Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs

Thananon Patinyasakdikul*, David Eberius*, George Bosilca* and Nathan Hjelm†
*University of Tennessee
Knoxville, TN 37921
†University of New Mexico
Albuquerque, NM 87106

***Abstract*—The Message Passing Interface (MPI) has been one of the most prominent programming paradigms in high-performance computing (HPC) for the past decade. Lately, with changes in modern hardware leading to a drastic increase in the number of processor cores, developers of parallel applications are moving toward more integrated parallel programming paradigms, where MPI is used along with other, possibly node-level, programming paradigms, or MPI+X. MPI+threads emerged as one of the favorite choices in HPC community, according to a survey of the HPC community. However, threading support in MPI comes with many compromises to the overall performance delivered, and, therefore, its adoption is compromised.**

This paper studies in depth the MPI multi-threaded implementation design in one of the leading MPI implementations, Open MPI, and expose some of the shortcomings of the current design. We propose, implement, and evaluate a new design of the internal handling of communication progress which allows for a significant boost in multi-threading performance, increasing the viability of MPI in the MPI+X programming paradigm.

***Keywords*—message passing, threads, hybrid MPI+threads**

I. Introduction

The Message Passing Interface (MPI) is nearly ubiquitous in HPC (according to [1] more than 90% of Esascale Computing Project [ECP] and ATDM application proposals use it either directly or indirectly). Therefore, the availability of high-quality, high-performance, and highly scalable MPI implementations which address the needs of applications and the challenges of novel hardware architectures is fundamental for the performance and scalability of parallel applications.

The MPI standard provides an efficient and portable communication-centric application programming interface (API) that defines a variety of capabilities to handle different types of data movements across processes, such as point-to-point messaging, collective communication, one-sided remote memory access (RMA), and file support (MPI-IO) [2]. This ensemble of communication capabilities gives applications a toolbox for satisfying complex and irregular communication needs in a setup that maintains portability and performance across different hardware architectures and operating systems. Owing to these characteristics, many scientific applications have adopted MPI as their communication engine and, therefore, rely on the efficiency of the MPI implementation to maximize the performance of their applications.

Recent hardware developments toward heavily threaded architectures have shifted the balance of computation vs. communication in favor of computations, which have become faster and more energy efficient. Over the last decade alone theoretical node-level compute power have increased 19×, while bandwidth available to applications has seen an increase by a factor of only 3×, resulting into a net decrease for byte per floating-point operation (FLOP) of 6× [3]. An increased rate of computations needs to be sustained by a matching increase in memory bandwidth, but physical constraints set hard limits on the latency and bandwidth of data transfers. The current solution to overcome these limitations has increased the number of memory hierarchies, with orders of magnitude variation in cost and performance between them. Essentially, current architectures represent execution environments where data movements are the most performance-critical and energy-critical components. This shift has greatly impacted the traditional programming approach where each computational core corresponds to a unique process, and all data movement, including at the node level, passes through a message passing layer. As the intra-node inter-process communication costs started to rise, efforts began to move applications toward a more dynamic and/or flexible programming paradigm.

While communication libraries can be improved, using a combination of processes and threads provides an approach that is capable of better relieving the pressure on the memory infrastructure, as there is no explicit communication between threads in the same process. However, while the use of multiple threads to alleviate intra-node data movement seems like a reasonable approach, this generates an entire set of new challenges, both at the programmability level and at the communication level. Threads behave better when they are loosely coupled, but more flexibility translates into reduced ordering between actions in different threads, including communication. Out-of-order communication is a chronic symptom of lack of send determinism in applications [4], and an epitome of out-of-sequence or unexpected messages. In a communication paradigm other than MPI, this could be a minor issue (as an example, in an Active Message [5] context), but the MPI API was designed with a different set of requirements in mind and is not necessarily compatible with such usage.

Current communication libraries struggle to efficiently support a large number of concurrent communications, imposing artificial limitations on the latency and the injection rate of messages. With that in mind, we propose in this paper several

978-1-7281-4734-5/19/$31.00 ©2019 IEEE

strategies to enhance MPI's performance in multi-threaded environments through an increased level of concurrency—for one-sided and two-sided communications—for communication progress and for message matching. We discuss our designs and implementation in section III and evaluate it with the Multirate benchmark [6] and a multi-threaded RMA benchmark [7] [8], along with the information from MPI's internal software counter [9] in section IV.

II. Background

In this section we provide a high-level background of important internal MPI operations for handling user-level communications for interacting with the hardware and the overall design of MPI in multi-threaded environments.

A. MPI Threading Level

The MPI-3.1 standard [2] provides four levels of threading support. During MPI initialization, more precisely during MPI_Init_thread, users can marshal with the MPI implementation the desired thread level for the application. However, most MPI implementations only provide thread-protection when the user initializes MPI with MPI_THREAD_MULTIPLE as it is the only mode that allows thread concurrency, which is the focus of this study.

B. Progress Engine

The progress engine is not a requirement from the MPI standard, but most MPI implementations adopted this scheme. From a high-level perspective, the MPI progress engine is the component that ensures communication progress, either by moving bytes across the hardware, ensuring the expected message matching, or guaranteeing MPI's first-in, first-out (FIFO) message order requirement. From an implementation perspective, the progress engine is the central place where every component in an MPI implementation registers its progressing routine—such as polling for incoming messages, processing pending outgoing messages, including messages for collective operations, or reporting completion to the user level.

As the MPI standard does not provide an API for explicitly progressing messaging, calls into the MPI progress engine occur under the hood during calls to other MPI routines. The decision to enter the progress engine or not on a given MPI function call is up to the MPI implementation, with the exception of blocking routines such as MPI_Send, MPI_Recv or MPI_Wait where message progression, at least related to the operation itself, is mandatory.

The main purpose of the progress engine is to give the MPI implementation the opportunity to check for message completion events from the network, and to ensure timely progress on non-blocking communications. MPI usually reads entries from the completion queues (CQs) for completion events on a particular network endpoint. Completion events can be from both incoming and outgoing messages. In the case of outgoing message completion, MPI marks the corresponding send request as completed, and doing so might release the user from a blocking call such as MPI_Send.

C. Matching Process

The matching engine is another important piece of an MPI implementation for handling incoming messages, as it is responsible for the correct matching of sends and receives. For single-threaded applications, the MPI standard offers the guarantee that all messages between a source and destination pair on the same MPI communicator are matched in a non-overtaking manner, ensuring that the send order is the matching order (or a FIFO ordering). This simplifies the semantics as it ensures that, in single-threaded applications, messages are always delivered in each communicator in a deterministic order. However, at the network level the story is different, as for performance and routing reasons networks do not provide any ordering guarantee by default and the messages might be delivered in an arbitrary order. This requires the MPI library to implement a software solution to provide users with the required message ordering guarantee. For multi-threaded usage, the MPI standard only guarantees message ordering within a single thread. Messages sent from different threads are only guaranteed to happen in some serialized order, as MPI communications, even blocking, are not synchronizing.

The algorithms to provide message ordering may be different for each MPI implementation, but they share a common approach: generate a sequence number for each message and pack it within the message header. To simplify management, this sequence number is generally per peer, per communicator. The receiver extracts the sequence number from the incoming header and uses it to ensure messages are processed in the same order they were sent. Any message arriving out-of-sequence needs to be saved for matching at a later time when that message sequence number is called for. The implementation has to allocate the necessary memory to store the out-of-sequence messages, making this operation more costly. Luckily, in a single-threaded scenario, the occurrence is usually rare, and therefore the cost is negligible. However, this is not the case for multi-threaded MPI. In the scenario—with multiple threads concurrently sending messages on the same communicator to the same destination MPI process—given the nature of their non-deterministic behavior, threads can easily compete and send the messages out of order. With more likelihood of out-of-sequence messages, multi-threaded MPI could suffer significant performance degradation as the number of threads increases due to increased stress on the matching process.

After the MPI implementation successfully validates the sequence number of an incoming message, the message is matched against a queue of the user's posted receives. This code region is a critical section and must be protected with a lock in a multi-thread scenario to prevent concurrent access to the queue. For example, races can occur when threads are simultaneously posting receives; or when a thread adds a request to the posted receive queue while another thread is in the progress engine, trying to match an incoming message with a request on the same queue.

D. Remote Memory Access

In addition to two-sided communication the MPI-3.1 standard provides support for one-sided (RMA) communication. This support allows an MPI implementation to directly expose hardware Remote Direct Memory Access (RDMA), a feature which is present on some high-performance networks (e.g., Infiniband and Cray Aries). This allows the MPI implementation to offload communication directly to the hardware. In addition, the one-sided model separates communication (data movement) from the synchronization (completion). There is no need for any explicit matching for one-sided communication removing a potential multi-threaded bottleneck. This makes RMA well suited for multi-threaded applications.

The current MPI 3.1 standard provides support for three different types of one-sided communication operations: put (remote write), get (remote read), and accumulate (remote atomic); and support for two classes of synchronization: active-target (fence, post-start-complete-wait), and passive-target (lock, flush). Active-target requires the target MPI process of an RMA operation to participate in the synchronization of the window. It is not well suited for multi-threaded applications as all synchronization needs to be funneled through a single thread. Passive-target flush, on the other hand, does not require the target of an RMA operation to participate in either the communication or synchronization and allows for concurrent synchronization. For this study we focus exclusively on the passive-target mode (MPI_Win_flush).

III. Design and Implementation

A. Resource Allocation

One major difference between using multiple MPI processes versus a single MPI process with multiple threads is the resources allocated for MPI operations. Resources such as buffer pools, network contexts and endpoints, or completion queues (CQs) are generally created per MPI process. In the process-to-process communication model, with this single producer–single consumer relationship, resource contention is limited. In the case of multiple threads in the same MPI process, these resources have to be protected, as concurrent access to a resource may not be supported or create race conditions that could compromise the correctness of the communication—or even corrupt the state of the MPI library. At the same time, this protection adds an extra cost to the operation and often increases with the number of concurrent threads.

B. Communication Resources Instance

There are a variety of critical internal MPI resources that must be protected in a multi-threaded environment, such as the network endpoints, network contexts, and CQs. In existing MPI implementations, a single network context is typically created per MPI process and a single network endpoint per source/destination pair. The CQ is usually attached to the network context to store completion events. For multi-threaded MPI, access to both network contexts and their CQs may have to be protected, thus creating a potential bottleneck.

To give multi-threaded MPI a fair chance, more resources need to be allocated for the entire MPI process. We use the concept of a Communication Resources Instance (CRI) to encompass resources such as network contexts, network endpoints, and CQs with per-instance level of protection to perform communication operations. The MPI implementation can allocate multiple CRIs internally for multi-threaded needs.

Currently, there is no interoperability between threading frameworks such as POSIX threads and MPI; therefore, the MPI implementation does not have a standardized way to get the number of threads that will be used for MPI communication from the application. Thus, it is challenging for the implementation to assess the proper number of CRIs to allocate. That being said, an implementation can provide the user with a way to give a hint via environment variable(s), MPI info key(s), or other means (MCA parameters [10] for Open MPI [11] or the new MPI control variables MPI_T_cvar) to let the implementation know how many threads the application intend to use for concurrent MPI operations. The implementation can then allocate the CRIs accordingly. In our implementation, MPI allocates a set of CRIs into a resource pool and creates a centralized body to assign the allocated instances to threads.

Ideally, there should be at least a one-to-one thread to CRI mapping to completely eliminate the potential for lock contention. However, in some cases external constraints limit the capability of creating CRIs. As an example, the Cray Aries network devices have a hardware limit on the number of network contexts users can create, so the design must also accommodate for cases where the number of CRIs is less than the number of threads.

Giving more resources to the threads might not be sufficient to increase communication performance for two-sided communication as the MPI implementation still serializes the calls to both the send operation and progress engine to prevent any potential race conditions. In order to benefit from more available resources, both the send and receive paths have to be redesigned to allow for more parallelism while maintaining thread safety and continuing to ensure the expected matching semantic.

C. Try-Lock Semantics

Using locks to protect critical resources is one of the simplest and most popular approaches to ensure thread safety for critical sections. These locks act as a funnel when multiple threads are going through the same code path, as lock contention will cause threads to block. In some cases, especially when the critical section is only performance critical (not correctness critical), we can mitigate the funneling effect by using try-lock semantics, a non-blocking version of lock, where the lock acquisition returns immediately if it fails to acquire the lock.

Try-lock semantics provide more opportunities for parallelism. When the lock is already taken, we can be certain that a thread is progressing that particular code path, and therefore, the current thread can move on and try to pick up another code

Algorithm 1 Utilizing multiple CRIs to allow concurrent sends.

```
1: function INIT
2:     for i ← 1, NumInstances do
3:         instance[i] ←CREATE-INSTANCE()

4: function SEND(msg)
5:     k ← GET-INSTANCE-ID()
6:     LOCK(instance[k] → lock)
7:     NETWORKSEND(instance[k], msg)
8:     UNLOCK(instance[k] → lock)

9: function GET-INSTANCE-ID–ROUND-ROBIN
10:    static current_id ← 0
11:    ret = current_id
12:    current_id ← current_id + 1
13:    return (ret mod numInstances )

14: function GET-INSTANCE-ID–DEDICATED
15:    static thread_local my_id ← undefined
16:    if my_id is defined then
17:        return my_id
18:    else
19:        my_id ← GET-INSTANCE-ID()–ROUND-ROBIN
20:        return my_id
```

path to execute, or become a helper thread and complete other menial work.

In the following subsections, we describe how we leverage try-lock semantics along with the communication resources instances (we will further refer to them as CRIs or "instances" in the following sections) to alleviate resource contention from MPI's internal receive path.

D. Concurrent Sends

For the MPI implementation to perform a send operation, it needs access to a network endpoint. In the multi-threaded case, the implementation usually protects the network context with a lock. In our design, the network context is associated with a CRI along with other resources, allowing us to move the protection down the software stack, basically from per-endpoint to per-instance. This move leads not only to finer grains locks but also to an increased parallelism in the communication infrastructure, allowing multiple threads to perform send operations simultaneously on different instances. Optimally assigning a CRI to a thread is a difficult question, and we focus on evaluating two strategies: round-robin and dedicated (Algorithm 1).

1) Round-Robin Assignment: In this strategy, every time a thread needs to communicate, it first acquires a CRI. The MPI implementation assigns an instance for a single use in a first-come, first-served manner, supported by the use of a circular array. Once the last available instance is assigned, the implementation will recycle the instances and then give out the first instance again. This approach eliminate the possibility of lock contention by assigning a different instance for every call, in exchange of a cheaper, atomic operation. It also improves load balancing by distributing the communication work among the allocated instances.

Algorithm 2 Dedicated instance assignment to give priority to the thread assigned instance before trying to progress others, ensuring eventual progress for every instance.

```
1: function COMMUNICATION PROGRESS
2:     count ← 0
3:     k ←GET-INSTANCE-ID()–DEDICATED

4:     if trylock → instance[k].lock = success then
5:         progress instance[k]
6:         count ← number of completions
7:         unlock → instance[k].lock

8:     if count = 0 then
9:         for i ← 1, NumInstances do
10:            k ← GET-INSTANCE-ID()–ROUND-ROBIN
11:            if trylock → instance[k].lock = success then
12:                progress instance[k]
13:                count ← number of completions
14:                unlock → instance[k].lock
15:            if count > 0 then
16:                return
```

2) Dedicated Assignment: To permanently assign a CRI to a thread, MPI can utilize thread-local storage (TLS), provided either by the threading library (e.g., POSIX threads) or the programming language (e.g., C11, C++11). This approach can only be implemented when the system or the compiler supports TLS, a pretty standard feature nowadays. In our implementation we use the native compiler support either from C11 or the GNU Compiler Collection (GCC). When checking for a CRI to use, the implementation can check if an instance information is stored in TLS. If not, it can assign an instance with a round-robin assignment and save the instance information in the TLS. With the dedicated assignment strategy, there is no possibility of lock contention on the instance as long as the number of threads is lower than or equal to the number of instances allocated. If not, some communicating threads might share the same instance and then introduce lock contention if they simultaneously communicate.

E. Concurrent Progress

Traditionally, Open MPI serializes calls into the progress engine, allowing only a single thread to progress communications. Such a coarse-grained protection under-utilizes the available thread parallelism, and limits the rate of message extraction to the power of a single thread. To allow threads to extract messages concurrently, we removed the serialization from the progress engine and exploited our instance-level protection to provide the required thread safety instead.

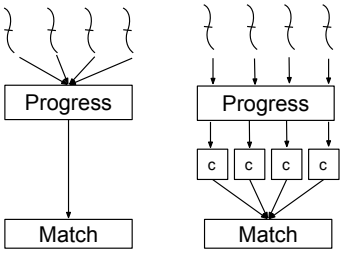


Fig. 1: Matching process is still a serial operation and become a major roadblock for multi-threaded MPI. Serial progress (left), Concurrent Progress with multiple CRIs effectively move the bottleneck to the matching process (right)

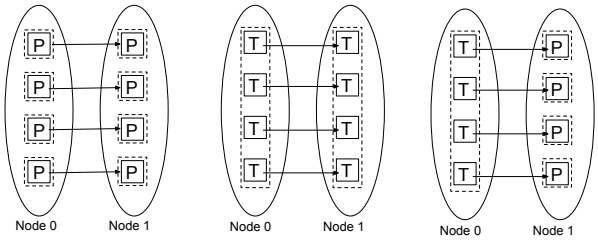


Fig. 2: Different modes of operations in Multirate–Pairwise benchmark binding CPU cores to communication entities.

The progress engine also suffers from the lack of threading information in MPI. When a thread makes a call into the progress engine, it requires an instance to progress. We utilize the same centralized body as for concurrent sends to assign an instance to a thread. The strategies to choose which instance to progress are similar to how we choose the instance for the send path—namely, *Round-robin* and *Dedicated* (Section III-D).

For the *Dedicated* strategy, with a permanent instance assigned to each thread, a few issues need to be addressed. First, the MPI implementation has to make sure that it progresses every allocated CRI to prevent deadlock scenarios where message completion is generated in an instance that is not progressed by the associated thread. Second, the user might destroy the thread and create orphaned CRIs that cannot be reused by other threads. To overcome this limitation, we have each thread try to progress their dedicated instance first, and, if there is no completion event, move on to try progressing other instances. This design will guarantee that every instance will eventually get progressed while still maintaining the optimization benefit from TLS.

Furthermore, the try-lock semantics on the instances become a valuable weapon to the efficiency of concurrent progress design (Algorithm 2). If a thread fails to acquire the lock for an instance, it assumes that another thread is progressing that particular instance, and the current thread can try to pick up another instance to progress or return.

F. Concurrent Matching

The matching process is possibly the only strictly serial operation in the MPI two-sided communication. By changing from a serial progress to a concurrent progress engine, we effectively move the bottleneck to the matching process. As long as the matching process cannot be performed in parallel, it will be challenging to get optimal performance from multi-threaded MPI (Figure 1), as there will always be a protected section in the message reception critical path.

The current message matching design from state-of-the-art, open-source MPI implementations such as MPICH and Open MPI drastically differ. Even in the context of the same MPI implementation, the matching infrastructure can be different depending on the network used (Portals provides hardware matching), the hardware capabilities (AVX provides opportunities for vector matching) and the configured software stack. As an example, Open MPI supports multiple methods for matching, going from hardware matching when available, to a single global queue when using the UCX PML; to a vector fuzzy-matching single global queue [12]; and finally to the default, more decentralized matching in the OB1 PML (with a matching queue per process per communicator with special arrangements for MPI_ANY_SOURCE that has the potential to minimize the contention lock for communications not between the same peers in the same communicator).

A study of optimized or parallel matching is not within the scope of this paper. For this study, we will show the potential of concurrent matching by utilizing OB1, a point-to-point matching layer (PML) component designed to perform the matching process per MPI communicator instead of globally. We can then simulate the concurrent matching process by creating multiple communicators and allowing threads to perform unhindered matching in parallel. While this approach might not be practical for some real-world applications, it is sufficient to demonstrate the potential of multi-threaded MPI.

IV. Experiments

Most of the design strategies described in this paper are generic, and can be applied to different MPI implementations. To assess their benefit and potential performance impact we implemented them in Open MPI, by taking advantage of the Open MPI modular design [13], and utilizing the OB1 point-to-point messaging component (pml/OB1) in conjunction with the *uct* (for Infiniband networks) and *ugni* (for Aries networks) Byte Transport Layer (BTL) components (btl/uct), which were updated to use multiple CRIs. We also modified the Open MPI progress engine (opal_progress) to allow multiple threads in the progress engine.

To gain low-level insights into the different statistics related to the communication engine, we took advantage of Open MPI's built-in Software-based Performance Counters (SPCs) [9] to expose internal MPI information with low overhead. SPCs offer a variety of measurements from the MPI level, such as the number of messages sent/received as well as MPI internal information, such as the number of unexpected or

TABLE I: Testbeds configuration.

	Alembert	Trinitite
Processor	Dual 10-core Intel Xeon E5-2650 v3 @2.3 Ghz Haswell	Dual 16-core Intel Xeon E5-2698 v3 @2.3 Ghz Haswell
Main Memory	64GB DDR4	128GB DDR4
Interconnect	InfiniBand EDR (100 Gbps)	Cray Aries (100 Gbps)
OS	Scientific Linux 7.3	Cray Suse Linux
Compiler	GCC 8.3.0	GCC 8.3.0

out-of-sequence messages, the cost of matching, or the length of the matching queues. For this study, we focus on two of these counters: the number of out-of-sequence messages and the total matching time.

To evaluate the impact of each strategy presented in this paper, we measure the message rate with the Multirate benchmark [6] in pairwise pattern for two-sided communication, and use the RMA-MT benchmark [14] for one-sided communication. We have run several hundred experiments and report in all instances the mean and the standard deviation in the figures, which should be noted is consistently very small.

Multirate–pairwise spawns pairs of communication entities which can be mapped to either an MPI process or a single thread to perform communication simultaneously (Figure 2). For two-sided communication experiments, we perform zero byte communications as they allows us to capture only the cost of the message envelope. Open MPI sends necessary matching information to be matched on the receiver side without any payload (the size of this matching header is small in Open MPI, around 28 bytes).

RMA-MT is a benchmark developed at Sandia National Lab (SNL) and Los Alamos National Laboratory (LANL) to stress-test an MPI implementation under a heavy multi-threaded Remote Memory Access (RMA) workload. The experimental testbeds specifications are presented in Table I, and are Alembert from the University of Tennessee (IV-A through IV-E) and LANL's Trinitite clusters (IV-F).

A. Concurrent Sends

Figure 3a demonstrates the effect of allocating additional internal resources, CRIs. We use the original design, which serializes progress and therefore only allows a single thread to perform the network extraction at a time. By only introducing changes on the sender side, these experiments demonstrate the impact of increasing resource availability, thus decreasing contention on the send path. This allows multiple threads to reach the lowest network level simultaneously, each in a different context, and to technically perform send operations concurrently. We employ the two strategies described in Section III-D to assign an instance to a thread: *round-robin* and *dedicated* presented by solid and dashed lines, respectively. Each color represents a different number of instances allocated for the experiment.

The red lines represent the base performance—the original multi-threading support in Open MPI—with a single instance shared between all threads. The impact of contention on the shared resource become visible very early, starting as soon as 2 threads. The scenario under investigation here is very demanding. As the only payload is the MPI matching envelope, threads sharing the same instance will continously fight for the same protection lock, and the lock will therefore always be contested.

Ideally, a one-to-one mapping from a thread to an instance should be the starting point to maximize the performance as, if handled correctly, even when all threads use the network there can be no contention on any instances. We achieve this scenario by employing the *Dedicated* strategy on this experiment, represented by the blue-dashed line (with 20 threads, 20 instances). Just by increasing the number of instances we can achieve a performance gain of up to 100% compared to our original case. If we reduce the number of instances to 10, we see a small performance drop after going over 10 threads as the threads start sharing the instances, thus introducing some congestion (green-dashed line).

Although the *round-robin* strategy (solid lines) does not give the best performance, it significantly softens the effect of the congestion by evenly spreading the CRIs among threads, thus reducing the lock congestion. It is still a viable strategy when *Dedicated* cannot be implemented due to lack of compiler support on the platform.

The performance metrics obtained from SPC are presented along with Figure 3 in Table II. Due to the space constraints, we only present the information from the last data point from the best result of each figure, at 20 thread pairs, 20 instances with *Dedicated* assignment strategy. In general, for serial progress, the SPCs show similar numbers of out-of-sequence messages (up to 90%) with similar time spent in matching.

B. Concurrent Progress

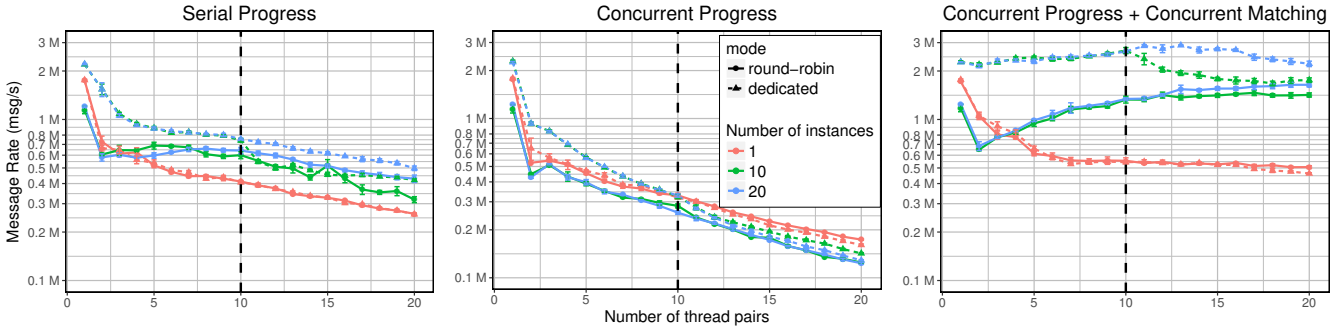
Figure 3b presents the performance impact from concurrent progress. The difference from the above experiment is the concurrent progress, which basically allows multiple threads to execute the progress engine simultaneously.

Concurrent progress hinders the performance instead of boosting it, even with increased parallelism (Figure 3b). The results show a funneling effect as the number of threads increases, regardless of number of instances or the assignment strategy, just as expected. The potential parallelism from concurrent progress is restricted by the next step in the execution path, the matching, and cannot boosts the performance as long as the matching process remains a serial operation; the approach effectively moves the bottleneck from the progress engine to the matching process (Figure 1).

The SPC information from Table II reveals that the MPI implementation is spending up to 300% more time in matching compared to our earlier experiment, which is consistent with our expectations.

C. Concurrent Matching

We relax the constraints on the matching to improve upon the previous case. To simulate a concurrent matching process,



(a) (b) (c)

Fig. 3: Zero byte message rate on different strategies.

TABLE II: Software Performance Counters information from last data point of the experiment in Figure 3. (20 thread pairs, *Dedicated* assignment, total messages = 2,585,600)

	Serial Progress			Concurrent Progress			Concurrent Progress + Matching		
Number of instances	1	10	20	1	10	20	1	10	20
Out-of-sequence messages	2,154,493	2,323,003	2,225,190	2,375,922	2,425,818	2,420,660	15,188	45	0
Out-of-sequence (%)	83.32%	89.98%	86.08%	91.89%	93.82%	93.62%	0.59%	$\approx$ 0%	0%
Match time (ms)	2,732	2,622	2,738	8,553	7,944	8,069	476	430	389

we create multiple communicators and take advantage of the matching logic in the OB1 PML, with matching queues private to each communicator. Since the pml/OB1 component in Open MPI performs matching per-communicator, this effectively provides support for concurrent matching.

Multirate–pairwise provides an option to assign a communicator per each pair of communicating threads. With a unique communicator per thread pair along with concurrent sends and concurrent progress, this part of the experiment represents the multi-threaded performance when the contention in the matching process is minimal.

Figure 3c highlights a major increase in the message rate for all strategies. Even the *round-robin* assignment (solid lines) shows performance improvement with the number of threads, a completely different outcome from our earlier experiments. The instance assignment strategy seems to perform well even after the number of threads is greater than the number of instances. For this strategy, messages from the same communicator can be sent out from different instances. There are chances that the receiver—as their threads extract the messages simultaneously from multiple instances—will perform matching on the messages from the same communicator and introduce some congestion (Figure 1).

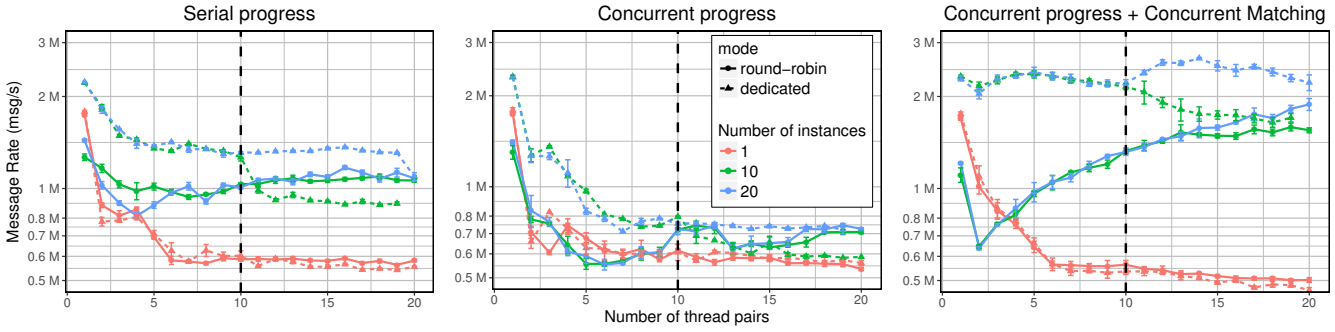
Dedicated assignment gives the best performance as each thread always uses the same network instance in addition to using the same communicator (dashed lines). The blue dashed line represents an ideal scenario with one-to-one mapping from thread to CRI to communicator. The performance scales with the number of threads but drops once the number of threads is large, suggesting other bottlenecks not yet identified. The green dashed line shows the same performance scaling until the threads have to share instances (at 11 threads and over) before dropping off similarly to the blue dashed line.

The information from the SPCs also shows a drastic improvement over earlier experiments as the number of out-of-sequence messages drops significantly after introducing more instances. The match time is minimal, as there is a guarantee for no contention on both the instance and the matching process. While this approach could be implemented in a customized benchmark, using dedicated communicators for each communication thread pair might not be practical for most applications. Nonetheless, this experiment successfully shows that the major bottleneck for multi-threaded MPI currently resides in the matching process contention.

D. Message Overtaking

We can break the matching process into two parts: sequence number validation, and the queue search to match messages with MPI requests. As explained earlier, out-of-sequence messages force the MPI implementation to allocate memory to buffer the message for processing later, which is a costly operation right in the middle of the critical path. It is possible to relax the matching order requirement in MPI, which translate to ignoring the sequence number validation, by providing the MPI info key *mpi_assert_allow_overtaking* to the communicator, allowing MPI to therefore immediately match every incoming message. This info key is not novel, it has been intensely discussed in the MPI Forum, and has been approved for inclusion in the next version of the MPI standard. This study can serve as a further validation of the usefulness of this info key in threaded scenarios.

Allowing the MPI implementation to match every incoming message immediately will lead to high stress on the queue search. When using multiple tags, the queue search is a linear operation where the cost increases with the queue length.



(a) (b) (c)

Fig. 4: Zero byte message rate when the message ordering is not enforced.

When a message is matched out of sequence, the average time to search the queue is increased because the request associated with the message might be at the end of the queue. To fully reap the benefits of message overtaking, we modify Multirate–pairwise to post the receive with a wildcard tag (MPI_ANY_TAG) to force the implementation to always match the incoming message with the first posted receive request, skipping the queue search entirely.

This experiment represents the multi-threaded MPI performance if the matching process cost is minimal. We perform the same set of experiments from earlier with our tweak and demonstrate the result in Figure 4. If we take a look at the serial progress performance (Figure 4a), for a single instance (red lines), we can still see that increasing the number of instances helps in giving some performance boost from the sender side. The message rate flattens out around 500K msg/s and remains unchanged with an increasing number of threads, similarly with our earlier experiment (Figure 3a). This suggests that the source of performance degradation in multi-thread MPI is mostly from the matching process.

Although concurrent progress still shows the same performance drop from matching congestion where multiple threads try to acquire the matching lock, the message rate still flattens out around the same point as serial progress (Figure 4b). While in the last case, with both concurrent progress and concurrent matching (Figure 4c), removing the ordering does not affect the performance because the matching process for this strategy, based on MPI_ANY_TAG, is already optimal.

E. Current State of MPI Threading

In this section, we compare our proposed strategies and with different state-of-the-art MPI implementations on the same configuration of Multirate–pairwise. To get a better understanding of where the threaded performances are overall, we also compare with the process-based mode, where communications—instead of happening between threads—now happen between processes placed on the same nodes as the original threads. Ideally, running on the same hardware with the same communication pattern should yield similar performance, regardless of whether processes or threads are used. Unfortunately, as we demonstrated in Figure 5, at the

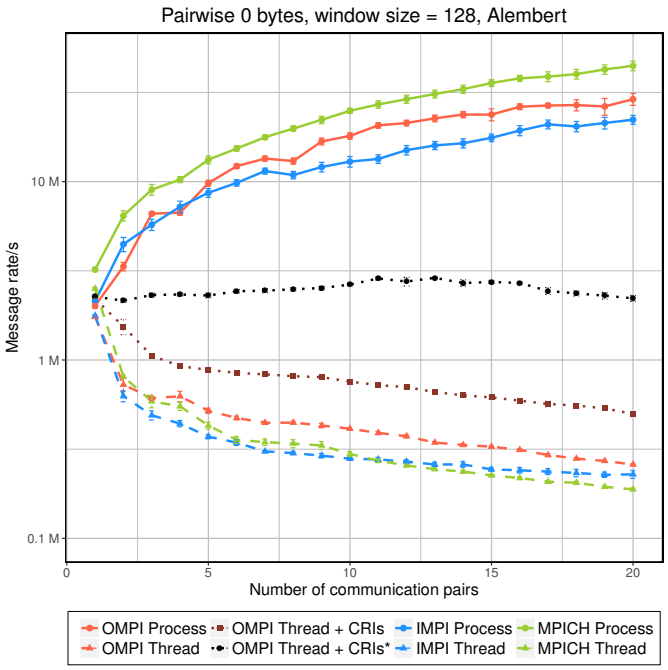


Fig. 5: Zero byte message rate from different state-of-the-art MPI implementations shows disparity between two mode of operations along with our threading improvements. (Note: Log scale on Y-axis.)

current stage of threading support in all MPI implementations, we are far from this ideal scenario.

The MPI implementations presented in this experiment are Intel MPI 2018.1 [15], MPICH 3.3 [16] and Open MPI 4.0.0 [11] with and without our modification. Each MPI implementation was compiled with GCC 8.3.0 with proper optimization flags (except for Intel MPI which is only available as a pre-compiled binary from the vendor).

Figure 5 highlights, using a log-scale Y axis from multi-thread standpoint, that there is little difference between MPI implementations (dashed lines)—they all perform similarly poorly. We observe roughly a 100% performance boost from our base implementation by employing try-lock semantics

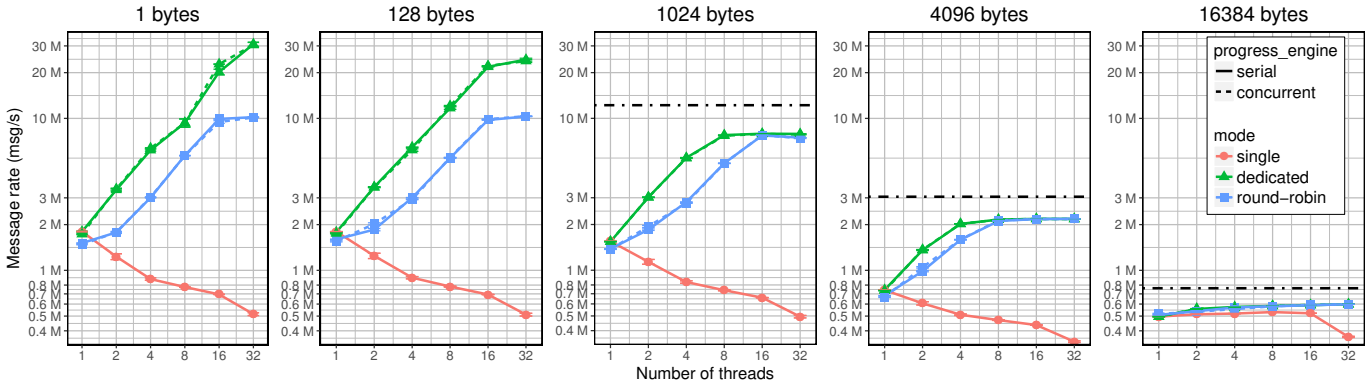


Fig. 6: RMA-MT performance using MPI_Put and MPI_Win_flush on Haswell architecture

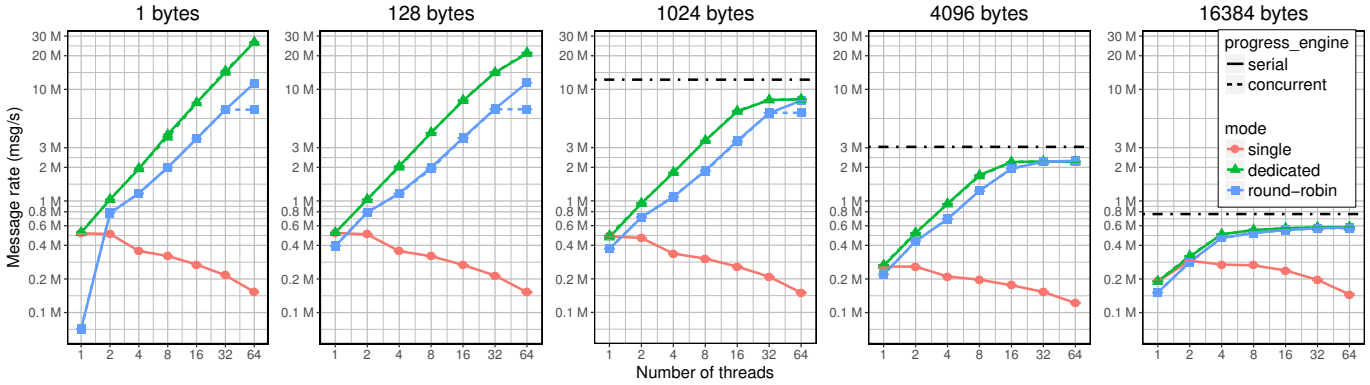


Fig. 7: RMA-MT performance using MPI_Put and MPI_Win_flush on KNL architecture.

with multiple CRIs (dark red), but these results should be put in a larger context and compared with the process-to-process performance. The black dotted line represents the CRI injunction with concurrent progress and concurrent matching, the most optimistic scenario for communicating threads. While we do notice a significant boost in performance, up to 10× compared with the base implementation, we still cannot reach the same level of performance as the non-threading mode, suggesting not yet identified bottlenecks for multi-thread MPI.

F. RMA Performance

To test the performance of our implementation with one-sided MPI, we ran experiments with the RMA-MT benchmark. The experiments were run on the Trinitite system at LANL using both Intel Knights Landing (KNL) and Haswell compute nodes. Open MPI was configured to use the ugni BTL and the RDMA one-sided component (osc). The ugni btl provides support for multiple CRIs for one-sided communication only. By default, the ugni btl will try to detect the number of cores available to the MPI process and will attempt to create one instance per available core. In the case of the RMA-MT benchmark, this creates 32 instances on Haswell nodes and 72 instances on KNL nodes.

All tests were configured to bind each benchmark thread to a dedicated CPU core (-x option). We ran the benchmarks from 1 to 32 threads on Haswell nodes, and 64 threads on KNL nodes, using the MPI_Put operation (-o put) and MPI_Win_flush synchronization (-s flush) with both round-robin and dedicated assignment strategies. This benchmark spawns a user-specified number of threads that for each message size perform 1000 put operations. The first thread then calls MPI_Win_sync to synchronize the window. The results for both Haswell and KNL architectures appear in Figures 6 and 7 where the black horizontal line in each sub-figure represents the theoretical peak message rate for that particular message size.

The results show that the performance when using dedicated instances for threads (triangles) significantly outperforms round-robin (square). The performance difference is similar on both KNL and Haswell nodes. When using a dedicated thread instance, the performance of the RMA-MT benchmark scales almost perfectly with the number of threads. The single instance performance (red) represents the performance before support was added for multiple network instances, where the performance drops with increasing numbers of threads due to the lock contention on a single instance.

As expected, there appears to be little benefit from concurrent progress in this configuration (dashed lines), likely due to the absence of involvement of the target process in one-sided communications, which annul the need for concurrently draining the network and matching the messages.

V. Related Work

Many studies have been conducted investigating ways to improve the efficiency of multi-threaded MPI. [17]–[19] proposed several strategies to minimize locking for MPI internals to mitigate the effect of lock contention, which becomes one of the main performance bottlenecks for multi-threaded MPI. The authors of [20] proposed software offloading to avoid the lock entirely by having one dedicated communication thread, centralizing the MPI operations between threads through a lock-less command queue. [21] [22] [23] [24] investigate alternative thread vs. process approaches and the use of shared memory mapping between MPI processes for intra-node communications, circumventing the use of threads and MPI_THREAD_MULTIPLE and avoid the cost of thread synchronization from MPI entirely.

In this study, we take advantage of the thread synchronization object of Open MPI's progress engine. Threads are bound to the object with events, allowing threads to get notification of event completion. Open MPI leverages the synchronization object to reduce the lock contention in the progress engine, similarly to the study of Dang et al., for MPICH [25].

Si et al. propose interoperability between the MPI and OpenMP runtimes [26] [27] to fully utilize idle application-level threads for MPI communications in many-core environments. Grant et al. studied an approach to aggregate small messages from multiple threads into a larger buffer before sending to peer to avoid the matching overhead incurred per message [28].

There is interest in extending the MPI standard to allow MPI users to create necessary communication endpoints to enhance multi-threaded communication performance by giving threads direct access to the hardware resources [29], [30]. These studies are somewhat similar to our work, but instead of proposing a solution hidden in the MPI software stack, they propose user-level solutions, a possibly more enticing approach for power-users.

A previous study investigated the performance of the multi-threaded RMA support in Open MPI when using multiple device contexts [8]. That work, however, did not look at the performance when binding network resources to threads. The results of this study show that there are additional performance benefits that can be achieved by dedicating a thread to a particular resource.

VI. Optimization Suggestions

In general, the MPI implementations could benefit from allocating more resources for threads to allow them to operate simultaneously. There are several strategies to assign the resources to threads. Our experiments confirm that the ideal approach is to have at least a one-to-one mapping from thread to the resource (dedicated assignment), similar to non-threading environments where each process has exclusive access to its network resources.

For two-sided communication, the likelihood of out-of-sequence messages increases with the number of threads, putting tremendous stress on the receiver side's matching process. Using an MPI info key to allow message overtaking from the application level might help in boosting the performance. However, it might only be suitable for some categories of application that do not rely on message ordering, such as task-based runtimes.

The matching process remains one of the major bottlenecks for two-sided communication, as it is a critical section that has to be protected. This study further demonstrates the potential of multi-threaded MPI if the matching process is parallelized, but while it is possible to argue that all the protection mechanisms can be optimized, it remains true that matching, as imposed by the MPI standard, is inherently sequential. Dropping the matching requirements for messages will either move the MPI two-sided communications performance and scalability toward one-sided communications—which come with their own set of constraints—or push in the direction of Active Messages, a field that has received little interest from the MPI community as yet.

One-sided communication reaps the most benefit from more allocated resources. Without matching process, the performance does not suffer from the funneling effect on the matching process serialization. Our experiment shows good performance scaling with the number of threads. However, one-sided communication imposes the burden of synchronization and programming complexity on the users.

VII. Conclusion

With the hope to make MPI a more suitable communication infrastructure for mixed programming paradigms (MPI+X), we assessed the performance of two-sided communications on several MPI implementations in a multi-threaded scenario. Confronted with an abysmal performance gap between threads-and processes-based communications, we proposed several strategies to address this performance gap, and implemented and evaluated them in the Open MPI library, looking at their impact on both one- and two-sided communications. While we implemented our proposed design in Open MPI, the design is highly portable and can be easily adopted for other MPI implementations. We have also proposed a few potential additions to the MPI standard that would allow for better threading support, topics we plan to continue to investigate in the future.

Our optimizations are partially available in Open MPI release version 4.0 and entirely in the master branch on the official Open MPI GitHub repository.[1]

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. #1664142 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy Office of Science and the National Nuclear Security Administration.

[1] https://www.github.com/open-mpi/ompi

REFERENCES

[1] D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, "Ecp milestone report a survey of mpi usage in the us exascale computing project wbs 2.3. 1.11 open mpi for exascale (ompi-x)(formerly wbs 1.3. 1.13), milestone stpm13-1/st-pr-13-1000."

[2] M. P. I. Forum, *MPI: A Message-Passing Interface Standard Version 3.1*, June 2015, http://mpi-forum.org/.

[3] S. Rumley, M. Bahadori, R. Polster, S. D. Hammond, D. M. Calhoun, K. Wen, A. Rodrigues, and K. B. man, "Optical interconnects for extreme scale computing systems," *Parallel Computing*, vol. 64, no. Supplement C, pp. 65 – 80, 2017, high-End Computing for Next-Generation Scientific Discovery. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819117300170

[4] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 989–1000.

[5] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schau ser, "Active messages: a mechanism for integrated communication and compu tation," in *Computer Architecture, 1992. Proceedings., The 19th Annual Inte rnational Symposium on*. IEEE, 1992, pp. 256–266.

[6] T. Patinyasakdikul, X. Lou, D. Eberius, and G. Bosilca, "Multirate: A flexible mpi benchmark for fast assessment of multithreaded communication performance," in *Submitted to Proceedings of the 26th European MPI Users' Group Meeting*, ser. EuroMPI '19, 2019.

[7] M. G. F. Dosanjh, T. Groves, R. E. Grant, R. Brightwell, and P. G. Bridges, "Rma-mt: A benchmark suite for assessing mpi multi-threaded rma performance," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 550–559.

[8] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving mpi multi-threaded rma communication performance," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 58:1–58:11. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225114

[9] D. Eberius, T. Patinyasakdikul, and G. Bosilca, "Using software-based performance counters to expose low-level open mpi performance information," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017, pp. 7:1–7:8. [Online]. Available: http://doi.acm.org/10.1145/3127024.3127039

[10] J. Squyres, "Modular component architecture," https://www.open-mpi.org/papers/workshop-2006/mon_06_mca_part_1.pdf, [Online; accesed 21-March-2019].

[11] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[12] W. Schonbein, M. G. F. Dosanjh, R. E. Grant, and P. G. Bridges, "Measuring multithreaded message matching misery," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 480–491.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104.

[14] [Online]. Available: https://github.com/hpc/rma-mt

[15] "Intel mpi library software," https://software.intel.com/en-us/mpi-library, [Online; accesed 21-March-2019].

[16] W. Gropp, "Mpich2: A new start for mpi implementations," in *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2002, pp. 7–. [Online]. Available: http://dl.acm.org/citation.cfm?id=648139.749473

[17] D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. d. Supinski, and R. Thakur, "Minimizing mpi resource contention in multithreaded multicore environments," in *2010 IEEE International Conference on Cluster Computing*, Sep. 2010, pp. 1–8.

[18] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded mpi communication," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 120–129. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_20

[19] 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 *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 11–20.

[20] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded mpi applications using software offloading," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 30.

[21] "Lockless performance." [Online]. Available: https://locklessinc.com/

[22] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for mpi rma on many-core architectures," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 665–676.

[23] M. Prache, P. Carribault, and H. Jourdren, "Mpc-mpi: An mpi implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users Group Meeting (EuroPVM/MPI 2009)*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2009, vol. 5759, pp. 94–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_16

[24] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-process: techniques for practical address-space sharing," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 131–143.

[25] H. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced thread synchronization for multithreaded mpi implementations," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 314–324.

[26] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: https://doi.org/10.1109/99.660313

[27] M. Si, A. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "Mt-mpi: multithreaded mpi for many-core environments," in *Proceedings of the International Conference on Supercomputing*, 06 2014.

[28] R. Grant, A. Skjellum, and P. V Bangalore, "Lightweight threading with mpi using persistent communications semantics," in *Workshop on Exascale MPI 2015 held in conjunction with Supercomputing (SC15)*, 11 2015.

[29] S. Sridharan, J. Dinan, and D. D. Kalamkar, "Enabling efficient multithreaded mpi communication through a library-based implementation of mpi endpoints," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 487–498.

[30] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling communication concurrency through flexible mpi endpoints," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 390–405, 2014.