Process-Based Asynchronous Progress Model for MPI Point-to-Point Communication

Min Si,* Pavan Balaji*

*Argonne National Laboratory, USA *msi,balaji@anl.gov*

***Abstract*—The MPI two-sided communication model has been widely used in scientific applications for decades. The nonblocking version of the two-sided routines allows the application to potentially improve performance on many systems by overlapping communication and computation. In practice, unfortunately, the overlap is hard to achieve because of the limitations of the MPI internal progress engine and the underlying network. The traditional approach to resolving this issue is to implement an asynchronous progress engine based on either additional threads or hardware interrupts; however, such approaches may result in reduced computing power or expensive overheads.**

In this paper, we present a portable process-based asynchronous progress approach for two-sided communication in the PMPI-based Casper framework. It allows the user to specify an arbitrary number of cores on a multicore or many-core architecture and offload the point-to-point communication to these cores, thus ensuring asynchronous progress with low overhead. Unlike our previous work that supports asynchronous progress for the MPI one-sided model, a completely new design is needed for the message-matching-based two-sided model in order to ensure comprehensive semantics correctness as defined in the MPI standard. We present a detailed design of this two-sided model and compare it with the traditional thread-based approach on both a multicore Intel Xeon cluster and a many-core Knights Landing cluster.

***Keywords*-Asynchronous progress; MPI point-to-point; two-sided communications; multi-core; many-core;**

I. Introduction

MPI two-sided communication, also known as point-to-point communication, was introduced in MPI-1 [1]. It defines a message-passing model in which one process sends messages, and the other process receives messages. This model is used extensively in scientific applications to parallelize computation on distributed-memory systems, and it dominates the communication cost [2–7].

The overlap of communication and computation is a nontrivial task in two-sided communication and is especially important for balancing computation loads in irregular applications on large-scale systems [7]. The nonblocking routines of the two-sided model allow the user application to perform message posting and completion in separate MPI calls, thus potentially allowing the MPI runtime to overlap the data transferring taken by network hardware with the user computation performed by CPU cores. In practice, however, most MPI implementations do not immediately transfer medium or large messages at the first posting call (i.e., `MPI_ISEND`), but delay such transfer to a later MPI call or even until the completion call (e.g., `MPI_WAIT`) is triggered. The reason is that the posting call issues only a handshake header to the remote process to ensure both sender and receiver buffers are ready; and hence the data can be directly carried from the send buffer to the receive buffer without an expensive temporary copy. This procedure is known as the rendezvous protocol. In such case, unfortunately, the heavy data movement and the computation become hard to overlap.

The concept of MPI asynchronous progress is introduced to specifically address the communication overlap issue. Two traditional approaches are well known in the community. The first is a thread-based approach, where a background thread is created by each MPI process and continuously polls the MPI stack of the bound process; thus it can handle the data movement asynchronously while the master thread of the process is concentrating on the computation. This approach is usually implemented as part of an MPI implementation [8–11]. However, this model is limited in that we have to create as many threads as the number of processes on a node, thus resulting in significantly reduced computing power or heavy core oversubscription. In the hybrid MPI+threads programming model, the reduced number of MPI processes on a node can reduce the need for CPU cores dedicated to asynchronous progress threads. However, the expensive overhead of MPI multithreading safety still cannot be avoided [12]. The second asynchronous progress approach, based on hardware interrupts [13–15], has been used in IBM Blue Gene and Cray systems. In this approach, the network hardware issues an interrupt to the user process to trigger a software operation at the time of the message arrival. The concept of this approach is straightforward, but it relies on a special lightweight interrupt mechanism supported by hardware; otherwise, the performance may degrade because of expensive overhead from frequent system interrupts.

In this paper, we propose a process-based asynchronous progress model for MPI two-sided communication, called CasperII, to address the critical communication overlapping issue. It allows the application user to specify an arbitrary number of cores as background "ghost processes" dedicated to asynchronous progress for communication on the other processes on the node. The process-based approach offers more flexible core deployment than does the thread-based approach, and it does not involve any overhead from multithreading or system interrupts. In previous work, we studied the process-based concept for MPI one-sided communication [16], [17]. In this paper, we exploit a lightweight offloading mechanism following this concept to address the asynchronous progress problem in MPI two-sided communication. With this mechanism, we can transparently offload heavy communication requests from the user process to the ghost process at the message posting time. Thus the data movement cost can be fully overlapped with application computation.

Although the offloading concept is not new, several critical challenges have to be addressed in the design and implementation of our framework in order to maintain the transparency to both the applications and the various MPI implementations, as well as to ensure high performance. In particular, the address space isolation of system processes significantly increases the complexity of communication offloading because the performance-sensitive data movement has to been additionally synchronized between different processes.

To address these challenges, we have designed an offloading mechanism that moves message information between user and ghost processes through a lightweight, lock-free, queue-based framework allocated over the shared-memory region by using the portable MPI-3 function `MPI_WIN_ALLOCATED_SHARED`. We carefully consider various MPI semantics such as strict message matching and wildcard message support, and we maintain the correctness transparently within the framework. We also evaluate the framework through a set of microbenchmarks on the NERSC Cori supercomputers, including both the multicore CPU cluster and the many-core Knights Landing (KNL) environment.

II. Background

In this section, we briefly introduce the semantics of the MPI two-sided communication model, and we discuss the Casper framework for one-sided communication proposed in our previous work [16].

A. MPI Two-Sided Communication

MPI is the dominant parallel programming model on distributed-memory systems. The two-sided communication model was introduced in MPI-1 standard. In this model, two processes communicate with each other by explicitly posting MPI *send* and *receive* calls to match and move data.

Matching: The matching of a send and a receive call relies on the message envelope consisting of the communicator, source, or destination rank in the communicator and the tag. To be specific, the message sent from a source process src with envelope $\{comm,\ dest,\ i\}$ matches only a receive call on the process $dest$ with $\{comm,\ src,\ i\}$ or a *wildcard* receive. The $comm$ is a specific communicator, and i is an integer value of the tag.

Ordering: The messages sent from the same source process through the same communicator are guaranteed to arrive at the same destination process in the posting order. However, MPI does not maintain the ordering between messages sent from different source processes or received by different destination processes. The user program must ensure that the ordering of message delivery does not conflict with the *tag-matching* policy.

Wildcard: The wildcard receive functionality is a useful tool for many user scenarios. The receiver process can post the receive call with `MPI_ANY_SOURCE` as the source rank. Thus it can match the message from an arbitrary source process in the communicator. Similarly, the user can specify `MPI_ANY_TAG` as the matching tag value in the receive call. The actual matched source rank or tag value can be queried through a status object upon receiving a completion call. The user can pass `MPI_STATUS_IGNORE` to the completion call if such information is unneeded.

We note that the two-sided send/receives mentioned in this paper are all nonblocking routines, namely, `MPI_ISEND` or `MPI_IRECV`. The user program can complete a send or receive operation through the completion call `MPI_TEST{ALL|ANY|SOME}` or `MPI_WAIT{ALL|ANY|SOME}`.

B. Casper

Casper is a process-based asynchronous progress model for MPI one-sided communication [16]. It allows the user to specify a few numbers of cores on modern multicore or many-core architectures as background "ghost processes"; those cores then are kept aside by Casper at MPI initialization and dedicated to communication asynchronous progress.

Ghost Isolation: To transparently hide the ghost processes from the user application, we create an internal `COMM_USER_WORLD` communicator that consists only of user processes, and we replace `MPI_COMM_WORLD` at every user MPI call with this new communicator through the PMPI interface.[1] Therefore, any user-created subcommunicator also excludes the ghost processes.

Communication Redirection: The MPI one-sided model (known as RMA) is based on window exposure. Thus Casper can intercept the user window allocation call and internally expose the user window buffer also to the ghost process on the same node through the powerful `MPI_WIN_ALLOCATE_SHARED` function. Moreover, the one-sided model requires only the sender side (i.e., origin process) to issue RMA operations but does not require the receiver side (i.e.. target process) to explicitly issue a matching MPI call to process the message. Therefore, a user process can redirect its RMA operation issued to a remote target process by redirecting the operation to the corresponding ghost process on that remote node and reach the same memory location on the user target process with appropriate offset translation.

III. Basic Design

From this section, we extend the Casper framework to cover the important two-sided communication model. Although the basic isolation of ghost processes can be reused, the communication portion must be completely redesigned. The reason is that, unlike the well-studied one-sided model, the data

[1]MPI Profiling Interface.

buffer passed to two-sided send/receive calls is not naturally managed by MPI. More important, the two-sided operations are based on the explicit matching as we introduced in the preceding section. Therefore, we propose a new framework, called CasperII, to enable the asynchronous progress of two-sided communication. In the rest of this section, we present the basic design of CasperII; we discuss its semantics correctness issues and solutions in Section IV.

A. Design Challenges

In two-sided communication, we desire to overlap communication and computation on both the sender and receiver processes. To asynchronously take communication, we can offload the communication operations to a ghost process, thus allowing the ghost process to handle data movement while the user process can simultaneously perform computation.

Although the concept of offloading is straightforward, we have to address three critical issues in our implementation. First, the ghost processes are always hidden from user communicators, as we introduced in Section II-B, however, we now need them to explicitly issue MPI calls over some user communicators. Second, we cannot implicitly map the user-managed buffers used in two-sided calls; thus, an additional buffer sharing mechanism is required. Third, the send or receive operation must return an `MPI_Request` object for the user process to wait or to test the request to ensure communication completion. These semantics issues require that we design a lightweight routine to synchronize the message completion on the ghost process to the user process.

B. Offloading Framework

We describe our detailed design by dividing the framework into the following four steps.

Communicator Expansion: As the first step, we intercept all communicator creation calls through PMPI. Thus, whenever the user creates a new subcommunicator, CasperII can create an internal communicator including all user processes in the user subcommunicator and the corresponding ghost processes. Only the user subcommunicator is returned to the user to ensure the transparency of the ghost processes.

Shared Buffer Allocation: To avoid expensive copying of the user data between the user and ghost processes, the second step is to map the user buffer into the memory of the ghost process on the same node. To simplify this work, we require the user to allocate the data buffer through the MPI-3 `MPI_WIN_ALLOCATE_SHARED` function to register it as an offloadable message location. Doing so allows us to internally allocate a shared buffer over both the user and ghost processes. Since most applications use only a few statically allocated buffers for message exchange, this tradeoff is negligible. We cache the address, the base address on ghost process, and the size of every buffer. Thus we can validate the buffer and translate its starting address to be accessed by the ghost process at a later message posting.

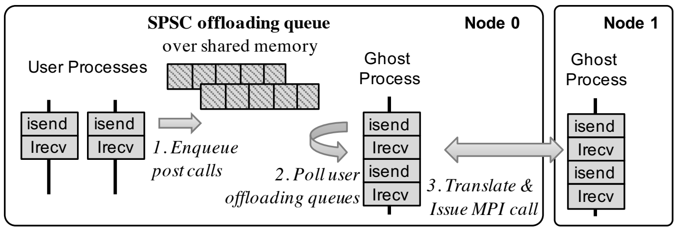


Fig. 1. Offloading user communication to a ghost process.

Post Offloading: The message posting call (i.e., nonblocking send or receive) is offloaded to the local ghost process through a *shared lock-free queue*. To minimize potential contention, we create a separate queue on each user process and let the ghost process poll one or multiple queues in a single-producer-single-consumer fashion. The queue elements are called *post cells*. We allocate both the shared queues and a fixed number of free cells on every process through the MPI-3 shared-memory functionality at MPI initialization. Figure 1 illustrates the example with a ghost process serving two user processes. We intercept all send/receive calls and enqueue a *post cell* with message information including the buffer address, count, datatype, tag, and communicator to the shared queue. An MPI generalized request (i.e., by calling `MPI_GREQUEST_START` function) is created and stored in a local hash with the corresponding cell's address and then returned to user program. On every ghost process, it always polls the user offloading queues. Once the ghost process has polled out an offloaded post, it replaces the user communicator to the internal expanded communicator and translates the source or destination rank to the corresponding ghost process's rank, then makes the corresponding call to MPI.

Asynchronous Completion: The ghost process frequently tests the completion of all issued communication requests. Once it has found a completed request, it updates an atomic flag located in the corresponding post cell. Whenever the user tests or waits for a request, we internally fetch the post cell associated with that request from the local hash and check its completion through the atomic flag. Once the communication is completed, the post cell is then returned to a free pool and reused by the next offloading.

To avoid unnecessary internal processing at every MPI message call, we enable this work only when the user passes an info hint to the communicator. To be specific, we allocate the shared buffer only when the user sets `shmbuf_regist=true`; we expand the communicator and offload messages only when the user specifies the `wildcard_used` hint with specific values as described in Section IV-C. We note that these info hints are not defined by the MPI standard and are CasperII-specific extensions.

IV. Correctness Challenges and Solution

Although the offloading framework enables asynchronous progress, several correctness issues may occur when multiple user processes share the same ghost process or when the user utilizes an MPI wildcard receive. In this section, we address these cases in CasperII in order to maintain correctness as specified in MPI standard while improving performance.

A. Message Matching with a Shared Ghost Process

On multicore and many-core systems, an MPI application launches multiple processes that reside on different CPU cores of the same node. CasperII keeps aside only one or a few cores from the application to reduce occupation of the computing resources. In this case, the same ghost process may be shared by multiple user processes. Therefore, different source or destination ranks in the messages offloaded from user processes can be translated to the rank of the same ghost process. Because of the message-matching policy as described in Section II-A, a send can mismatch a receive if the original destination process and the actual receiver process share the same ghost process in the same communicator and with the same tag.

To address this critical issue, we study two approaches. One is based on duplicated communicators for different user processes, and the other is based on internal tag encoding.

Duplicating Communicators: Consider a set of $\{$*communicator*, *rank*, and *tag*$\}$ information that compose a unique message-matching id. When the uniqueness of rank is weakened, we can transfer the differences of rank to the communicator portion, thus ensuring the same uniqueness. In detail, because the rank of multiple user processes may be confused only when they share the same ghost process, we can duplicate a separate communicator for each of those user processes at the user subcommunicator creation and use a different communicator for the communication offloaded from each user process. More specifically, we can duplicate N internal communicators for every user-created subcommunicator, where N is the number of user processes sharing the same ghost process. A send call targeting to the ith user process bound to ghost process g is translated to a call over the ith communicator with g as the destination rank. A receive call posted by the ith user process of g is offloaded to g and issued over the ith communicator. The user-specified tag remains the same. Following this strategy, the messages can always match correctly on the ghost processes. We note that for a highly optimized MPI implementation, the memory consumed by a duplicated communicator can be reduced to a constant value [18]; thus the side effect of this approach should be small.

Tag Encoding: The second approach is to keep the same *communicator* but shift the difference to the *tag* portion. MPI defines the attribute `MPI_TAG_UB` to indicate the upper bound of the tag value supported by the MPI implementation. We note that although the tag parameter is an integer variable, MPI defines a valid tag value that must be in the range from 0 to the value returned for `MPI_TAG_UB`, inclusive; the upper boundary must be at least $(2^{15}-1)$, as specified in standard. However, most MPI implementations support a larger range (e.g., $(2^{30}-1)$). Therefore, we can reserve a few bits in the tag range by replacing the `MPI_TAG_UB` value returned to the user program and encode the user rank information into the reserved bits. Figure 2 shows the structure of such an encoded tag. Instead of rank, here we use the user process's relative offset in its process group sharing the same ghost process as the "id"; thus the required bits are compressed to be only N_u/N_g, where N_u and N_g are the number of user processes and the number of ghost processes on a node, respectively. In practice, this value is usually within two digits, thus requiring at most 7 bits. By using the reserved bits, we can encode the destination rank's offset in the tag portion for every message, similar to the translation in the first approach, thus ensuring correct matching.

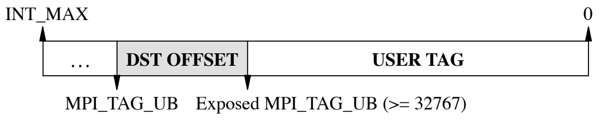


Fig. 2. Encoding rank offset in the tag integer.

Unlike the first approach, the tag-encoding method does not involve any communicator duplication, thus reducing the pressure of subcommunicator creation. However, we notice a limitation when the communication involves the wildcard tag (i.e., receive with `MPI_ANY_TAG` as the tag value). The reason is that the encoded tag value can no longer be recognized as a wildcard value in MPI. Therefore, the tag-encoding approach should be used only when the user explicitly hints that no wildcard tag is used and we have internally determined that sufficient tag space is available for any possible user offset.

B. Wildcard Receive

As introduced in Section II-A, a receive call may be specified with the wildcard `MPI_ANY_SOURCE` value for the source rank. Thus it can match with the message sent from any source in the communicator. A similar rule applies to the wildcard tag. Although the duplicating communicator approach ensures correct matching of any wildcard receive, a wrong source value can be returned in the receive `MPI_Status` object, since the message is sent by an internal ghost process.

To transfer the real user source rank, we can reuse the tag-encoding method. In other words, after replacing the communicator in the offloaded send call, we also encode the send user's offset into the tag value; on the receiver side, we replace the receive tag by `MPI_ANY_TAG`. Therefore, the offset of the user source can be transferred to the receiver ghost process and recovered to the original rank by integrating with the ghost source rank from the internal `status.MPI_SOURCE`. To simplify the recovery processing, we reorder ranks for the internal communicator at the time of communicator expansion in order to ensure that a user rank can always be calculated by $g+i$, where g is the rank of its ghost process and i is the offset.

Unfortunately, this method still cannot cover the case where the user specifies a receive with a wildcard source together with a distinct tag. The reason is that the receive tag is internally replaced by `MPI_ANY_TAG` in order to match an arbitrary source, thus resulting in loss of the user-specific tag.

C. *Info-Based Algorithm Selection*

We next define the hint interface to assist us in choosing the appropriate internal algorithm. We define a communicator info key `wildcard_used` with values `any_src|any_tag_same_tag|none`. The default `any_src` is the most relaxed value, which means that the user program may use wildcard source with any format of tag in the communication. In the case of `wildcard_used=any_src`, we have to disable the asynchronous progress for the communicator because of the uncovered wildcard corner issue. The second `any_tag_same_tag` means only that the wildcard tag or the same tag (e.g., all communication calls use tag 0) is used. Thus, with this info value we enable the asynchronous progress for the communicator and choose the communicator duplicating approach, including the case of `any_src|any_tag_same_tag`, which also enables the source offset encoding. The strictest value is `none`, which means that no wildcard source or tag is used. In this case, we choose the most lightweight tag-encoding approach. Note that `none` cannot be combined with the others.

V. Evaluation

In this section, we evaluate CasperII on the NERSC Cori Cray XC40 supercomputer (https://www.nersc.gov/users/computational-systems/cori/configuration/). Cori consists of a Haswell cluster and a KNL cluster. The Haswell compute node is composed of two 16-core Intel Xeon E5-2698 processors at 2.3 GHz, and the KNL compute node is composed of a single-socket Intel Xeon Phi Processor 7250 processor with 68 cores at 1.4 GHz. Each compute node is connected via the Cray Aries interconnect with Dragonfly topology.

We evaluate the proposed work on both clusters (using the default *quad cache* mode on KNL) and use the Cray MPI (version 7.4.4) and the Intel icc compiler (version 2017.2.174) in all experiments, by comparing with both the original MPI and two conventional thread-based implementations with dedicated cores. The first implementation is the default version in MPICH (denoted by Thread), [2] and the second is the CrayMPI optimized version (denoted by Thread(opt)). [3] We omit the comparison with the interrupt-based approach in CrayMPI since its significant overhead is already well studied in our previous work [16]. All experiments were compiled with an `-O2` flag and executed with explicit core-binding to ensure a dedicated core per process or thread. Every experiment was run ten times, and the average result is reported.

A. *Overhead Analysis*

In the first set of experiments, we analyze the additional overhead caused by the offloading framework.

[2]Set `MPICH_ASYNC_PROGRESS`=1; the thread always polls progress.

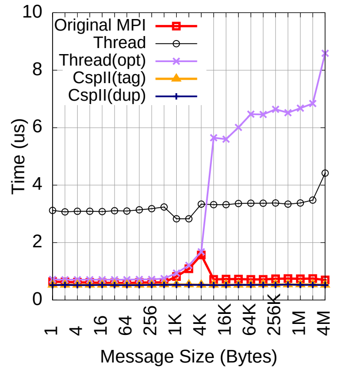
[3]Set `MPICH_NEMESIS_ASYNC_PROGRESS`=1; the thread polls progress only for rendezvous messages.

1) Offloading Overhead: We modify the OSU point-to-point latency microbenchmark [19] by replacing the blocking *send-recv* ping-pong with the nonblocking *isend-irecv*, followed by a *waitall* call. We also replace the data buffer allocation by using `MPI_WIN_ALLOCATE_SHARED` with `shmbuf_regist=true` info (see Section III-B) and specify the `wildcard_used` to `none` and `any_tag_same_tag` to trigger the tag-encoding approach and the communicator duplication approach, respectively. We denote the former as CspII(tag) and the latter as CspII(dup). We perform the experiment on two nodes each with a user process and a ghost process (or a helper thread in the thread approaches).

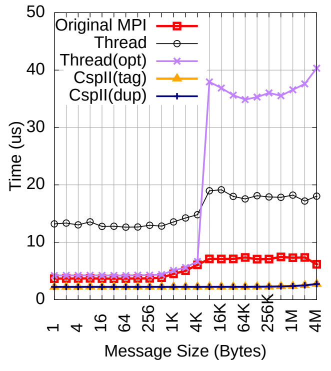
Figure 3(a) compares the message posting overhead (i.e., time of *isend-irecv* calls) on Haswell. The CasperII approaches always outperform the original MPI, reporting 0.1 to 1 μs less cost for message sizes from 0 bytes to 4 Kbytes and a consistent 0.2 μs gap for large messages ($\geq$ 8 Kbytes). The reason is that the user process only locally enqueues a post cell with message information to the shared queue at the posting call in CasperII. In contrast, the original MPI transfers the message data at the posting call for small messages ($<$ 8 Kbytes, known as the eager protocol) and issues handshake packets for large messages (known as the rendezvous protocol). The default thread-based approach reports more than 2 μs overhead than that of the original MPI because of lock contention in MPI. The CrayMPI optimized version eliminates such overhead for eager messages by disabling the thread polling; however, it delivers the most expensive overhead when such polling is enabled for rendezvous messages. We note that the visible overhead of the original MPI for 4 Kbytes message is because of an inappropriate eager threshold set at the system, a topic that is out of the scope for this work.

Figure 3(b) shows a similar trend of results on KNL. The original MPI takes 1 to 5 μs more overhead than CasperII, the default thread approach takes close to 10 μs more overhead than the original MPI, and the CrayMPI optimized thread approach reports up to 40 μs overhead when thread polling is enabled.

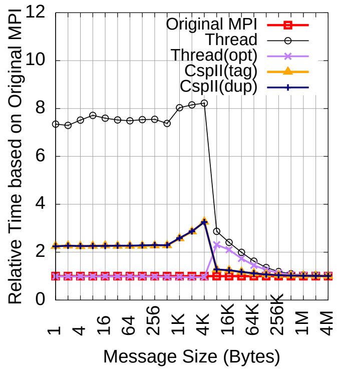
Figure 3(c) compares the overhead of the *waitall* call on Haswell. The baseline original MPI take 0.8 to 1 μs time for eager messages. The overhead increases with increasing of message size in the rendezvous range and eventually increases to 568 μs at 4 Mbytes message. Compared against the baseline, we observe close to 2 to 3x overhead in the offloading approaches for eager message sizes up to 8 Kbytes. This overhead gradually decreases with increasing of message size and eventually becomes negligible at large messages. The visible overhead at eager messages is caused by the slight delay of the offloaded message posting and the synchronization of completion between user and ghost processes. On the KNL platform, the offloading overhead seems less significant, as shown in Figure 3(d), reporting 1.6 to 2x higher cost than that of the original MPI. This overhead becomes negligible when message size is greater than 8 Kbytes. In contrast, the thread-based approaches are always more expensive when thread polling is enabled.



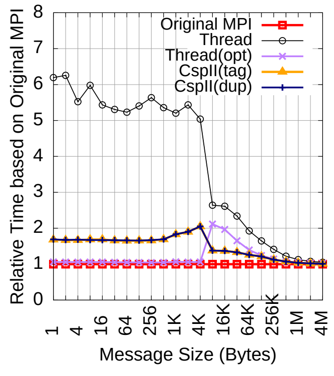
(a) Posting time on Haswell.



(b) Posting time on KNL.



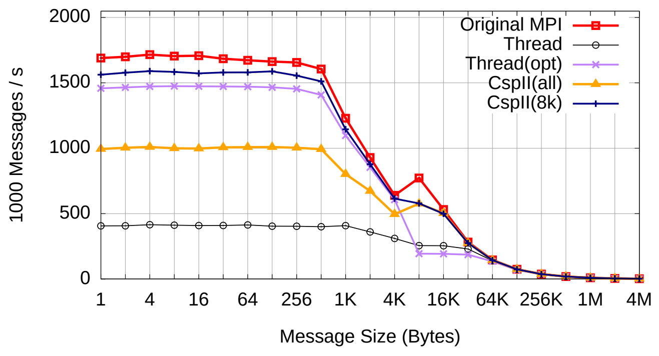
(c) Waitall relative time on Haswell.



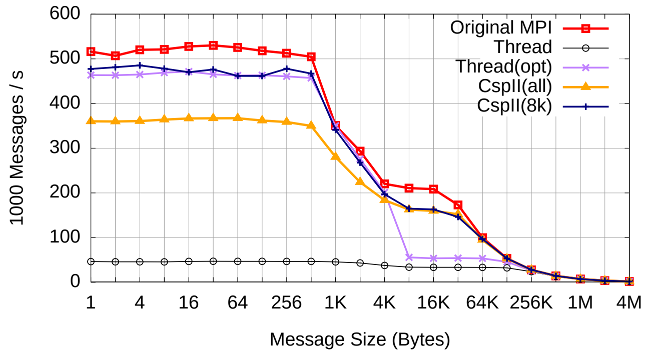
(d) Waitall relative time on KNL.

Fig. 3. Offloading overhead in nonblocking ping-pong. (Note: in order to demonstrate the overhead, we enable offloading for all messages by setting `offload_min_msgsz=0` in CasperII.)

Observing a consistent trend in both the CspII(tag) and CspII(dup) approaches, we conclude that the selection of different algorithms does not have an impact on the performance-critical routines. Thus we omit the second communicator duplication approach and show the tag-encoding-based approach in the remaining experiments. Moreover, we introduce the `offload_min_msgsz` communicator info hint as the offloading threshold based on message size (in bytes). Messages smaller than this value are issued through the original MPI routine similar to the optimized thread approach. This allows us to further investigate the impact of offloading-based asynchronous progress on small messages.



(a) Haswell.



(b) KNL.

Fig. 4. Message rate evaluation.

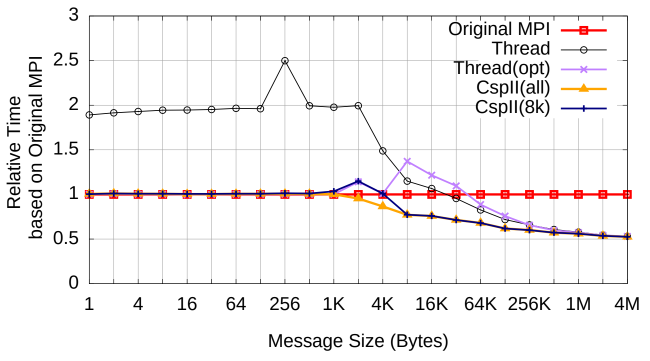
2) Message Rate: Having isolated each portion's overhead in the latency-style experiment, we compared the message rate of each approach. Figure 4(a) shows the results measured by using the OSU point-to-point message rate microbenchmark on Haswell. We show two options of CasperII. The first enables offloading for all message sizes (denoted by CspII(all)) by setting `offload_min_msgsz=0`, and the second enables offloading only for messages larger than 4 Kbytes by setting `offload_min_msgsz=8192` (denoted by CspII(8k)). The most expensive default thread approach reports up to a 4x degraded message rate compared with that of the original MPI. Both the optimized thread approach and the CspII(8k) approach disable asynchronous progress for eager messages. CspII(8k) outperforms the former; it reports up to 1.15x degradation at eager messages, and eventually becomes close to the original approach at a message size of 16 Kbytes. CspII(all) reports more overhead than CspII(8k) at small messages because of the offloading overhead as analyzed in Section V-A1. Figure 4(b) reports the results on KNL. CspII(8k) degrades only at most 1.3x and becomes close to the original MPI for medium-sized and large messages.

B. Asynchronous Progress

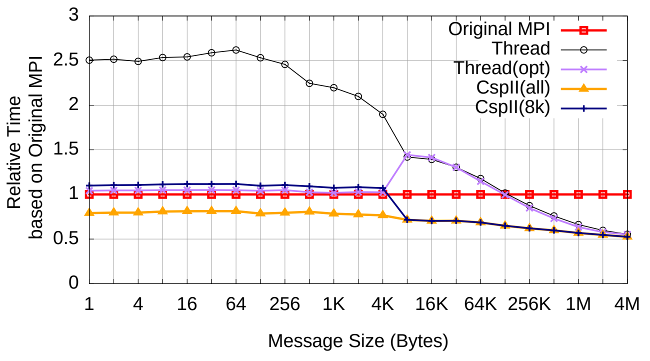
In the second set of experiments, we analyze the asynchronous progress improvements achieved in various scenarios.

1) Overlap: First, we focus on the computation and communication overlap between two processes. We extend the nonblocking ping-pong experiment used in Section V-A1 by adding a computation delay between the posting call and the completion call. Thus, every process performs as *isend-irecv-delay-waitall*. We still execute the experiment on two nodes each with a single user process. We vary the delay time with increasing message size by using the latency obtained in the offload overhead experiment (sum of the post time and the wait time) on each platform.

Figure 5(a) shows the execution time obtained by each approach on Haswell. Compared with the original MPI, although both CasperII and the thread-based approaches can reduce the



(a) Execution time on Haswell.



(b) Execution time on KNL.

Fig. 5. Computation and communication overlap in nonblocking ping-pong.

execution time by up to 50% (shown as 0.5x relative time) for large messages, the default thread approach results in about 50% degradation in small messages because of the overhead of multithreading. The optimized thread approach starts thread polling from 8 Kbytes. However, its threads contention overhead is more expensive than the benefit from asynchronous progress, thus resulting in up to 30% degradation in medium-sized messages. To better compare the overlap status, we then compare the overhead distribution of every internal portion for each approach, shown as Figures 6(a), 6(b), 6(c), and 6(d), respectively. The results clearly indicate that the *waitall* overhead can be fully overlapped with the *computation* portion in CasperII but that it cannot be hidden with only the original MPI and dominates the overall time at close to 50%.

Figure 5(b) shows the time observed on KNL. The thread-based approaches deliver more significant overhead at small and medium-sized messages, while the CasperII approaches can always reduce the time and deliver 20% to 50% improved performance when offloading is enabled.

2) Scalability: In our next experiment we scalethe comparison to multiple nodes. Instead of simple ping-pong, we choose the nearest-neighbor communication pattern with a two-dimensional Cartesian topology common in domain applications. Every process handles a 65536×65536 matrix with double elements and exchanges the boundaries with its four neighbors by using *isend-irecv* calls and a *waitall*. We insert a 300 μs computation delay between the posting calls and the waitall call and scale from 2 nodes to 512 nodes with a user process (and a ghost process in CasperII or a thread in the thread-based approach) on every node. Because only large messages (512 Kbytes) are transferred, we compared the original MPI with only the optimized thread approach and CasperII with an 8192 offloading threshold.

Figure 7(a) shows the execution time of each approach on Haswell. Both the thread approach and CspII(8k) consistently improve the performance compared with the original MPI. CspII(8k) delivers the best performance, taking 40% to 10% reduced time. By profiling the overhead of each portion, we notice that although the thread approach reduces a similar amount of overhead in the *waitall* call, it increases the posting overhead from 4 μs to 15 μs compared with the original MPI, while CspII(8k) costs less than 2 μs.

Figure 7(b) shows a similar trend on the KNL platform. We report up to 30% improved time in the CspII(8k) approach, while the optimized thread approach achieves only at most 15%.

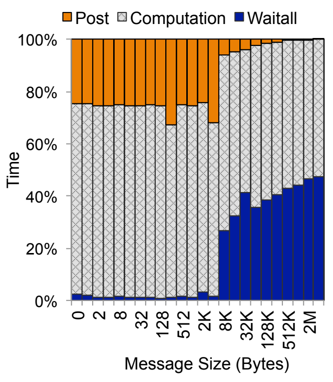
VI. Related Work

Traditional approaches to enable asynchronous progress for MPI two-sided communication usually rely on thread-based or interrupt-based models. Because of the capability of message asynchronous completion in the eager protocol, researchers also investigated the ways to utilize such feature. We then summarize the related work in each approach.

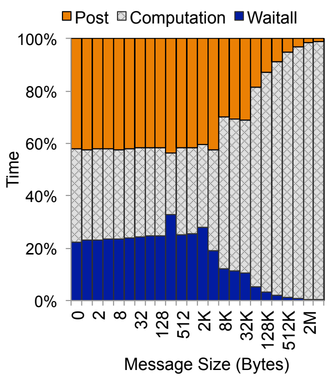
Thread-based asynchronous progress: The thread-based approach is the most common approach for supporting software progress and is supported in many MPI implementations such as MPICH and its derivatives [8][9][10]. This model requires every MPI process to create a background thread to asynchronously poll MPI internal progress in order to process messages. While being a generic approach for various MPI communication models, this approach suffers from the restriction that a background thread can make progress only for the process that spawned it. Thus it has to deploy at least as many background threads as MPI processes on the computing node. Consequently, the user must choose either to dedicate half of the computing resources or to perform expensive core oversubscription. Furthermore, this model forces the MPI runtime to maintain multithreaded safety, which results in further overheads because of lock contention and memory barriers [12].

Kandalla et al. [20] proposes a functional partitioning approach for asynchronous nonblocking alltoall by utilizing the concept of shared-memory mapping and progress threads. This work is implemented inside the MVAPICH runtime, thus the severe threads contention issue can be minimized with support from the specific environment.

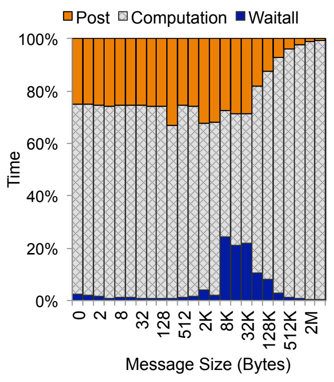
PIOMan [21] is a multithreaded communication engine supporting thread-based asynchronous progress. It divides rendezvous handshakes into multiple tasks and offloads them to background threads running only on idle cores. This approach, however, also suffers from a non-negligible overhead derived from the necessary multithreaded safety [22].



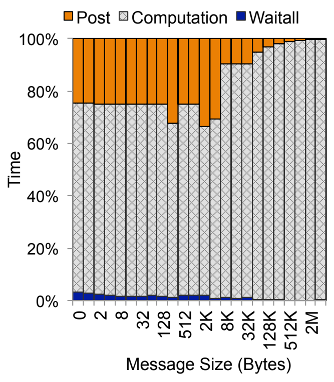
(a) Original MPI.



(b) Default thread approach.

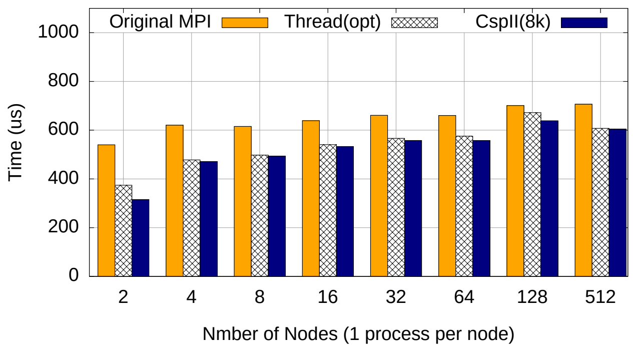


(c) Optimized thread approach.

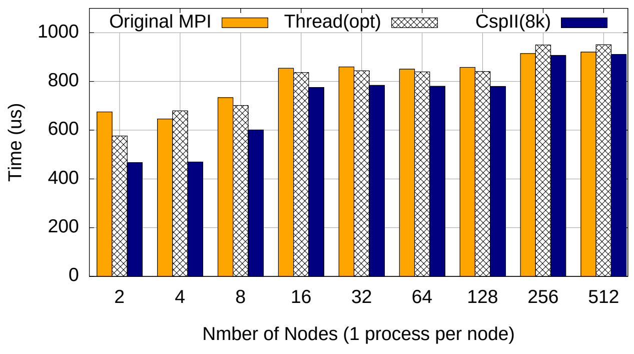


(d) CasperII (all message offloading).

Fig. 6. Profiling of computation and communication overlap in nonblocking ping-pong on Haswell .



(a) Execution time on Haswell.



(b) Execution time on KNL.

Fig. 7. 2D nearest-neighbor communication with asynchronous progress.

PAMI [23] utilizes the communication threads on IBM Blue Gene/Q for asynchronous progress. It uses a similar *offloading* approach where the communication requests are offloaded to the communication thread through atomic enqueue/dequeue operations. However, this work relies on special hardware and kernel support to reduce the overhead of offloading.

Vaidyanathan et al. [24] contributed a new approach for portable asynchronous progress in "MPI+X" applications by utilizing a dedicated thread together with a lock-free command queue. The MPI+X model often utilizes multiple threads over multicore or many-core systems to parallelize computation and employs only a single MPI process per node for internode communication. Thus only a single dedicated core is required per node. This is the most similar approach to the offloading framework in CasperII where the entire communication operation is offloaded to a separate CPU core. We note, however, that CasperII also natively support the traditional MPI-only model, which still widely exists in domain applications. Moreover, offloading communication operations to a separate process rather than a thread dramatically increases the complexity of the work to transparently maintain the semantics correctness.

Interrupt-based asynchronous progress: The other well-known asynchronous progress approach in the MPI community is the interrupt-based approach, which has been supported on both Cray [15] and IBM systems [13], [14]. This approach assumes that all processes are busy in external computation, thus utilizing a system interrupt to awaken the kernel thread to asynchronously trigger software operation at message arrival. The design is straightforward. However, the implementation often relies on a platform-specific interrupt engine; otherwise severe performance degradation might occur because of frequently issued interrupts.

Asynchronous progress in eager protocol: The eager protocol is wildly used in various MPI implementations for small message transfer. A key feature of this protocol is that a send message can be "eagerly" issued and finished asynchronously regardless of the arrival of the receive call. To utilize this feature also for large messages, Brightwell et al. [25] studied an eager-optimized rendezvous protocol on Portals network and summarized that the eager optimization still relies on the frequency of MPI calls made in the application and requires a large number of pre-posted receive calls. Different from the thread-based or interrupt-based approaches, the communication overlap can be obtained from the eager protocol is known to be limited because of the additional copy overhead and resource restriction from MPI implementations.

VII. Conclusion and Future Work

In this paper, we present an efficient and portable process-based asynchronous progress approach for MPI point-to-point communication. The user can specify a few cores as background ghost processes to be dedicated to asynchronous communication for the remaining processes on the node. The approach is based on message offloading where any user message buffer allocated from `MPI_WIN_ALLOCATED_SHARED` can be mapped into the memory address space of a ghost process on the node. Thus the communication operations can be offloaded to the ghost process through lightweight, shared lock-free queue without extra data copy.

Although the basic framework is straightforward, we have to resolve several critical issues in order to ensure the semantics correctness as defined in the MPI standard. We discuss efficient solutions to address these problems and demonstrate significantly improved performance and overlap of communication and computation through several microbenchmarks on both a multicore cluster and a many-core KNL environment.

We plan to further analyze and study the load balance of communication when multiple user and ghost processes are involved on a node, and we will study this approach in real applications. We plan to extend the Casper framework to cover MPI collective communication.

Acknowledgments

This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The experimental resources for this paper were provided by the National Energy Research Scientific Computing Center (NERSC) on the Cori Cray XC40 supercomputer. We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

References

[1] "MPI: A Message-Passing Interface Standard," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Sep. 2012.

[2] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "SWAP-Assembler: Scalable and Efficient Genome Assembly Towards Thousands of Cores," *BMC Bioinformatics*, Sep. 2014. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-15-S9-S2

[3] B. Joó, D. D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. W. Lee, P. Dubey, and W. Watson, *Lattice QCD on Intel Xeon PhiTM Coprocessors*. Springer Berlin Heidelberg, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38750-0_4

[4] B. Rosa and L.-P. Wang, "Parallel Implementation of Particle Tracking and Collision in a Turbulent Flow," in *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part II*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 388–397. [Online]. Available: http://dl.acm.org/citation.cfm?id=1893586.1893634

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[6] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. L. Marques, E. K. U. Gross, and A. Rubio, "Octopus: A Tool for the Application of Time-Dependent Density Functional Theory," *physica status solidi (b)*, vol. 243, no. 11, pp. 2465–2488, 2006. [Online]. Available: http://dx.doi.org/10.1002/pssb.200642067

[7] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang, "A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields," in *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 580–591.

[8] Argonne National Laboratory, "MPICH — High-Performance Portable MPI," http://www.mpich.org, 2014.

[9] The Ohio State University, "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu, 2014.

[10] Intel Corporation, "Intel MPI Library," http://software.intel.com/en-us/intel-mpi-library, 2014.

[11] Cray Inc., "Cray Message Passing Toolkit," http://docs.cray.com/books/S-3689-24, Cray Inc., Tech. Rep., 2004.

[12] W. Gropp and R. Thakur, "Thread-Safety in an MPI Implementation: Requirements and Analysis," *Parallel Comput.*, vol. 33, no. 9, pp. 595–604, 2007.

[13] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008, pp. 94–103.

[14] S. Kumar, Y. Sun, and L. V. Kale, "Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13, 2013, pp. 689–699.

[15] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems," in *Proceedings of the Cray User's Group Meeting (CUG)*, 2012.

[16] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures," in *Parallel and Distributed Processing, 2015. IPDPS 2015.*

[17] M. Si, A. J. Peña, J. Hammond, P. Balaji, and Y. Ishikawa, "Scaling NWChem with Efficient and Portable Asynchronous Communication in MPI RMA," *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium*, 2015.

[18] Y. Guo, C. J. Archer, M. Blocksome, S. Parker, W. Bland, K. Raffenetti, and P. Balaji, "Memory Compression Techniques for Network Address Management in MPI," in *Proceedings of the 2017 IEEE 31th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '17, 2016.

[19] The Ohio State University, "OSU Micro-Benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks, 2013.

[20] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, and D. K. Panda, "A Novel Functional Partitioning Approach to Design High-Performance MPI-3 Non-blocking Alltoallv Collective on Multi-core Systems," in *2013 42nd International Conference on Parallel Processing*, Oct. 2013, pp. 611–620.

[21] F. Trahay and A. Denis, "A Scalable and Generic Task Scheduling System for Communication Libraries," in *IEEE Cluster*, Sep. 2009.

[22] F. Trahay, É. Brunet, and A. Denis, "An Analysis of the Impact of Multi-Threading on Communication Performance," in *9th Workshop on Communication Architecture for Clusters (CAC)*, May 2009.

[23] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow, "PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 763–773.

[24] 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*, ser. SC '15, 2015.

[25] R. Brightwell and K. Underwood, *Evaluation of an Eager Protocol Optimization for MPI*. Springer Berlin Heidelberg, 2003, pp. 327–334.