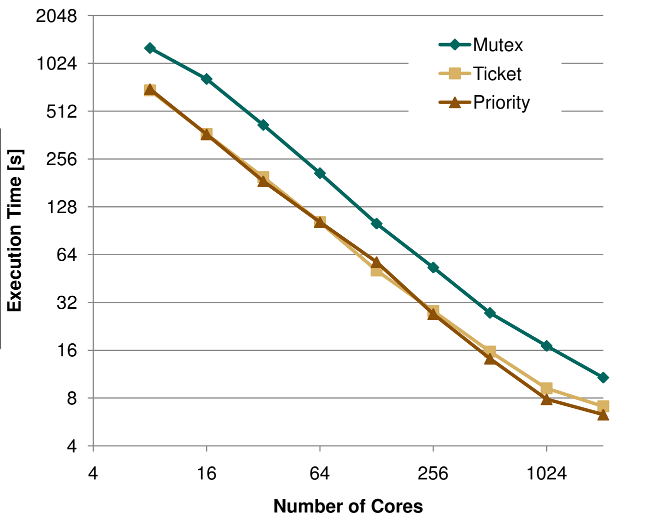
We evaluated our methods using the SWAP-Assembler [21], an application that targets processing massive sequencing data on large-scale parallel architectures. It abstracts the genome assembly problem with a multistep bidirected graph and relies on a scalable framework, SWAP (Small World Asynchronous Parallel) [20], to perform computational work in parallel (Figure 12a). The SWAP framework is implemented on top of MPI to ensure interprocess communication. Each process spawns two threads, one for sending and another for receiving data from other processes using blocking `MPI_Send`/`MPI_Recv` operations. We performed a strong-scaling experiment with a synthetic sequence of 1 million reads, where each read contains 36 nucleotides. The results are shown in Figure 12b. For each data point we used four processes per node and two threads per process to utilize all cores. We observe an average 2x speedup independent of the core count. We note that this improvement in processing time did not incur any modification in the application or the underlying hardware. This fact is important for applications in production environments, since no additional investment is required to speed up the time to solution.

7. Discussion

In our introduction to thread-safety in Section 2, we considered a critical-section granularity as orthogonal to how it is arbitrated. That is, regardless of the granularity, resource acquisition–related issues such as starvation may occur, and appropriate arbitration methods will be required. Thus, we believe that combining those approaches 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 library. A possible model would be to start



(a) Genome assembly process in SWAP-Assembler [21]



(b) SWAP-Assembler strong scaling

Figure 12. Description of the genome assembly process and performance of the SWAP-Assembler using all methods

with a global critical section, explore effective arbitration methods, reduce granularity if high contention persists, and repeat the process.

In addition to the lower overhead of the ticket lock compared with the priority lock (fewer atomic operations), cases may arise where the ticket method will perform better because of fair arbitration. MPI runtimes are sensible to the number of queued requests because the associated internal data structures and algorithm complexity are proportional. Since the priority lock gives high priority to feeding the runtime with requests, ticket may reduce the rate of issuing requests and thus their associated overhead. However, there exists another related issue that affects the cache performance of the MPI runtime and applications. The order in which threads acquire resources can affect the data locality of MPI internal structures, such as shared queues, and thus runtime performance. Similarity, it can also affect the computation part of an application. Assuming that the aggregated threads working sets cannot all fit in the last level of cache, critical section arbitration might impact the amount of data reuse and cache line evictions. These intricacies require further analysis and experimentation.

The idea of using a socket-aware high-priority method that prioritizes threads on the main path and the same socket before moving to another socket seems attractive for reducing intersocket synchronization. However, this approach may lead to starvation. For instance, if the user issues nonblocking operations and waits for them by polling with `MPI_Test`, threads on the same socket will monopolize the lock. We also did not consider a mutex-based priority arbitration. Changing the kernel scheduling is not effective, as discussed in Section 2. Using three mutex locks to establish a two-level hierarchy, similar to our priority lock, is also not effective because mutexes do not guarantee fairness within the same priority class and, worse, low-priority threads can monopolize the lock over the high-priority threads.

8. Related Work

Several researchers have addressed the issue of thread-safety challenges in MPI implementations. Gropp et al. [16] presented an exhaustive thread-safety requirement analysis of MPI functions and their implementation issues. Performance implications of thread safety were exemplified with an efficient algorithm for generating context ids. Thakur et al. also proposed a method to obtain insight into the performance of multithreaded MPI implementations [27]. The method consists of a test suite composed of multiple benchmarks that simulate typical application scenarios. This method is useful for comparing different multithreaded MPI implementations or measuring the impact of certain optimizations, such as using a dedicated progress thread. However, it gives only a general performance feedback and does not pinpoint the exact performance bottlenecks.

Efficiency and thread safety can be orthogonal objectives that are difficult to achieve at the same time when designing a software library. Goodell et al. showed that concurrent accesses from multiple threads to MPI objects can be a bottleneck when using reference counts [15] and proposed more scalable solutions. Hoefler et al. identified that multithreaded MPI messaging involving `MPI_Prob` is thread unsafe and that a conventional lock-based implementation is not scalable [17]. They proposed an efficient solution that goes beyond the implementation level and requires changing the MPI standard.

The tradeoff between using threads for efficient intranode computation and relying on multiple processes to drive the network makes tuning a hybrid implementation a difficult task. For instance, application developers try to tune the number of number of processes per node and the number of threads per processes in order to improve performance. This challenge, thread-safety overheads, and a number of programmability considerations pushed the community to consider extending the MPI standard to better support multithreaded communication. As a result, concepts such as MPI Endpoints emerged as an attractive solution, which ensures contention-free multithreaded communication through independent endpoints [11]. However, MPI Endpoints are not yet part of the MPI standard, and no MPI implementation supports this feature at the moment of this writing.

A large body of work has been dedicated to efficient synchronization on multiprocessors. Busy-waiting methods, such as spinlocks, were criticized for their heavy usage of memory barriers and induced coherency traffic. Researchers have studied traditional spinlocks, such as TAS, TTAS, and ticket lock, and proposed alternative locks that spin on local cache lines (e.g., MCS [18]). A recent study by David et al. [9] compared synchronization constructs, including spinlocks, queue-based locks, and software transactional memory on modern SMP architectures. One of their main conclusions was that the *ticket lock*, despite its simplicity, performs well on most platforms and in various contention scenarios. Although fairness was one of the goals addressed by previous works, their main target was to study the performance of the methods in a general context, thus ignoring arbitration policies that could enhance a more specific workload such as the case with communication runtimes.

9. Conclusion

We addressed in this work the MPI runtime inefficiency with multithreaded communication. Our analysis demonstrated that one of the major drawbacks is unfair arbitration inside the MPI runtime caused by using a mutex-based critical section. We then addressed this issue by ensuring FCFS access to the critical section. In addition, we proposed another method that biases the arbitration to favor threads with higher probability of making progress. Our results with various benchmarks, application kernels, and a genome-assembly application showed up to 5-fold improvements over mutex.

Despite those benefits, the performance difference between our methods and a pure-MPI setting with a simple throughput benchmark suggests that room for improvement remains. We are currently investigating even more flexible methods that target reducing wasted time by the threads inside the MPI runtime. One example is selective thread wake-up triggered by events such as message arrival. We are also considering combining fine-grained critical sections with custom lock arbitrations.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 23220003 and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

[1] MPICH. URL `www.mpich.org`.

[2] OpenMP. URL `openmp.org`.

[3] OSU microbenchmarks suite. URL `mvapich.cse.ohio-state.edu/benchmarks`.

[4] *Introducing the Graph 500*, May 2010. Cray User's Group (CUG).

[5] MPI: A message-passing interface standard version 3.0, Sept. 2012. URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`.

[6] P. Balaji, D. Buntinas, D. Goodell, W. D. Gropp, and R. Thakur. Fine-grained multithreading support for hybrid threaded MPI programming. *Int. J. High Perform. Comput.Appl.*, 24:49–57, Feb. 2010.

[7] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[8] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), 2012*, pages 378–389, May 2012.

[9] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[10] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the global arrays PGAS model using MPI one-sided communication. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 739–750, May 2012.

[11] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling MPI interoperability through flexible communication endpoints. 2013.

[12] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. 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, pages 11–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] U. Drepper. Futexes are tricky. Dec. 2005.

[14] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *AUUG Conference Proceedings*, page 85, 2002.

[15] D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. de Supinski, and R. Thakur. Minimizing MPI resource contention in multithreaded multicore environments. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 1–8, Washington, DC, USA, 2010. IEEE Computer Society.

[16] W. Gropp and R. Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Comput.*, 33:595–604, Sept. 2007.

[17] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sept. 2010.

[18] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[19] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 269–278. ACM, 1991.

[20] J. Meng, J. Yuan, J. Cheng, Y. Wei, and S. Feng. Small world asynchronous parallel model for genome assembly. In J. Park, A. Zomaya, S.-S. Yeo, and S. Sahni, editors, *Network and Parallel Computing*, volume 7513, chapter Lecture Notes in Computer Science, pages 145–155. Springer Berlin Heidelberg, 2012.

[21] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji. SWAP-assembler: scalable and efficient genome assembly towards thousands of cores. *BMC Bioinformatics*, 15(Suppl 9):–2, 2014.

[22] I. Molnar. The native POSIX thread library for Linux. Technical report, 2003.

[23] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[24] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, May 2006.

[25] C. Pheatt. Intel^{-} threading building blocks. *J. Comput. Sci. Coll.*, 23 (4):298–298, Apr. 2008.

[26] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka. Performance characteristics of graph500 on large-scale distributed environment. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 149–158, Nov. 2011.

[27] R. Thakur and W. Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Computing*, 35(12):608–617, 2009.

[28] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.