SAND2019-3335C

Finepoints: Partitioned Multithreaded MPI Communication *

Ryan E. Grant[1], Matthew G. F. Dosanjh[1], Michael J. Levenhagen[1], Ron Brightwell[1], and Anthony Skjellum[2]

[1] Sandia National Laboratories
(regrant,mdosanj,mjleven,rbbrigh)@sandia.gov
[2] University of Tennessee at Chattanooga
skjellum@utc.edu

Abstract. The MPI multithreading model has been historically difficult to optimize; the interface that it provides for threads was designed as a process-level interface. This model has led to implementations that treat function calls as critical regions and protect them with locks to avoid race conditions. We hypothesize that an interface designed specifically for threads can provide superior performance than current approaches and even outperform single-threaded MPI.
In this paper, we describe a design for partitioned communication in MPI that we call finepoints. First, we assess the existing communication models for MPI two-sided communication and then introduce finepoints as a hybrid of MPI models that has the best features of each existing MPI communciation model. In addition, "partitioned communication" created with finepoints leverages new network hardware features that cannot be exploited with current MPI point-to-point semantics, making this new approach both innovative and useful both now and in the future.
To demonstrate the validity of our hypothesis, we implement a finepoints library and show improvements against a state-of-the-art multithreaded optimized Open MPI implementation on a Cray XC40 with an Aries network. Our experiments demonstrate up to a 12× reduction in wait time for completion of send operations. This new model is shown working on a nuclear reactor physics neutron-transport proxy-application, providing up to 26.1% improvement in communication time and up to 4.8% improvement in runtime over the best performing MPI communication mode, single-threaded MPI.

1 Introduction

The Message Passing Interface (MPI) [15] has supported a threaded interface for user applications since 1997. Despite being supported for a long time, MPI multithreading

* Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

is not widely used today. There are several factors preventing MPI multithreading from widespread use. One factor is that MPI multithreading support remains poorly optimized in some common implementations and their commercial derivatives. Lack of consistency in performance is another major issue preventing widespread use, and this deficiency is understandable: making a highly performant multithreaded MPI library is complex and challenging. Additionally, it can be difficult for thread-based code to interact with MPI in an efficient manner. For these reasons, many hybrid codes today do not allow thread interaction with MPI, opting instead to coordinate to allow a single thread to perform MPI calls. For example, an MPI+OpenMP hybrid code might perform a computation using many threads but still use a single master thread to communicate using MPI.

The MPI threading model treats threads in much the same way as processes. Threads can perform all of the functions available in MPI; however, many codes do not require the full MPI multithreaded support that exists today. Alternative interfaces designed specifically for thread interaction with MPI could be designed to provide easy-to-use semantics and performance benefits over existing MPI interfaces. Performance improvements could be realized by leveraging thread behaviors and isolating the portion of the MPI API that needs to be thread-safe.

Contemporary MPI implementations typically use *pessimistic* serialization to enforce thread safety to MPI calls (*e.g.,* locks), allowing only a single thread to interact within certain MPI critical paths or data (*e.g.,* a given communicator) at a given time. While this restriction may be desirable for a general threading case where the behavior of threads is unknown, it can be problematic for threads that would otherwise not interfere with each other in their participation in a communication. For example, if multiple threads each write to a shared memory buffer using non-overlapping offsets, no interference would occur.

A 2018 survey [2] highlighted application developer concerns with MPI related to the US Department of Energy's Exascale Computing Project. All of the developers not currently using `thread_multiple` cited performance as the reason they were not using it, however, a large majority of those developers (86%) *want* to interact with MPI using multiple threads. Historically, some implementations used a single global lock on the MPI library. However, many implementations have recently moved to locking at a fine granularity. We have used an optimized fine-grain locking MPI for comparison to our proposed MPI threading interface enhancements, *finepoints*, in this work. Finepoints uses lightweight synchronization, requiring only a single atomic for synchronization (which is the minimum synchronization overhead achievable on modern hardware). It also works with emerging hardware to fully offload the threading synchronization overhead. Finepoints is the first solution to offer many of the optimizations that are available to one-sided communication methods in a two-sided model.

In this paper we will detail the design of our proposed solution, finepoints, a partitioned communication interface for MPI. This approach partitions buffers in MPI, allowing threads to contribute individual parts to a single communication operation. We will describe the interface and show the proposed MPI function calls and detail the reasoning behind the design as well as the benefits that the design provides. Next, we will present an implementation of finepoints and evaluate it on a Cray XC40 platform to assess its performance impact. Further, we will detail the changes that we made to two applications to adapt them to use finepoints and show the results of using the interface on a reactor neutron transport proxy application and a finite element code. Finally, we will summarize the results and discuss how these findings relate to the existing work in the area.

This paper makes the following contributions:

– A two-sided, optimizable MPI interface designed specifically for threading/tasking support;
– A design for how finepoints can be fully offloaded to future MPI message matching NIC hardware; and,
– The first finepoints-integrated proxy applications, a reactor neutron transport simulation and a finite element code.

2 Background

The model for multithreading in MPI is simple: threads in a process can access MPI however they wish, using the full interface that is used on a per-process basis in a non-threaded MPI program. There are no modifications to the interface for threads, and all threads share the MPI address (rank) of their parent process. MPI threading modes simply serve to dictate what level of thread-safety is provided by MPI—either `MPI_THREAD_MULTIPLE`, where all calls are thread safe, or funneled and serialized mode, in which the user is responsible for managing thread-safety with MPI. There is a fourth mode, single, in which threads do not exist. In this paper, we refer to threads as the mechanism by which tasks are run/completed.

MPI provides two main point-to-point interfaces, two-sided send/recv where each message sent matches a receive posted at the target. This results in per-message completion notifications and strong ordering guarantees for messages sent on a given communicator. MPI also provides a one-sided interface called Remote Memory Access (RMA). RMA uses a put/get semantic that can be conceptualized as a remote load/store model. RMA requires that an application explicitly handle synchronization of communication buffers. MPI RMA code typically requires major algorithm changes to use it effectively [14] and its expected use at exascale still remains low (<25% [2]). While work on the RMA model is promising in terms of performance [8, 12], the application level code changes required to existing code bases [14] compels exploration of a two-sided model.

The endpoints proposal, which is no longer under consideration by the MPI standardization committee, was an attempt to allow for increased utility for multiple threads (*cf,* [5]). Endpoints addressed threads by assigning a logical address (rank) to each thread or user defined group of threads, enabling per-thread addressability. Endpoints did not fundamentally change the communication model, it only added additional addressability to the existing multithreaded model in MPI. There is no public implementation of MPI endpoints and given its current status with the standards committee, it is unlikely that one will be released in the foreseeable future. Unfortunately, this means that performance comparisons are not possible. However, we can estimate the overheads of Endpoints matching, by emulating it's behavior with a traditional match list implementation that separates traffic by communicator.

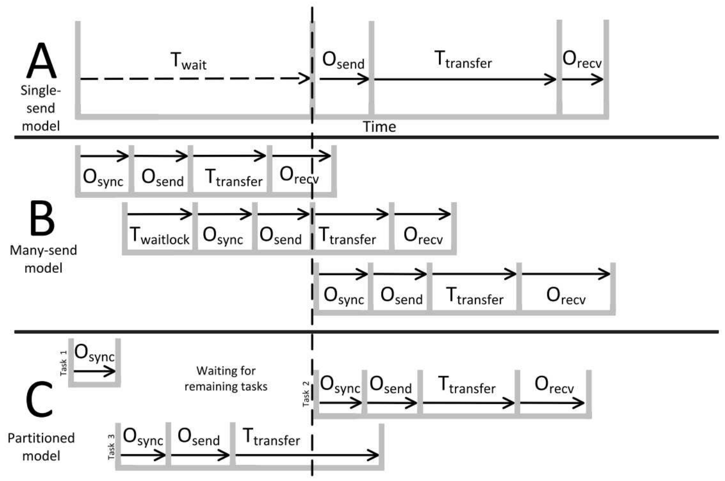


Fig. 1. MPI models for data transmission with multiple tasks/threads. Model A is the traditional single-send approach in an MPI+X applications. Model B is the current multithreaded send-per-thread model. Model C is our proposed partitioned model which leverages the completion model of A and transfer model of B.

2.1 MPI Multithreaded Communication Models

Figure 1 demonstrates the two main threading models used with MPI. Model A shows the single-send (single-threaded) MPI model, where only a single thread calls into MPI, regardless of how many tasks may be used in the non-communication regions of code. There is a time, T_{wait} that is required to wait for all task dependencies on the communication buffer to complete. No communication can happen even if some tasks are ready to send their data. Once all of the data is complete, there is some overhead for issuing the send commands in MPI, O_{send}, after which data is transferred over the network, $T_{transfer}$, and finally some overhead on the receiver-side for matching the message and marking the request as complete, O_{recv}.

Model B demonstrates the many-send model, in which each task sends data as it completes. This results in having no T_{wait} period as data can be sent when it becomes available (all dependancies for the data are complete and it is ready-to-send). Note the use of the time before the dotted vertical line that denotes the beginning of the transfer in model A. However, this also results in synchronization overhead, O_{sync} needed to ensure thread-safety in the MPI library, and $num_{threads}$ times the send and recv overheads. In addition, the message matching overhead for model B will likely be much larger than model A. Because matching is done serially, this message matching overhead will not only significantly increase all of the instances of O_{recv}, it can lead to cascading delays in subsequent instances of O_{recv}.

Therefore, we have two models, A, the single-send model, which cannot take advantage of T_{wait} and B, the many-send model, that has more overhead than model A. In current applications, the single-send model is the more popular approach as it has higher performance in many scenarios. However, there is broad interest in using a model similar to the many-send model as previously discussed [2]. To address this, we propose that a third model, C, that combines the best characteristics of both current models by taking advantage of T_{wait} like model B and offering the minimal receive-side overhead of model A. Model C allows tasks to notify the MPI library that their data is available to send, but allows MPI to make the decision on when to transfer data, ensuring that send overheads and network packet level efficiency can be controlled by MPI. In addition, MPI can manage multiple outstanding operations, allowing model C to take advantage of NIC-level parallelism, unlike model A. The number of receive-side notifications/completions is controllable between 1 and N_{tasks}, allowing for control over receive-side overhead. By setting the number of send-side data partitions to 1, model C can approximate model A; there is no inherent 1:1 relationship between tasks and partitions.

3 Hybrid-model Design Requirements

There are three requirements for the design of partitioned communication (model C in figure 1). First, the design must allow communication to occur when tasks complete, rather than synchronizing on a thread barrier or join before communicating. This addresses the weakness in model A, that there is wasted time due to a monolithic task dependency of all tasks completing before communication. This allows partitioned communication to emulate the strengths of model B, a lack of this monolithic task dependency. We will refer to the productive use of the time that would be spent waiting in model A as early-bird communication.

Second, the design must address the weakness in model B by minimizing or hiding the overheads of thread-safety, send operations, and receive-side overhead. Low receive-side overhead is achieved by reducing number of matching operations and request completions/notifications. There can be an advantage to having multiple completions, namely that task dependencies on a subset of the data can progress when the required remote data becomes available. However, this needs to be balanced with the increased overhead caused by multiple completions. For the purposes of this paper, we choose to minimize receive-side overhead by using a single match/completion operation. This will keep the design in line with the desirable low overhead in model A and avoid matching issues. The last requirement is that the MPI implementation should be free to send data whenever it is most efficient to do so. This allows data to be sent with good wire efficiency (header to payload ratio), a benefit of model A and weakness of model B. By controlling the size of the data sent and allowing for aggregation inside the MPI library we can control the wire efficiency of data transfers for model C. This is something that both models A and B are not able to take advantage of; each send operation is sent as a distinct message in modern MPI libraries.

To allow for easy use/adoption of the partition communication models, there are three main objectives that the design has to meet. The first objective was to align our design with current practice. Existing concurrency models like OpenMP allow for tasks and threads to work cooperatively on a shared buffer—in the most simple sense, like a SIMD model, where multiple data items in a buffer are acted on simultaneously. Threading can be more complicated but the basic concepts behind this approach can be leveraged to better match the thread usage model for MPI. This widely adopted tasking/threading model provides a single buffer that multiple actors (thread, tasks, etc.) can operate on. By matching the semantics of the MPI calls to the semantics of common multithreading models, mismatch can be avoided at the interface level and programmers can easily translate existing threading code to communication.

The second objective was to align our design with legacy practice. Legacy applications have leveraged an MPI+X model with distinct communication and computation phases in a bulk synchronous model. Multi-million-line MPI codes require great effort to modify and revalidate. Therefore, minimal code change at the MPI level is desirable in terms of time and cost for updating and revalidating legacy code bases. This is useful to ensure the widest adoption possible.

Finally, the third objective was to aline our design with anticipated future practices. Proposed models for future applications include task based threading, over decomposition, adaptive workflows, in-situ analytics, etc. These models increase the complexity of requirements for the communication layer of an HPC system. Changing the communication model of legacy and modern codes is a significant undertaking, often performed by non-experts of MPI. To reduce the burden on application developers the interface needs to be adaptable to future hardware and programming models with little to no impact on interface provided by an implementation.

3.1 Finepoints: Partitioned Communication

Finepoints is a MPI interface for partitioned communication designed to match the characteristics of the hybrid communication model C in figure 1. Partitioned communication is a new concept in MPI. With partitioned communication we propose breaking the monolithic nature of the single send model by allowing tasks to express data availability to the MPI library by reporting parts of an operation as ready. This allows data to be moved as a portion of a larger operation. The larger operation will have the same receiver-side overhead as the single-send model. We will present the design of finepoints. We target the two-sided communication model in MPI for our design as it is by far the most popular communication method in MPI applications. However, we will discuss the use of MPI's one-sided model where it is a possible option for meeting our requirements.

The first requirement was to use the time that would otherwise be wasted in the case where many threads or tasks need to synchronize before sending any data. Partitioning a send operation accomplishes this goal by notifying MPI as portions of the data buffer become available. This gives MPI the opportunity to send data if doing so is desirable. There are situations in which delaying sending the data is the correct decision, for example, when the available data is too small in size. Another situation in which a system would want to delay data transfer is when the data in the buffer is too fragmented to be sent efficiently. Some networking hardware can efficiently handle strided data or IOvecs, which can describe fragments of data to be collected from memory and sent (gathered) and distributed back to target memory on the receive side (scattered). These features make it desirable to leverage hardware that supports such gather/scatter operations, such as Remote Direct Memory Access (RDMA) as it maps well to these capabilities.

To address the low message processing overhead requirement we must keep the number of messages that MPI must match small, similar to a single-threaded MPI process. One way to get around the matching requirement is the use one-sided communication, which does not provide matching. The drawback to this approach is that the method of completing a given communication with one-sided code is much different than that of two-sided send/recv. Send/recv provides clear message arrival notification as a completed request. One sided communication requires synchronizing a memory window between the sending and receiving nodes. This change in semantics can significantly impact application code, in some cases requiring changing the underlying algorithms to better fit the communication semantics. To avoid these drawbacks a design must leverage two-sided send/recv semantics and must produce as few messages that need to be matched as possible. Along these lines, partitioned communication matches these requirements. It reduces the number of messages that must be processed (matched and notified of completion) but also allows for fine-grained notification of parts of a buffer becoming available to send to MPI. This allows MPI to optimize how the data is sent for any given network architecture, but still allows for the well known send/recv completion semantics. This addresses the issues that can afflict the many-send situation (model B) of each thread/task sending its own messages. This also addresses the third requirement for our design, controllable wire efficiency.

Now that all of the desirable traits and potential designs have been discussed, we can outline a basic design and API for finepoints. Finepoints will use a partitioned send, allowing threads/tasks to notify MPI when portions of a larger shared buffer become available to send. We will only allow a limited number of completions on the receive side to minimize message processing overhead. We can allow some receive side partitioning/notification, but we must be careful not to create too high of an overhead from matching/request completion. Next, we will require some sort of buffer negotiation, as the buffer can arrive in chunks instead of all at once. This can be done using a persistent operation to reduce setup overhead or an on-the-fly one-time-use buffer negotiation. We will provide both interfaces for send operations.

3.2 Partitioned MPI Communication Interface

Partitioned communication in MPI as a concept can be applied to almost all of MPI's existing communication calls, both point-to-point and collectives. For the purposes of our paper, we will only cover point-to-point communication. We present our proposed additions to the MPI-3 or MPI-4 standard in C; for brevity, we omit the Fortran versions of the calls.

The persistent communication approach requires that certain information about the partitioned operation be expressed to MPI prior to writing to any buffers. First, the operation must be initialized; that is, the required information to set up the buffers and synchronization methods must be provided.We propose a `MPIX_Partitioned_send_init` function call defined below. This function can be used to initialize the partitioned send, which is similar to a persistent operation setup, but introduces the concept of message partitioning.

```
int MPIX_Partitioned_send_init(
 void *buf, int count, MPI_Datatype data_type, int to_rank, int to_tag,
 int num_partitions, MPI_Info info, MPI_Comm comm, MPI_Request *request);
```

Similarly, a recv version of this call must be created to allow for the sender and receiver sides to agree on a buffer for the partial messages (which may be the application buffer on the target side).

```
int MPIX_Partitioned_recv_init(
  void *buf, int count, MPI_Datatype data_type, int from_rank, int from_tag,
  MPI_Info info, MPI_Comm comm, MPI_Request *request);
```

These initialization functions match via tags, sender/receiver rank, and communicator at initialization to form a two-process persistent operation (channel). While wildcard sources/tags may be used for `from_rank` and `from_tag` in the `MPIX_Partitioned_recv_init` call, it is up to the programmer to make sure that there is logical consistency between the sender and receiver that connect during this process. Unlike normal point-to-point persistent send/recv, these operations *may* communicate. To reduce complexity in initialization, these calls should be non-blocking. The output of this function is a `request` that can be used immediately in a `MPIX_Pready` call.

When a request is active on the send side, buffer partition elements may be added with the following API:

```
int MPIX_Pready(
   void* buf, int count, MPI_Datatype in_datatype, int offset_index,
   MPI_Request *request);
```

For non-persistent communication, a normal recv operation is used at the target and a partitioned send request can be started with the following API:

```
int MPIX_Ipsend(
   void *buf, int count, MPI_Datatype data_type, int to_rank, int to_tag,
   int num_partitions, MPI_Info info, MPI_Comm comm, MPI_Request *request);
```

When the request is in progress, it waits for `num_partitions` `MPIX_Pready` calls. When the number of buffer partitions added equals the `num_partitions` argument given at initialization, no more partitions may be added prior to a completion operation (`MPI_Wait`). The total size of the buffer is the `count` value times the size of the datatype, in bytes, given at initialization.

When using the persistent version of Pready/wait calls (e.g. `MPIX_Pready`, not `MPIX_Ipsend`), `MPIX_Pready` calls for subsequent rounds of communication can only be made after a successful `MPI_Wait` or `MPI_Test` call on that request. The buffer should not be altered until it is confirmed that the send operation is complete. This motivates the use of multiple send buffers as it allows tasks to continue to execute and overlap their computation with communication. With even a limited number of buffers, an application can avoid waiting long periods for communication completion. When combined with the non-synchronous nature of `MPIX_Pready` calls, this will enable applications to spend essentially no time in synchronization barriers for coordinating send operations or waiting on their completion. Tasks will still be required to wait for incoming data if it is not available.

It should be noted that extending the partitioned communication interface to support partial receives is trivial. However, we leave such extensions to future work as the optimization space for receive side partitioning is large and warrants its own full-scale investigation. We instead concentrate on the performance benefits of the send-side partitioning in this paper as an introduction to the general partitioned communication concept.

`MPIX_Ipsend` calls are expected to return immediately. `MPIX_Pready` calls are subsequently used to indicate partition readiness. `MPIX_Ipsend` calls are similar to existing persistent communications interfaces, except that there are no requirements for communication-initiating calls prior to calling `MPIX_Ipsend`, the setup happens when `MPIX_Ipsend` is called. It is required that the receiver-side post a non-blocking receive that will match the `MPIX_Ipsend` call. If no match is found `MPIX_Ipsend` will return an error code indicating that the operation is not ready and the user should try again. This error reporting is not fatal, following the precedent set by file I/O in MPI.

There are no blocking versions of `MPIX_Pready`; it is always a non-blocking call. The offset_index is an integer that specifies what internal index the datatypes have in the buffer. For example, for a simple contiguous buffer case, the first element would have index 0. Complex datatypes are supported, and the index associated with those datatypes should be interpreted as their logical placement in the buffer compared to the other expected contributions of datatypes. There is no demand that buffer contributions be non-overlapping in memory; however, we will not define the behavior for overlapping buffer additions here.

`MPIX_Pready` and `MPIX_Ipsend` calls can be made thread-safe independently of the other thread concurrency requirements of the MPI library because of the threading isolation that these functions provide. Since partitioned communication does not rely on other parts of the MPI library that have potential thread safety issues, we propose that the finepoints calls use a new threading mode, `MPI_THREAD_PARTITIONED`, which allows high-performance lock-free MPI calls for the majority of the library by isolating a thread-specific interface for handling concurrency. Partitioned communication need not share significant internal MPI data structures, and the only concurrency required is an atomic fetch and increment to determine if the partitioned operation has reached its `num_partitions` threshold. Finepoints leverages the knowledge of thread/task interaction at the application and runtime levels to allow for overhead much lower than an MPI implementation with a traditional send/recv type interface could reasonably be expected to provide. An example of finepoints code is provided in Algorithm 2 which demonstrates a simple microbenchmark that we will use in Section 4.

3.3 Hardware Support for Partitioned Send

We can design full-featured hardware support for partitioned send operations from basic operations of some MPI message matching NICs without the need for new hardware. An example of a networking solution that can support partitioned sends today is Bull's BXI interconnect [3]. Bull's BXI network uses the Portals 4 networking API [1], which supports triggered operations. Triggered operations use a hardware counter on networking devices to accumulate counts of certain events that can be associated with them. Consequently, on the receive side, a Portals-compatible NIC can keep a count of the number of expected contributions to a buffer and deliver immediate notification of completion to the target. The send-side MPI library can leverage triggered operations as well, by staging multiple requested send operations with the different counts on which they are triggered. Using the `PtlCTInc` function in the Portals 4 API, MPI can perform the bookkeeping required for subsections of the partitioned buffer on the NIC hardware. Once a given sub-partition of the overall buffer has been placed, the hardware automatically triggers the send to occur. This automatic send allows for increased network efficiency while offloading a large portion of the work that would otherwise have to occur in software (counting incoming segments and determining when a request is complete). This concept is the same one behind the mechanism of offloading MPI collectives with Portals-compatible

```
Data: buf: application buffer, msg_size: message size
Result: transmission of data buffer to remote node using finepoints
if sender then
    MPIX_Partitioned_send_init(&buf, msg_size, MPI_INT,
      receiver_rank, my_tag, num_partitions, info, MPI_COMM_WORLD, &request);
else
    MPIX_Partitioned_recv_init(&buf, msg_size,
      MPI_INT, sender_rank, my_tag, info, MPI_COMM_WORLD, &request);
end
for iteration = 0; iteration++; iteration == 1000 do
    #pragma omp parallel {
    /* We are
      only using one buffer, so need to wait on send completion before re-using it*/
    if !first_loop then
        MPI_Wait(&request);
    end
    compute_loop(compute_time);
    first_loop = false;
    msg_chunk_size = msg_size/omp_get_num_threads();
    if sender && (delay > 0) && my_thread_id == 0 then
        wait(delay);
    end
    if sender then
        MPIX_Pready(&buf,
          msg_chunk_size, MPI_INT, msg_chunk_size*omp_thread_num(), &request);
        /*When all partitions are ready, remote Recv will match and proceed*/
    else
        if my_thread_id == 0 then
            MPI_Recv(&buf, msg_size,
              MPI_INT, sender_rank, my_tag, MPI_COMM_WORLD, &status);
        end
    end
    }
end
comm_time = comm_time / 1000;
```

Fig. 2. Pseudo code for Finepoints Microbenchmark

hardware [18]. Therefore, finepoints allows the utilization of network offloading capabilities that are currently being applied for MPI collective offloads to be used for point-to-point communication as well. Networking offload is desirable at exascale [7], and therefore we expect that such offloading capabilities will be widely available in the near future.

4 Experimental Results

In this section, we detail our experimental platform and assess the performance of finepoints via extensive microbenchmark experimentation and the evaluation of two proxy applications, a finite elements code, MiniFE [11], and a nuclear reactor physics code, SimpleMOC [10], both part of the application set for the Exascale Computing Project run by the US Department of Energy (DOE).

To assess finepoints, we have implemented a library on top of MPI that allows partitioned communication (finepoints) to be layered on top of existing MPI calls, particularly the MPI RMA interface. These results demonstrate the performance of a non-hardware implementation of finepoints.

4.1 Experimental Platform

Our experiments were run on a Cray XC40 system. XC40s have two different node types: a dual-socket node with Haswell E5-2698v3 CPUs and 128GB RAM, and a single-socket node with a Knights Landing (KNL) Xeon Phi 7250 many-core CPU with 96GB RAM and 16GB MCDRAM. This model has 68 cores each with support for 4-way SMT. For this reason, microbenchmarks use 64 threads while the application study extends to 256. For purposes of the evaluation, the number of partitions in the subsequent experiments is set to be the number of threads. While there are other configurations available, this is the primary use case we expect to see in finepoints applications. We used the KNL nodes exclusively, as many-core architectures let us explore large amounts of thread concurrency. These results all utilize the same Cray Aries Interconnect with a theoretical maximum bandwidth of 10.2 GiB/s. Open MPI 3.0 is used as the thread-optimized MPI library to interface with our finepoints library. Open MPI's message matching solution combined with different tags for each multi-send message mean that message matching overhead is minimal, approximating the performance of traditional as well as persistent multi-send (`MPI_THREAD_MULTIPLE` overhead is still significant).

4.2 Microbenchmarks

Microbenchmark Setup In order to evaluate the fine-grain behavior of finepoints, we have created microbenchmarks that assess performance during OpenMP parallel loop execution for data exchanges. Our benchmarks allow for the independent variables of message size, number of threads, compute time per loop, and compute time variation. The compute time variation variable represents typical application performance variation that results from imbalances in the amount of work to be done per process, due to OS noise and process placement on large systems. This variation allows us to explore finepoints ability to leverage the idle thread time caused by this noise, as finepoints decouples individual thread completion from communication dependency. This noise represents the T_{wait} time in the communication models comparison from Figure 1. To implement this compute time variation, we delay a victim thread by the required noise amount. After this noise is injected, the microbenchmarks communicate using the selected communication model. For single send, the threads synchronize after which a single large message is sent. For multi send and finepoints, each thread sends an equal portion of the message using an MPI_Send or a Pready call. For the single-thread case, this delays the thread synchronization and thus the only send call; for multithread, it is the last send call to occur; and for finepoints, it is the `MPI_Pready` call time for the completing call.

The dependent variable from this microbenchmark is perceived bandwidth. Perceived bandwidth is bandwidth required for an single threaded `MPI_Send` to complete in an equivalent time. This is measured by instrumenting the time of the final thread joining the communication region, where the `MPI_Send` would have been called in the single threaded model, to the completion of all communication for the iteration. From our communication models in Figure 1, this corresponds to calculating the bandwidth from the all-tasks complete point in time (the dotted vertical line). We do this as it provides a baseline for performance centered on the single-send model, the dominant MPI communication model. For a traditional `MPI_Send`, the perceived bandwidth is the whole transfer time of the message. For finepoints and traditional multi-send, the perceived bandwidth is the bandwidth that the single-send model *would need* in order to complete the communication after the all-tasks complete point to match the wait time of finepoints or multi-send. While these numbers could be presented as time spent waiting after the all-tasks complete point, perceived bandwidth allows a comparison to a well-known metric that can be scaled with trivial effort for future generations of hardware.

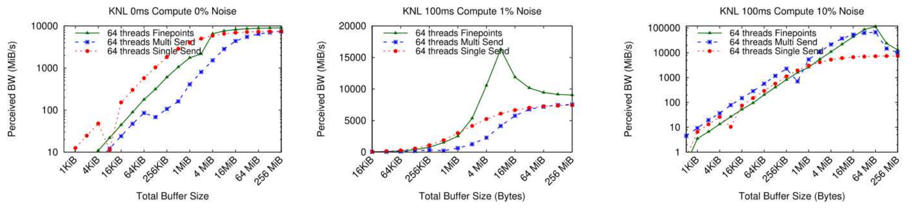
Perceived bandwidth can be significantly higher than the actual bandwidth on a system. For example, if 100 workers all need to send one piece of data to a neighbor node and 95 workers complete but 5 workers take significantly longer, 95% of the data transfer can occur with finepoints and multi-send before the last workers reach their communication calls. Thus, the observed communication call could take 95% less time than a call that used the single-thread method, which requires all data to be collected before sending the first byte. Each microbenchmark experiment was run for 50 iterations and the data in this section represents an average over those iterations.

Microbenchmark Results Our first experiments vary only message size and thread count. Figure 3a shows the performance of finepoints versus send/recv in MPI for both single-thread and traditional multi-send MPI. These results exclude any compute time in the communication loops or any noise.

Figure 3a shows typical trends that are expected by MPI experts. The single-send model's MPI operation is superior to the multi-send model's MPI version and, as lock contention increases with the number of threads, the performance gap between single-send and multi-send models grows, even for our multithreading-optimized MPI implementation. The single-send model clearly outperforms multi-send and finepoints at small message sizes when ignoring the drop in performance that occurs during the eager-rendezvous protocol switch. Breaking up a small message into 16, 32, or 64 parts operates the network in the lower part of a typical bandwidth curve, where packet overheads dominate costs. For small message sizes, performing a single send operation is still preferable, while the benefits of the finepoints approach are clear with larger message sizes. It should be noted that finepoints can accommodate aggregation of smaller messages such that the performance of singlethreaded MPI can be approximated. The results for these benchmarks do not use aggregation.

The performance of small message transfers may appear to be problematic as many MPI applications use small to medium sized messages frequently. However, with the shift towards fewer MPI processes and more threads per process, the overall amount of data needed to be transmitted by a single process will grow. A 64 process MPI-everywhere solution will have to send 64$\times$ more data when it is run in a one-process, 64 thread configuration, leading to larger message sizes for multithreaded codes. This will push many application into an area where finepoints performs well. Based on previous work exploring message sizes used by applications of interest [6], we find that message sizes in the 8KiB-16KiB range are important and many applications send messages of 1 MiB or more, resulting in the vast majority of network usage for codes. Therefore, when we move to a multithreaded code, we expect messages to be $N_{threads}$ times larger in size, and this is well within the message size range where finepoints is the clear winner. Notably, at a 1 MiB transfer size (total, not per thread), finepoints outperforms the single-send model performance for all thread counts below 64 (results not shown for space). At 64 threads, a 2 MiB transfer is required to outperform the single-send model performance. This performance gap is significant for large transfers when all of the bandwidth curves flatten out. The difference in performance, from 7,200 MiB per second to 9,000 MiB per second on finepoints, represents a significant 25% increase in throughput.

This difference results from several factors. First, finepoints can easily leverage hardware RDMA data transfers, allowing for high-performance messaging. Second, the MPI library *expects* the finepoints transfer to occur; with traditional MPI send/recv, the library must react to the transfer with no advance setup. What is most promising here is that the observed improvement comprises a worst-case outcome for finepoints, since there is no time variance in the compute or noise in the system that allows finepoints to take advantage of available bandwidth in the network while laggard threads finish their compute tasks.



(a) no noise, no comp. (b) 1% noise, 100ms comp. (note: linear scale) (c) 10% noise, 100ms comp.

Fig. 3. Partitioned Communication with varying noise and compute load

Figure 3b shows finepoints working with a 100ms compute loop with 1% noise. This reflects real codes on production machines better than no-noise situations do, as 2-4% noise has been common on systems for many years [16]. Both multi-send and finepoints can show bandwidths greater than the available bandwidth from the NIC using this approach because they have the opportunity to send portions of the overall transfer before the final process/thread reaches the communication call. With 64 threads, we see the drawback of MPI's `THREAD_MULTIPLE` mode. Lock contention is high with large thread counts, impacting the performance of the multi-send approach. While multi-send quickly degrades in performance and even underperforms the single-send model, finepoints gains performance from having many threads. Eventually with large enough message sizes, we see finepoints and multi-send converging back to native wire rates, this happens when there is so much data that the early-bird overlapping cannot preemptively send a large enough portion of the data to see major performance gains. It should be noted that even in these cases, finepoints performance is no worse than the singlethreaded case. Finepoints starts to see a small drop in performance at the 64-thread level illustrated in our Figure 3b, but still significantly outperforms the best competitor at message sizes of greater than 1 MiB total. To put these bandwidth numbers in context, finepoints with a 8MiB message will spend only 0.5% of its time waiting for communication with a 100ms compute loop, compared to 1.9% for the single-send model and 1.3% for multithreaded MPI. This result highlights one of the key performance benefits of finepoints, namely that multiple threads can initiate data movement, exploiting

T_{wait} from our communication models without the locking overhead $T_{waitlock}$ and much smaller O_{sync} compared to the multi-send mode.

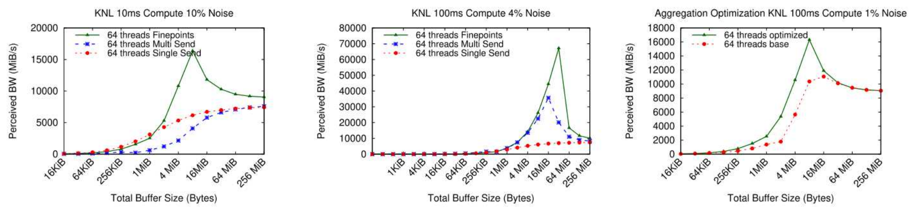
Shorter compute times can impact the amount of overlap that finepoints can exploit, with compute times of 10ms demonstrating up to a 3× improvement in performance versus the single-send model as shown in Figure 4a. Figure 3c shows early-bird communication mostly completing before the final thread arrives at the partitioned communication call. With the chance to send this data in advance, the perceived bandwidth when the final thread reaches the partitioned communication call is 12× greater than a single threaded approach for 64 threads at message sizes of 64 MiB. At a 64 MiB message size finepoints only spends 0.5% of time waiting for communication while the single-send model spends 8.8% and multi-send spends 0.95%. For the 4% noise case, which we expect to be typical of future systems we can observe in Figure 4b that with reasonable compute times, finepoints can beat single threaded MPI by up to than 9.5× and multi-send by 3.3× in the best case. Thus, early-bird communication can help alleviate a major cost of bulk-synchronous parallelism (BSP). It achieves this result by reducing the time penalty for poor synchronization, reducing the delay after all threads have reached the synchronization point as much as possible.

To ensure that the results observed for finepoints are not a result of the lightweight cores used for the experiments, we have also conducted similar testing on the Haswell partition of our system. These results are omitted for space, but the general trends hold on a Haswell system as well: finepoints outperforms both multi-send and the single-send model for message sizes larger than 1MB across a spectrum of no-noise to noisy execution. For example, at 32 threads (one thread per core), finepoints beats single by 34% and multi-send by 99% at 64MiB message sizes with no aggregation, with the latest version of Cray MPICH.

4.3 Message Aggregation Optimizations

The results presented thus far have sent messages as soon as any data was added to the partitioned buffer; however, finepoints can also optimize the transfers out of the partitioned buffer by aggregating traffic to the target node. We have implemented an aggregation scheme that allows the user to specify an aggregation threshold for their network. Our aggregation scheme attempts to combine send operations that occur close together in time that are in contiguous memory, up to the aggregation threshold size. Timeouts will cause data to move regardless of aggregation if operations are sufficient spread out in time.

Aggregation is most effective when there is a large number of threads, which corresponds to more numerous and smaller individual data transfers. Figure 4c shows the benefit of this aggregation versus a baseline finepoints without aggregation for 64 threads with a 512 KiB aggregation size. We observe that the aggregation optimization can have large impact on the overall performance of finepoints at high thread counts. For the 100ms compute loop results shown, the maximum gain is 199.5% at 2MiB, and the optimized version is always better than the baseline finepoints case.



(a) 10% noise, 10ms comp. (b) 4% noise, 100ms comp. (c) 1% noise with aggregation, 100ms comp.

Fig. 4. Three experiments exploring the effects of finepoints in situations of short iteration times, realistic noise, and with aggregation

4.4 Application Proxies

To demonstrate finepoints with an application, we chose two application proxies to test the impact of finepoints. MiniFE is a proxy application from the Mantevo suite [11] that uses conjugate gradient solver on a finite elements problem. The main communication pattern is a fully packed halo exchange, optimized for the single-send model. MiniFE is essentially a worst case for finepoints as it is optimized to send small messages and is highly tuned for the single-send model. To leverage finepoints in this code, we modified the application in the most direct manner possible, where each thread sends a subset of the overall buffer. This results in a significantly larger number of messages being sent by each peer compared to original serialized code. To provide a comparison to current multithreaded paradigms, we have included a multi-send version as well that decomposes messages in the same manner of the finepoints version.

Figure 5 shows the results of MiniFE run with 16 nodes (1 process per node) and a 330^3 problem size per process with no injected variation in the communication phase. Each data point represents the average of three runs. In this graph we show communication time on the primary y axis and cg-solve time on the secondary y axis. The general trends in this data show that finepoints performs better than multi-send but worse than the original serialized code. Because of a bug we encountered in Open MPI, multi-send runs leveraging more than 32 threads failed to complete. At this scale, the communication in finepoints is a factor of 2 better than the multi-send baseline. In follow-on experiments leveraging Cray's MPI, we found that multi-send spent 61% of the CG solve time in communication at 256 threads. In contrast, finepoints spends just 11% of its time communicating with 256 threads. As the message decomposition strategy results in a larger number of smaller size messages, it is unsurprising that finepoints spends more time communicating than the single threaded case.

While finepoints does spend a larger percentage of its solve time doing communication than the single-send model (11% vs 2% at 256 threads), there are a two promising things to note. First, this is a worst-case application for finepoints, MiniFE has a highly optimized communication pattern, sending as little data as possible in it's halo exchange. Given this, application developers can use finepoints to leverage multithreaded communication patterns with a small to negligible impact in application performance. Second, the application use of finepoints is unoptimized and cannot take advantage of variations in compute time. In the current implementation, MiniFE exits the parallel region and then starts a new parallel region for communication.

This means the time still includes the $T_W ait$ portion model A and no early bird communication can occur. This is an example of how an unoptimized finepoints code can perform significantly better than current multithreaded communication models. Leveraging knowledge of the data dependencies and thread behavior, application developers could integrate these communication calls into their compute threads and enable more early-bird communication reducing runtime. This case highlights the fact that, while there are cases where applications will need to be optimized to see benefit from finepoints, the overhead from finepoints is low enough that the impacts to a "worst case" application, with small halo exchanges and little noise are minimal.

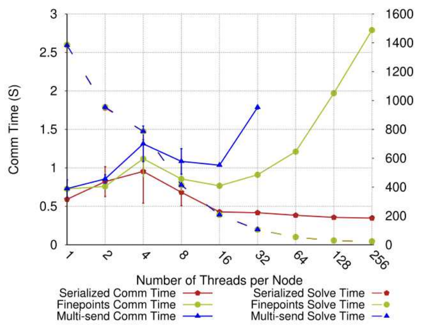


Fig. 5. Finepoints impact on MiniFE performance

SimpleMOC, a reactor physics proxy application from MIT and Argonne National Laboratory, is part of the DOE's ECP application set. We chose this code because it simulates a real problem (albeit not easily adaptable to other problems, which is why it is a proxy application). Also, we can convert its existing communication pattern to partitioned communication without the need to re-factor its data packing routines, making it possible to ensure continued program validity when modified by non–reactor physics experts. This also means that the volume of code changes for such applications are less than 100 lines of code. The changes that need to be made to adopt finepoints should also be easy to implement for application developers who are domain area specialists. We expect that all codes that utilize halo-type message exchanges can benefit from our approach.

SimpleMOC demonstrates the method-of-characteristics technique to solve partial differential equations with a specific application to 3D neutron transport in a light-water nuclear reactor at full scale. SimpleMOC requires multiples of four for MPI process count and communicates only in groups of four; therefore, we have used four KNL nodes for our tests allowing us to use 1024 cores total, with 256 threads per node. Using a larger number of nodes will not provide more insight. Due to the communication pattern used by the code scaling up the number of nodes will simply duplicate the communication pattern. Therefore out results demonstrate the improvement in a given "cell" of the problem breakdown that will be applicable to much larger problem sizes.

For experimental purposes, we have added barriers to the communication portion of the code and included code that allows artificial injection of noise in proportion to the compute loop time. This modification is useful, as we have observed 2%-5%

noise impacts in regular runs of the SimpleMOC communication section. By using barriers and artificial injection, we can tightly control the occurrence of this noise in the communication region, making experimentation and understanding easier. We have also run tests without noise controls to demonstrate production performance expectations. In order to let communication begin as soon as possible, we have eliminated the synchronization barrier entirely, allowing the first thread to complete to begin communication (and report the solver completion time). This is the best case for computation as the time reported for our unconstrained finepoints application is the time the first thread gets to the communication point. However, this makes the communication time longer as the communication cannot complete until all compute threads have completed, the performance variation is observed in the communication time and is similar to our large variance case, where overlap occurs, but its benefits are degraded due to the time spent waiting for laggard threads in which there is no communication to perform. The net effect of allowing communication to begin as soon as possible is that the performance variation between threads is observed in the communication phase. Overall this is similar to the time required when 2%-3% variation is injected in the communication phase, but the speedups in overall time shows up in the solve completion due to our eager recording of the solve time and our early start time for communication. All data for SimpleMOC is an average of 10 runs, and we use our aggregation-optimized MPI library with Open MPI 3.0. SimpleMOC was configured according to the recommended small problem size and then scaled in terms of azimuth values (32) and height (1200) to expand the problem size to the MCDRAM capacity on the KNL. The KNLs were run in quad mode with 100% of MCDRAM operating in cache mode.

The results of this testing are shown in Figure 6. We can observe that finepoints provides a significant improvement in application performance in both communication time and application total runtime compared to the single-send model optimized version of SimpleMOC. SimpleMOC supports varying numbers of threads in its main compute loop, and we present results using a sweep from 64 threads to 256 threads on each of the 4 KNL nodes.

For the noise controlled runs, communication time improvement sees a low point of 18.5% at 256 compute threads and 10% noise, and it peaks at 26.1% at 64 compute threads and 2% noise. Application runtime improvement ranges from 2.6% for 64 compute threads and 0% noise to 4.8% with 128 compute threads and 2% noise. Overall, both runtime and communication time improvements are relatively similar over the ranges of artificial noise injection because of the nature of the communication that occurs: the communication is small enough in size (approx. 130 MiB total) that even small noise percentages allow good early-bird communication.

5 Related Work

There have been past attempts to integrate threading within MPI, such as FG-MPI [13]. FG-MPI promoted threads to being the equivalent of MPI processes, which while it allowed many concurrent threads, creates a large amount of state for each thread/process. Other efforts have included work on providing benchmarks for testing and profiling MPI RMA multithreaded behavior [8]. The general concept of composing RDMA messages into a large transaction has been explored for

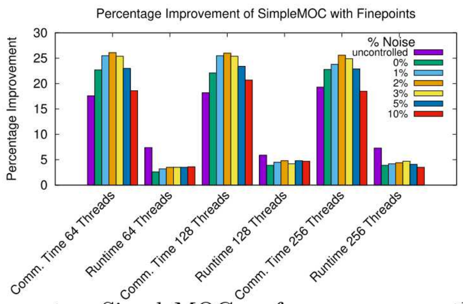


Fig. 6. Finepoints impact on SimpleMOC performance versus the single-send model

application in unreliable datagram networks at the hardware level [9, 17]. Similar benchmarks have also been developed for other one-sided communication APIs like OpenSHMEM [19]. Lastly, commercial MPI's such as MPI/Pro, which were designed for internal concurrency and the option of blocking completion notification (to avoid polling), are no longer widely available [4].

Message aggregation is a well known method for networks, having been explored for one-sided communication methods [12], these methods are also common with TCP/Ethernet networking.

The MPI forum had a proposal before it to enhance support for threads through endpoints [5], in which each thread can be assigned a unique rank in an endpoint communicator. However, endpoints never attempted to address the underlying communication model, only add the ability to address messages to specific threads. This work differs from previous efforts by the requirements it places on applications and the corresponding decrease both in resources needed by MPI and in synchronization overhead achieved.

6 Conclusions and Future Work

In this work, we introduced finepoints, a partitioned buffer communication two-sided approach for MPI. Partitioned sends allow data to be transmitted as completed by the application, or else be aggregated by MPI. We discussed the existing concurrency models in MPI and illustrated how desirable features of each model can be combined, resulting in our design of finepoints. Providing threading support with partitioned operations allows for ultra-low overhead thread safety that beats a current highly optimized threading-optimized MPI implementation and fits the existing application code methodologies. A prototype implementation that incorporates early-bird communication provides up to 4.8% improvement in runtime and 26.1% improvement in communications for a reactor physics neutron transport code. Furthermore, this performance improvement did not require major application reformulation, unlike MPI 1-sided communication (RMA).

Partitioned send is only a part of the overall finepoints concept. It is possible to extend finepoints to receive-partitioning. Receive-side partitioning solves a different problem than send-side partitioning; that is, it is an independent concept. The flexibility of receive-side partitioning must be juxtaposed against the increased cost of notification (with reduction of maximum message rate), making it a subject for future study.

[1] B. W. Barrett, R. Brightwell, et al. The Portals 4.1 networking programming interface. Tech. Rep. SAND2017-3825, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2017.

[2] D. E. Bernholdt, S. Boehm, et al. A survey of MPI usage in the U.S. Exascale Computing Project. *Concurrency and Computation: Practice and Experience*, 2018.

[3] S. Derradji, T. Palfer-Sollier, et al. The BXI interconnect architecture. In *Proceedings of the 23rd Annual Symposium on High Performance Interconnects*, HOTI '15. IEEE, 2015.

[4] R. Dimitrov and A. Skjellum. Software architecture and performance comparison of MPI/Pro and MPICH. In P. M. A. Sloot, D. Abramson, et al., eds., *Int. Conf. Computational Science (ICCS)*, pp. 307–315. 2003. ISBN 3-540-40196-2.

[5] J. Dinan, R. E. Grant, et al. Enabling communication concurrency through flexible MPI endpoints. *Int. Jour. of High Performance Computing Applications*, 28(4):390–405, 2014.

[6] D. W. Doerfler, M. Rajan, et al. A comparison of the performance characteristics of capability and capacity class hpc systems. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2011.

[7] M. G. F. Dosanjh, R. E. Grant, et al. Re-evaluating network onload vs. offload for the many-core era. In *IEEE Intl. Conf. on Cluster Computing (CLUSTER)*, pp. 342–350. IEEE, 2015.

[8] M. G. F. Dosanjh, T. Groves, et al. RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 550–559. IEEE, 2016.

[9] R. E. Grant, M. J. Rashti, et al. RDMA capable iWARP over datagrams. In *IEEE Int. Parallel & Distributed Processing Symp. (IPDPS)*, pp. 628–639. IEEE, 2011.

[10] G. Gunow, J. R. Tramm, et al. SimpleMOC - a performance abstraction for 3D MOC. In *ANS MC2015*. American Nuclear Society, American Nuclear Society, 2015.

[11] M. A. Heroux, D. W. Doerfler, et al. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[12] N. Hjelm, M. G. F. Dosanjh, et al. Improving MPI multi-threaded RMA communication performance. In *Proc. of the Int. Conf. on Parallel Processing*, pp. 1–10. 2018.

[13] H. Kamal and A. Wagner. An integrated fine-grain runtime system for MPI. *Computing*, 96(4):293–309, 2014. ISSN 0010-485X.

[14] P. Mendygral, N. Radcliffe, et al. WOMBAT: A scalable and high-performance astrophysical magnetohydrodynamics code. *The Astrophysical Journal Supplement Series*, 228(2):23, 2017.

[15] MPI Forum. MPI: A message-passing interface standard version 3.1. Tech. rep., University of Tennessee, Knoxville, 2015.

[16] F. Petrini, D. J. Kerbyson, et al. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, p. 55. 2003.

[17] M. J. Rashti, R. E. Grant, et al. iWARP redefined: Scalable connectionless communication over high-speed ethernet. In *Intl. Conf. on High Performance Computing (HiPC)*, pp. 1–10. IEEE, 2010.

[18] T. Schneider, T. Hoefler, et al. Protocols for fully offloaded collective operations on accelerated network adapters. In *42nd Int. Conf. on Parallel Processing (ICPP'13)*. Lyon, France, Oct. 2013.

[19] H. Weeks, M. G. F. Dosanjh, et al. SHMEM-MT: A benchmark suite for assessing multi-threaded SHMEM performance. In *Workshop on OpenSHMEM and Related Technologies*, pp. 227–231. Springer, 2016.