SmartInterrupts: A Node-Wide Asynchronous Message Progression Technique

Kaushal Kumar*
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, Canada
kaushal.kumar@queensu.ca

Judicael A. Zounmevo
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, Canada
judicael.zounmevo@queensu.ca

Ahmad Afsahi
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON, Canada
ahmad.afsahi@queensu.ca

ABSTRACT

Traditional asynchronous message progression approaches have relied on either polling or interrupt threads. However, both of these approaches have limitations. Polling suffers from inefficient resource utilization, and interrupt-based approaches lead to the triggering of interrupts, even in scenarios where a natural overlap is expected. In this paper, we propose SmartInterrupts, a novel asynchronous message progression technique, where progression is performed using interrupt threads; however, these interrupts are not generated by the NIC, but instead by helper processes. These processes assist the MPI processes by polling on the incoming control messages and generating interrupts when conditions for message progression are met. A single helper process may be associated with multiple MPI processes. In our experimental evaluation, we found that just one or two helper processes are sufficient to provide close to 100% communication/computation overlap for most Rendezvous and RMA messages. We also observed significantly lower message latencies in our progression micro-benchmark. The overhead of our design is negligible, and it adds little to the memory footprint. Application results show performance improvements.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**;

KEYWORDS

Communication/computation overlap, Asynchronous message progression, MPI, RDMA

1 INTRODUCTION

The Message Passing Interface (MPI) [12] is the most popular programming model used in high-performance computing (HPC). MPI implementations typically use two different protocols for transferring small and large messages: *Eager* and *Rendezvous*, respectively. In the eager protocol, the sender transfers the data without any synchronization with the receiver. The Rendezvous protocol, on the other hand, makes use of control signals to synchronize between the sender and the receiver. Due to the synchronizations required in the Rendezvous protocol, non-timely arrival of the peers can lead to a serialization of communication and computation. That is, the communication may be deferred entirely to a blocking MPI call, without any overlap with the computation. This is detrimental to the performance of parallel applications. Similar inefficiencies are possible in one-sided communication as well. For example, in active target RMA synchronization, a non-timely opening of the exposure epoch can lead to the deferring of communications to the access epoch-closing call. Such inefficiencies are also possible in exclusive lock based passive target synchronizations.

Communication/computation overlap is a well-studied topic in HPC. Asynchronous message progression is one of the techniques to achieve overlap. It is a host-based approach [4, 7, 10, 22], where dedicated threads are responsible for progressing the communications initiated by the application threads. In a hardware-assisted approach, these threads may run on a specialized hardware like a NIC; in a host-based approach, these threads run on the host processor cores. This paper describes a novel host-based asynchronous message progression approach.

Traditional host-based progression threads employ either polling or interrupts. However, neither of them is perfect, and each has their own disadvantages [8]. Polling threads are more responsive but suffer from the problem of suboptimal resource utilization. Interrupt threads are not resource intensive but they are associated with several types of overheads. In this paper, we propose an approach that addresses the disadvantages of both polling and interrupt-based approaches. We do so, by associating several MPI processes with a single polling process called a *helper process* (HP). Also, each MPI process has its own interrupt thread which asynchronously calls the progress engine when interrupted. The HPs are made aware of the incoming control messages and the MPI calls that are looking for those messages. So, when they find a match, they trigger an interrupt to the interrupt thread of the appropriate MPI process. The contributions of the paper are as follows:

- We propose a novel asynchronous message progression mechanism, *SmartInterrupts*, that can improve the overlap of two-sided communications as well as one-sided communications.
- Our approach offers node-wide message progression by combining the strengths of both polling and interrupts. Our design requires a much smaller percentage of CPU cores for polling than the traditional polling threads.

*The first author is currently with Microsoft in Vancouver, BC.

EuroMPI '21, September 7-8, 2021, Garching, Germany

- For two-sided communication, the triggering of interrupts in our design is completely deterministic, hence, there are no overheads caused by futile interrupts. For one-sided communications, certain fringe situations exist where futile interrupts are triggered but we found this to be a necessary tradeoff to assure the scalability of the design.
- We observed close to 100% overlap for most Rendezvous and RMA messages used in the micro-benchmark studies. For two-sided communication, the results for asynchronous message progression were found to be ideal for almost all message sizes. The latency overheads of our designs were found to be negligible, with a small increase in the memory footprint. We also found that our design scales well with the number of MPI processes per node. Our application studies was performed using the NAS-SP and LU-Decomposition, and the results were observed to be quite positive.

The rest of this document is organized as follows. Section 2 presents the relevant background information and highlights the motivations behind our proposal. Section 3 discusses the research related to communication/computation overlap and message progression. In Section 4, we present the design of our proposed approach and describe how we have implemented it. We evaluate the performance of our design in Section 5. Section 6 concludes the paper and comments on future directions.

2 BACKGROUND AND MOTIVATION

Communication/computation overlap plays an important role in determining the performance of parallel applications. This is made possible by modern HPC interconnects in which the host involvement in communication is minimized by offloading the data transfer task to the NIC. In the context of MPI, overlap is said to be achieved if a message transfer initiated before the computation is *progressed* either partially or completely. Message progression can be referred to as the various steps that are performed by the MPI runtime to transfer a message from the application buffer of one MPI process to another. If a significant part of a message gets progressed in a blocking MPI call such as MPI_Wait, then that communication is said to be serialized with the computation. Since most of the communication burden is borne by the NIC, the time spent by the application thread in such a blocking call is unproductive and it would be better utilized in performing computations. This is detrimental to the performance of parallel applications where the execution times are of utmost importance.

It is possible to improve the overlap at the application layer by writing efficient code; however, we will focus on the techniques employed in the middleware. For point-to-point communication, such techniques can be classified into protocol improvement [17, 19, 23], hardware-assisted approaches [6, 26] and host-based approaches [7, 10, 22]. Hardware-assisted and host-based [9, 22] techniques exist for one-sided communication as well. However, most of the initial research focused on proposing non-blocking RMA synchronization alternatives to the existing blocking counterparts [25, 30]. An important motivation behind our work was to develop an overlap approach that could be applied to both one-sided and two-sided communication. Other than host-based techniques, most overlap improvement approaches are often unilateral. Hence, this paper is centred around the idea of asynchronous message progression, which is a common type of host-based overlap technique. In this approach, additional threads or processes are spawned, along with the MPI application processes, that are responsible for progressing the messages in parallel with the execution of the application threads.

Eager messages are progressed immediately at the sender side, and they do not provide much of an opportunity for overlap. For Rendezvous messages, the RDMA Read based Rendezvous protocol demonstrates a better potential for overlap than the RDMA write based protocol [24]. However, there is one scenario where an overlap cannot be achieved without other supporting measures. This scenario occurs when the receiver arrives ahead of the sender and is involved in a long computation when the control message arrives. When the receiver arrives at MPI_Irecv and does not find its *Ready To Send* (RTS) control message, it adds an entry to the *Posted Receive Queue* (PRQ) and returns. If the matching RTS for this MPI_Irecv arrives when the receiver is busy in a computation, then it will remain unacknowledged until the next call to the progress engine. If that call is the matching MPI_Wait, then the receiver cannot return until the entire message has been copied from the sender to its application buffer. This serializes the progression of the entire message with other computations, leading to no overlap.

Non-blocking RMA synchronizations offer a better scope for overlap when compared to the blocking versions. However, there are several scenarios where the communications may still end up being serialized. Interestingly, when compared to the inefficiency associated with the RDMA Read based protocol, the root-cause is not too different; that is, the message cannot be progressed timely due to the late acknowledgment of the critical control signal. For fence and *General Active Target Synchronization* (GATS), this control signal is the exposure epoch-opening signal. For an exclusive lock-based epoch, the lock-granting signal from the target is the critical signal. In each of these scenarios, if the origin fails to react timely to these signals, then the progression of the RMA operations can be deferred to the final epoch-closing call, leading to unproductive waits.

Polling and interrupt are the two basic mechanisms that are used in the progress engine of popular MPI implementations. Consequently, the asynchronous progression approaches are also based on them. This implies that the asynchronous progress thread can either be polling or based on interrupts. However, both ideas have their advantages and disadvantages.

The polling-based approach has long been regarded as the "silver bullet" [8] for asynchronous message progression and employed in popular MPI implementations like MPICH [13] and MVAPICH [14]. The idea involves the polling of work completions in a busy loop. To maintain the responsiveness, sleeping of the asynchronous thread is avoided. So, this approach leads to a 100% CPU utilization. Its advantage is that it is more responsive than an interrupt-based approach, but it leads to two major disadvantages: non-optimal resource utilization and oversubscription. In single threaded MPI applications, this may lead to the occupation of half of the CPU cores by the progress threads. These threads may not be actively involved in progressing communications but nonetheless use valuable computing resources. Using polling threads would lead to the oversubscription of the cores, incurring performance penalties.

The interrupt-based approach involves the use of a progression thread that sleeps until it is awoken by an interrupt. This interrupt is caused by a completion of a communication request. Upon waking, the thread progresses the recently arrived messages, requests another interrupt and goes back to sleep again. The advantage of an interrupt based approach is that it does not consume any CPU cycles while it is sleeping. The disadvantage, however, is the overheads associated with it. They are the interrupt cost, the cost of context switch that happens when the thread wakes up, and the cost of locking into the progress engine. In [10], the cost of the progress engine lock is replaced by an interrupt cost that is incurred in signaling the application thread to call the progress engine. In [24], the authors propose an interrupt-thread mechanism for sender-initiated RDMA Read based Rendezvous protocol. With this protocol, for non-blocking receive, there can be two scenarios, (a) the sender arrives before the receiver, or (b) the sender arrives after the receiver. In the latter scenario, an interrupt would be useful because the message can potentially be progressed earlier than the MPI_Wait at the receiver, leading to a receiver side overlap. In the former scenario, however, an interrupt will incur all the previously discussed overheads, without accomplishing anything useful. The proposal in [24] selectively generates interrupts and dynamically requests them to minimize the overheads but is not immune. For instance, consider a situation shown in Figure 1. Since MPI_Irecv1 does not find its RTS (from MPI_Isend1), the requests for interrupts will be turned on. Now, if the sender calls MPI_Isend for some other MPI_Irecv, say MPI_Isend2 for MPI_Irecv2, and if MPI_Irecv2 has not been posted yet then an undesired interrupt will occur, leading to all the previously discussed overheads.

Based on the above discussions, our research was aimed at addressing the following limitations associated with the polling and interrupt based asynchronous message progression:

- If a polling based mechanism is employed, then to optimize the resource utilization, we associate several MPI processes with a single asynchronous thread or process. We believe that modern clusters with fat-nodes would benefit from node-wide polling threads. This way, the performance of polling is achieved with minimal cost.
- If an interrupt based mechanism is employed, then to minimize the overheads, we either suppress futile interrupts or deterministically generate them only when they are useful.

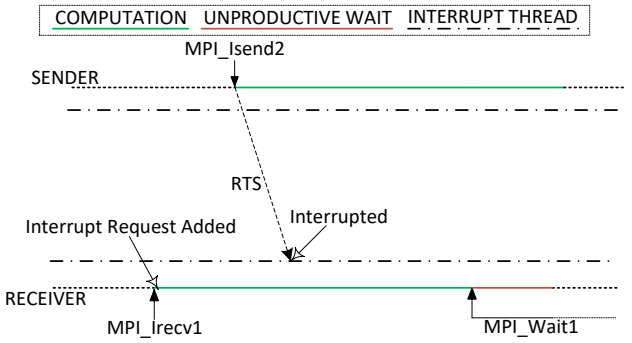


Figure 1: Interrupt thread based asynchronous progression

3 RELATED WORK

3.1 Point-to-point Communication

For host-based overlap in GPUs, the authors in [7] propose a scheme of over-decomposing the tasks and then over-subscribing the hardware with more threads than the hardware limits. Traditional methods like polling and interrupt based asynchronous message progression approaches [8, 10, 24] are discussed in depth in Section 2. In [4, 5, 28], the authors describe schemes that opportunistically use CPU cores for message progression. In [4, 5], the communication can be offloaded to ltasks which are similar to tasklets available in the kernel-space. These are scheduled to run asynchronously on idle cores when they become available. If idle cores are not available, then ltask execution is triggered by timers or at explicit polling points. Similarly, [28] presents opportunistic progression by stealing CPU cycles from the application thread to poll for message completions. In [20], the authors propose a mechanism to detect when asynchronous communication progress is needed to minimize the context switching between the main thread and the asynchronous progression thread.

The MPI Profiling Interface (PMPI) is employed in [21, 27] for asynchronous progression. In [27], non-blocking point-to-point MPI calls and I/O calls are redirected through PMPI to the progress thread. In the progress thread, MPI_Test or MPI_Wait may be called for message progression. Depending on the implementation, this may lead to inheriting the shortcomings of polling or interrupts. CasperII [21] is a process based asynchronous message progression technique for non-blocking point-to-point MPI communications. The progression processes are called ghost processes, which can progress messages for multiple MPI processes. In this approach, the MPI middleware uses PMPI call redirection to intercept the non-blocking calls from various processes and offloads them to the ghost processes. The ghost process then issues the redirected MPI call to the intended sender or receiver and progress the message on behalf of the associated MPI process. Similar to SmartInterrupts' helper processes, a user-defined quantity of ghost processes can be spawned and each process may be responsible for the progression of multiple MPI processes. However, these two approaches are fundamentally different. Unlike CasperII, SmartInterrupts does not rely upon PMPI and the helper processes do not progress the messages themselves, but instead trigger the interrupt threads for this task. Also, in order to use CasperII, certain hints need to be used in the application that are not part of the MPI standard. In contrast, the SmartInterrupts approach does not require any change to the application code.

3.2 RMA Communication

Like two-sided communication, host-based approaches have been proposed for RMA as well. These approaches include opportunistic message progression [30] and asynchronous message progression [9, 22]. RMA GET/PUT operations do not need to involve the target as those calls are supported in hardware through RDMA Read/Write [11]. However, for RMA operations involving non-contiguous datatype like MPI_PACKED, the target must be involved to unpack the data received on the contiguous temporary buffer and

get the non-contiguous message. In [9], a thread-based approach is used to asynchronously progress RMA operations involving non-contiguous data. Intra-node RMA communications require active participation of the CPU as there can be no support from the NIC for such communications. In [29], the authors noted this inefficiency and observed that in certain scenarios, inter-node RMA communications exhibit 100% overlap with some spare computation time. They exploit this residual overlap potential to overlap intra-node communications by deferring the transfer of such messages to the spare computation time. To address the overlap inefficiencies associated with the non-blocking RMA synchronizations, [30] proposes an opportunistic message progression technique. This approach involves the stealing of CPU cycles from the application thread to progress pending RMA communications.

As in CasperII [21], Casper [22] is a process based asynchronous message progression technique for RMA communications. It uses PMPI call redirection to intercept the RMA operation calls from the origins and offloads them to the ghost processes. The progression of the RMA operation then becomes the responsibility of the associated ghost process. The RMA operations are automatically redirected in Casper, therefore, the MPI processes must rely on their ghost processes, even in a situation where an MPI process is free to progress its own RMA operation. This is inefficient because it causes a wastage of CPU cycles at the MPI process and because redirections are slower than having the MPI process call its progress engine itself. In SmartInterrupts, the progression thread is not activated if a call to the progress engine is active at its MPI process.

4 DESIGN AND IMPLEMENTATION

In this section, we discuss the design and implementation of SmartInterrupts, which is aimed at addressing the research problems highlighted in Section 2. SmartInterrupts is a hybrid approach that uses both polling and interrupts to complement the shortcomings of each other. Like other interrupt based asynchronous message progression approaches, the progression thread in SmartInterrupts is an interrupt thread that progresses messages whenever interrupted. However, the interrupts are not generated by the NIC's device driver, and they are not part of the network software stack. Instead, the interrupts are triggered by polling processes called Helper Processes. Each of these helper processes may be associated with multiple MPI processes.

4.1 Point-to-Point Communication

4.1.1 Asynchronous Message Progression Mechanism. Figure 2 illustrates the data-movement and signaling involved in the two-sided SmartInterrupts approach. It is designed around the sender-initiated RDMA Read based Rendezvous protocol to leverage its natural overlap potential. When the sender arrives at MPI_Isend (C), it sends a modified RTS control message to the receiver. This modified control message has two parts. The first part is the original unmodified RTS control message that contains the message envelope, and the second part is a duplicate of the message matching information of the first part. When this modified RTS control message arrives at the receiver, the first part gets copied to the receiver's communication buffer (F) and the second part gets copied

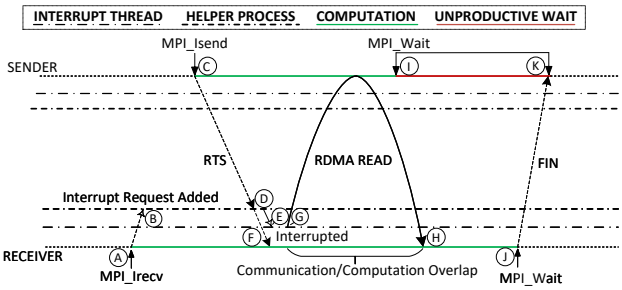


Figure 2: Asynchronous message progression mechanism of two-sided SmartInterrupts

to a shared buffer between the receiver and its helper process (D). This is required to make the helper process aware of the incoming RTS. We call this buffer the receiving *Interrupt Control Buffer* (ICB) and the data that gets copied to this buffer from the sender is called *Interrupt Control Data* (ICD). Note, that both copies at (D) and (F) are done using Send/Receive semantics.

When the receiver arrives at the matching MPI_Irecv call (A) and does not find its RTS, it requests an interrupt from its helper process (B). It does so by adding a request to another shared buffer between the MPI process and its helper process. This buffer is referred to as the *Interrupt Request Buffer* (IRB) and each request added to this buffer is called an *Interrupt Request Data* (IRD).

Using these buffers, the helper processes continuously poll on the incoming RTSs and try to match them with the interrupt requests submitted to them (B). An interrupt is triggered to the receiver's progression thread if a match is found for its request. In the illustrated scenario, the MPI_Irecv's request is already present in the shared buffer when the RTS arrives at the receiver. The helper process eventually finds the matching RTS for the initially submitted request and triggers an interrupt to the interrupt thread (E). This causes the interrupt thread to wake up and call the progress engine. The progress engine then progresses the message by issuing the RMDA Read (G). Since the communications are offloaded to the NIC, the interrupt thread can immediately go back to sleep after issuing the communication calls. For the illustrated scenario, this leads to a communication/computation overlap between (G) and (H). Since the communication is entirely progressed before the call to the MPI_Wait at (J), the receiver does not have to block and can send the FIN control signal instantly, which is also offloaded. On the other side, the delay at the sender's MPI_Wait depends upon when it gets called, as it cannot return until it receives the FIN control signal. In this case, the sender has to wait between (I) and (K); however, it could have been wait free if it had issued the call after the arrival of FIN (K). This mechanism is further augmented by a few optimizations that make the helper processes more efficient and eliminate unnecessary overheads. The details of these optimizations are discussed in Section 4.1.3.

4.1.2 Core Components. Figure 3 illustrates the important components of SmartInterrupts, in which each MPI process has one application thread and one interrupt thread for asynchronous message progression. The application thread is the main thread which can be safely oversubscribed with the interrupt thread, as the latter

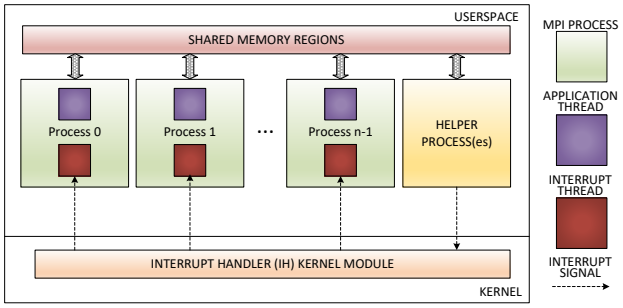


Figure 3: Core components of SmartInterrupts

does not poll periodically and lays dormant until interrupted. Other important entities are the interrupt handler kernel module, the helper processes and the two shared memory regions between the MPI processes and the helper processes, namely, ICB and IRB. The kernel module implements a read() system call that the interrupt-threads call to go to sleep. It also implements a write() system call that can be called by the helper processes to generate an interrupt to a specific sleeping thread. The kernel module creates an array of wait queue elements, one for each interrupt thread. The module then uses these wait queue elements to change the state of a sleeping thread when an interrupt generation request is made for it by a helper process. As the name suggests, the helper processes assist the interrupt threads and in turn assist the MPI processes to asynchronously progress messages. However, it is important to note that they only trigger the interrupts. They do not call any network related API functions, and calling the progress engine is still the responsibility of the asynchronous progression interrupt thread. The interrupts are triggered based on the data in the shared buffers, which bridge the MPI process to its helper process. The number of helper processes per node can be specified as a command line argument at the time of execution. However, a minimum of one per node is required to trigger any interrupt to the progression threads. The helper processes use a busy loop, so for maximum performance, each of them needs to run on an individual CPU core.

4.1.3 Implementation. We have implemented our design in MVAPICH, which supports InfiniBand [1] through OFED [15]. Necessary modifications were made to the buffer structures of MVAPICH2 to address the different design objectives of SmartInterrupts. In MVAPICH2, the registered memory regions used for Send/Receive InfiniBand communications are called Virtual Buffers (VBUF), and they are of two types, sending VBUFs and receiving VBUFs. To the existing pool of VBUFs, one more is added to each of the types. These are called sending and receiving ICB. As mentioned previously, the receiving ICB is a shared memory region between the MPI process and its Helper Process. This shared buffer is stridded, with each stride allocated to an MPI process.

MVAPICH2 provides two ways of sending eager messages, RDMA Write and Send/Receive, but the latter suits the design better as it allows storing data in a non-contiguous remote buffer. In MVAPICH2, the eager message is copied to the sending VBUF of the sender which, after being progressed, ends up at the receiving VBUF of the receiver. In SmartInterrupts, the eager message is copied to the sending VBUF like before. In addition to that, if the eager message is an RTS then its message matching tuple is copied to the sending ICB. This tuple is referred to as the ICD. If the eager message is not an RTS, then it is appropriately indicated in the ICD. After the send request gets processed, the eager message lands at one of the receiving VBUFs and the ICD lands at the receiving ICB. In Figure 2, these events are indicated as point F and point D, respectively.

In addition to the sending and receiving ICBs, another shared buffer, called the IRB is added between the MPI processes and their helper processes. If the receiver arrives at MPI_Irecv and does not find its RTS in the Unexpected Message Queue (UMQ), then it adds a request to the Posted Receive Queue, as well as an entry to the IRB (point (B) in Figure 2). This entry, called IRD, is essentially a request to the helper process to trigger an interrupt to its interrupt thread.

At this juncture, the helper process knows about the RTSs that have arrived, as well as the receive requests that are looking for them. Therefore, it has enough information to trigger an interrupt and progress the message asynchronously if a match is found between the ICDs and IRDs. However, the implementation is augmented by the following optimizations which ultimately decide the triggering of an interrupt.

Optimization 1: The helper processes continuously poll on their ICBs to look for RTSs. If the progress engine is already active in an MPI process, then it will most likely progress the pending messages of that process. In such a case, polling for its ICDs at the same time by the helper process serves no purpose and delays the message matching of other MPI processes that are associated with the helper process. To avoid this, in addition to storing the IRDs, a region in the IRB also contains information about the progress engine semaphore. This makes it possible for the helper process to know if the progress engine of its MPI process is already active.

Optimization 2: If a match is found at the helper process, then an interrupt may be triggered to the interrupt thread of the MPI process. However, it is possible that the MPI process has acquired the progress engine lock during the message matching process. Generating an interrupt in this case would incur several unnecessary overheads. To avoid this, the status of the MPI process' progress engine lock is examined before triggering the interrupt, and the interrupt thread is only awakened if the lock has not been already acquired.

In addition to the above-mentioned optimizations, the relative locality of the helper processes and their MPI processes has also been accounted for in the implementation using the hwloc library [3]. Wherever feasible, this ensures that the helper process is pinned to the same CPU socket as the rest of its MPI processes.

4.2 One-Sided Communication

4.2.1 Asynchronous Message Progression Mechanism. With respect to an origin epoch O, we say that a target epoch T is *late at opening* ($late_1$) if it opens after O opens; and T is *late for transfer* ($late_2$) if T is still not open by the time O tries to ship its first RMA payload towards T. The scenario discussed in $late_1$ has no latency impact on O but $late_2$ does; hence, the $late_2$ scenario is discussed in the motivation and the subject of attention of this research.

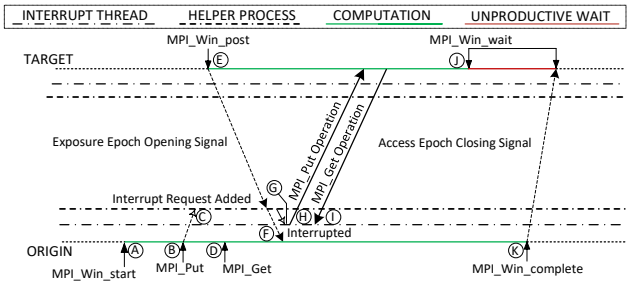


Figure 4: Asynchronous message progression mechanism of one-sided SmartInterrupts for GATS

Figure 4 illustrates the data-movement and signaling with SmartInterrupts in a GATS epoch. This figure shows SmartInterrupts' solution for the $late_2$ scenario. When the origin issues its RMA operation at (B) and realizes that the target has not opened the exposure epoch yet, then it queues the RMA communication for later and writes to the IRB at (C) to request help from its helper process. This essentially enables the interrupt mechanism by informing the helper process that an interrupt should be triggered to the asynchronous thread when the exposure epoch opening signal arrives.

When the target calls the MPI_Win_post function at (E), it informs the origin about the opening of the exposure epoch and writes to the ICB at (F). Like the two-sided design, the helper process continuously polls on the data in the ICB. When it detects the exposure epoch opening signal, it checks if the origin has requested for interrupts. When the two conditions are satisfied, the helper process triggers an interrupt at (G) to progress the pending MPI_Put and MPI_Get RMA operations. SmartInterrupts addresses similar inefficiencies associated with Fence and exclusive lock-based RMA syncronizations; however, due to space limitations, we limit our discussion to GATS.

4.2.2 Implementation. The core components of the One-sided SmartInterrupts are the same as its point-to-point counterpart. The only difference being the type of message-matching information stored in the shared buffers, which eventually decides the interrupt triggering mechanism. One-Sided SmartInterrupts was implemented on top of NewRMA which is based on MVAPICH2 and incorporates the designs of [29, 30].

At initialization time, all the processes in the same node create, in shared memory, an array of ICDs and IRDs. The ICD/IRD at index i belongs to the process of rank i in the communicator that encompasses all the MPI processes on the node. This implies that there is a single pair of ICD and IRD associated with each MPI process. This is in contrast with the two-sided implementation where the count of these fields is dependent on the number of MPI communications. In One-Sided SmartInterrupts, each ICD is a flag that is set to True at the origin by its remote target when it opens its exposure epoch. This field is the only one that can be manipulated remotely by out-of-node targets. For a helper process, the IRD field is a Boolean flag. However, for the MPI process that owns this field, it is a counter that contains the number of distinct target epochs that are $late_2$ in opening their exposure epoch with respect to the current MPI process. When IRD is zero, it is perceived as False by the helper process.

The design enforces the following actions. If a given target is still not open by the time the origin epoch attempts to issue the first RMA operation towards it (Point B in Figure 4), then, the origin increments its IRD by one. This is perceived by the helper process as setting the IRD to True. When the target opens its exposure epoch (Point E in Figure 4), it sets its origin's ICD to True by performing an RDMA-write into the address space of the origin. The helper process continuously polls on the ICD and IRD values of its associated MPI processes. When the ICD and IRD of a particular MPI process are simultaneously found to be True, the helper process triggers an interrupt to the asynchronous progression thread of that MPI process, which causes the progression of the pending RMA operations of that process' epoch (Point G, Point H and Point I of Figure 4, respectively). Since the progress engine can be fired by either the helper process or the application thread, the interrupt thread first checks if the progress engine lock is available. If unavailable, it is likely that the pending RMA operations will be progressed by the other progress engine call; hence, the interrupt thread goes back to sleep in this situation. Each time the progress engine is fired in an origin, regardless of the thread that fired it, its ICD is reset to False. Also, its IRD is decremented once for each target epoch that was previously flagged as $late_2$ and is now opened. This last sentence implies that a single interrupt can be amortized over multiple late targets.

Assuming that there are n distinct remote target epochs linked to origin epochs in P, the following guarantees exist:

- G0: The helper process will fire the progress engine at most n times.
- G1: If m of the previously late targets are open before the next time the helper process reads IRD of a given process P, then the number of times the progress engine would fire in the future is reduced by m; and that reduction is associated with an O(1) operation.

If a process P is linked to two remote target epochs T_0 and T_1; and T_0 is $late_2$ (IRD is not 0) while T_1 is only $late_1$, the interrupt thread can be awakened when T_1 gets opened. In the context of this research, such an interrupt is futile. At any given time, assuming that there are n out-of-node targets linked to P, and m of them are currently $late_2$ and k are currently $late_1$, then the following guarantees exist:

- G2: The number of futile interrupts is currently capped by k. When a target transitions from $late_1$ to $late_2$, k decreases and so does the possible number of futile interrupts.
- G3: Futile interrupts are possible only if both m and k are greater than 0. This guarantee means that there will never be any futile interrupt if help is not needed.

5 PERFORMANCE EVALUATION AND ANALYSIS

5.1 Experimental Setup

We used a 32-node cluster at HPC-AI Advisory Council to evaluate the micro-benchmark and application performance of SmartInterrupts. The nodes are connected to a single Switch-IB SB7700

switch. Each node is equipped with two 10-core Intel Xeon CPUs (E5-2680) running at 2.8GHz, 64GB DDR3 memory and a Mellanox ConnextX-4 NIC. The software environment consisted of the Red Hat Enterprise Linux 7 with kernel version 2.6.32-431, OFED 3.4-1, and MVAPICH2-2.2a.

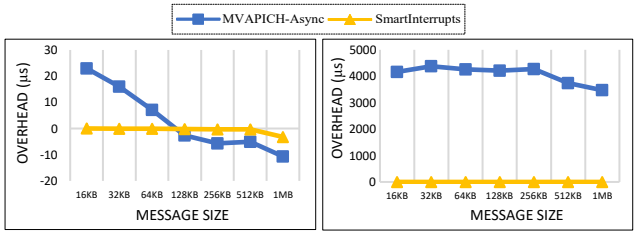
5.2 Point-to-Point Communication

The micro-benchmark performance evaluation of two-sided SmartInterrupts focuses on latency overhead, communication/computation overlap, asynchronous message progression, scalability and memory footprint. For this, we designed micro-benchmarks based on the ideas presented in [18], and we will describe them below. To evaluate the effectiveness of our design, we use several pairs of senders on one node and receivers on another node communicating in parallel. Each MPI sender process communicates exclusively with one MPI receiver process. However, before the start of each communication, the senders and receivers are synchronized so that all MPI_Isends are posted together and all MPI_Irecvs are posted together. Also, since our design is aimed at receiver side overlap, all the timing measurements are performed at the receiving end.

In all our micro-benchmarks, we use 10,000 iterations and discard the first 200 iterations to account for cache warm-up. The results of our implementation were compared to that of MVAPICH polling based asynchronous progression and with its default, no asynchronous progression setting. We refer to them as MVAPICH-Async and MVAPICH, respectively. We don't compare with Open MPI [16] because it had no support for asynchronous progression.

On our system, the default eager threshold of MVAPICH2 is 16KB and all of the following experiments were conducted with messages sizes of 16KB-1MB. We experimented with several different configurations of MPI processes and helper processes on each node, however, due to space limitations we will limit the results to two configurations. The first with 9 MPI processes and 1 helper process per node (9-1) and the second with 18 MPI processes and 2 helper process per node (18-2). This is to be able to compare the performance of SmartInterrupts with both non-oversubscribed and oversubscribed scenarios of MVAPICH-Async. Note that the interrupt threads in SmartInterrupts remain oversubscribed with the main thread even if there are spare cores available.

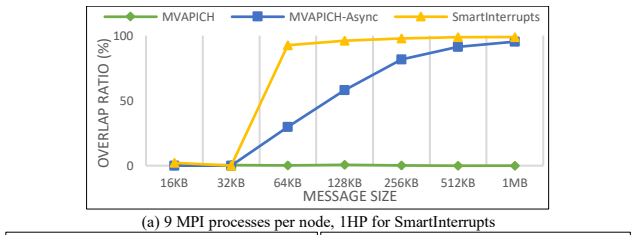
5.2.1 Latency Overhead. The latency overhead is the difference between the latencies of the asynchronous message progression approaches and MVAPICH at the receiver. The latency is the time stamp difference between the entrance of the MPI_Irecv and the exit of the associated MPI_Wait, with no computation inserted in between. We denote this as l_0. In Figure 5, we compare the overhead of SmartInterrupts with MVAPICH-Async for different message sizes. In Figure 5(a), the overheads of both implementations are low, and just a few hundred nanoseconds for SmartInterrupts. In Figure 5(b), SmartInterrupts continues this trend; however, the overhead of MVAPICH-Async significantly increases. This is because, there are 18 MPI processes on each node which causes the oversubscription of main threads and poling threads. In SmartInterrupts, oversubscription only happens when the interrupt thread is called into action, which is unlikely in this micro-benchmark because of the immediate arrival of MPI_Wait after MPI_Irecv. Note, that the negative values of overhead is caused by the relative timing of



(a) 9 MPI processes per node, 1HPs for SmartInterrupts

(b) 18 MPI processes per node, 2HPs for SmartInterrupts

Figure 5: Two-sided latency overhead results over MVAPICH



(a) 9 MPI processes per node, 1HP for SmartInterrupts

(b) 18 MPI processes per node, 2HP for SmartInterrupts

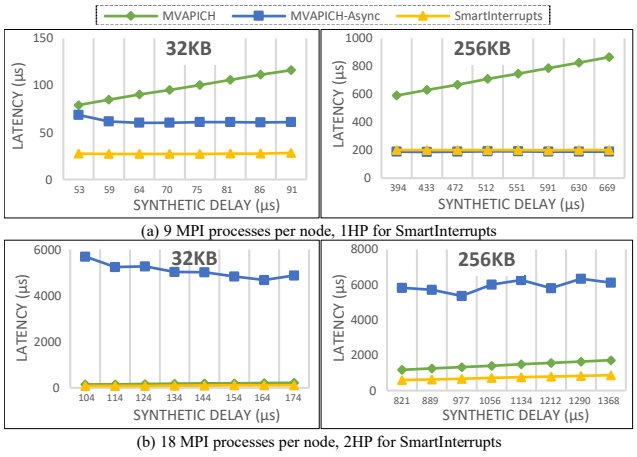
(c) 17 MPI processes per node, 2HP for SmartInterrupts

Figure 6: Communication/computation overlap for point-to-point communication

the progress engine call between MVAPICH and the asynchronous progression approaches. In the latter, the message progression does not need to wait until the call to the MPI_Wait in the application, as the asynchronous progression thread can make progress on the message transfer even before the call to the MPI_Wait.

5.2.2 Communication/Computation Overlap. To measure the overlap potential of the different implementations, we followed the procedure mentioned in [18]. This involves the insertion of a synthetic delay between the MPI_Irecv and its MPI_Wait. This delay, denoted as c_m, simulates a real-world computation. The process pairs are made to communicate with progressively increasing values of c_m, starting with 0 and reaching 1.1 times of l_0. To ensure that all the process pairs cease to communicate upon reaching the stopping criteria, the value of c_m remains consistent across each receiver. When the iterations are over, all the receiving processes send the value of T_{diff} and *overlap_ratio* to the local process 0 of the receiving node, where $T_{\text{diff}} = 1.1\ l_0 - l_m$. Here, l_m is the latency with the synthetic delay. The largest iteration for which T_{diff} is either positive or zero across all process is used for the calculation of the average overlap ratio.

Figure 6(a), Figure 6(b), and Figure 6(c) show the communication/computation overlap for 9, 18 and 17 MPI processes, respectively. As expected, MVAPICH exhibited negligible overlap in all



(a) 9 MPI processes per node, 1HP for SmartInterrupts

(b) 18 MPI processes per node, 2HP for SmartInterrupts

Figure 7: Asynchronous message progression of point-to-point communication

scenarios. For 9 processes, both the asynchronous progression techniques performed well with close to 100% overlap for messages greater than 64KB. SmartInterrupts performed slightly better than MVAPICH-Async, even though the polling threads were not oversubscribed. For message sizes of 16KB and 32KB, the message latencies are small, leading to even shorter synthetic work. Also, the computation required for their processing is comparable to their communication time. These reduce the overlap potential and explain why neither of the implementations show any overlap for those messages. For 18 processes, the results of MVAPICH-Async suffer due to oversubscription but SmartInterrupts shows overlap in the range of 70-75% for messages greater than 64KB. However, this overlap increases to almost 100% if one of the cores is left spare for the operating system's processes and the process that handles the Verbs API calls. We show this in Figure 6(c). Note that for 17 and 18 processes per node, the increased number of communicating pairs increases the network load as well. This provides enough time to the helper processes, resulting in the improved overlap of 32KB messages.

5.2.3 Asynchronous Message Progression. The progression results are shown in Figure 7. Here, x-axis is the synthetic delay ranging between 2 to 3.5 times of l_0 and y-axis is the message latency at the receiver. In these results, the message latencies of MVAPICH serves as the baseline as it has no asynchronous progression support. In Figure 7(a), the latency values of both MVAPICH-Async and SmartInterrupts are lower than the baseline. This demonstrates the presence of asynchronous progression. For 18 processes in Figure 7(b), SmartInterrupts continues the trend and exhibits significantly lower message latencies. However, the same cannot be said about MVAPICH-Async as its performance is heavily impaired due to the presence of oversubscription.

5.2.4 Scalability. Figure 8 shows the scalability of MVAPICH-Async and SmartInterrupts. We do this based on two criteria, message size and the number of processes. The graph reports the overlap for each combination of those criteria. As previously discussed, the overlap of relatively smaller Rendezvous messages is expected

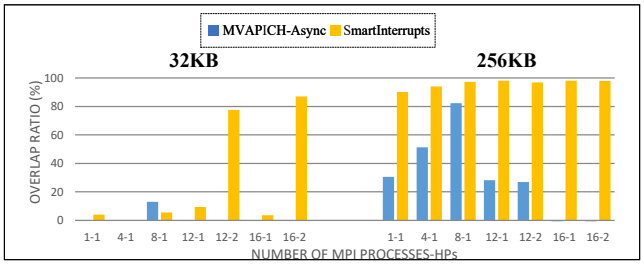


Figure 8: Scalability analysis of SmartInterrupts

to be small. So, the message size of 32KB was chosen to represent them. 256KB messages were chosen to represent larger messages. Here, we report results with 1, 4, 8, 12 and 16 MPI processes per node. For 12 and 16 MPI processes, we also show results with different number of helper processes. It can be observed that SmartInterrupts scales well with both the parameters of message size and number of processes. Also, in case of 12 and 16 MPI processes, 1 helper process is enough to achieve close to 100 percent overlap for messages of size of 256KB but for 32KB messages, a minimum of 2 helper processes are required for overlap in our experimental setup.

5.2.5 Memory Footprint Analysis. The memory footprint of SmartInterrupts was analyzed by looking at the physical memory usage of the processes. Also, to know how the memory usage scales, we measured it by doing incremental changes to the number of MPI processes per node. We observed the memory usage of SmartInterrupts to be slightly greater than MVAPICH, by approximately 300KB per process. This difference remained constant regardless of the number of processes per node.

5.2.6 Application Results. As shown in Figure 9, SmartInterrupts was evaluated with the Scalar Penta-diagonal solver (SP) benchmark [2] using different problem classes and different number of MPI processes under strong scaling, and the results were compared to MVAPICH. Figure 9 shows the results for the problem class E with 324, 400 and 484 MPI processes, distributed evenly across 21, 25 and 31 compute nodes, respectively. Since oversubscription is known to adversely impact the performance of MVAPICH-Async, it was not considered for comparison. Throughout the experiments with SmartInterrupts, two helper processes were used per node. The results show that the proposed approach can indeed make a difference in a practical scenario. The most interesting result is the one with 324 MPI processes, in which it shows an improvement of about 19%. However, SmartInterrupts shows little improvement with 400 and 484 MPI processes. This is because the application scales strongly and the size of the Rendezvous messages is inversely proportional to the number of MPI processes. With 400 and 484 MPI processes, the size of most messages drop below the eager threshold, so they end up being transferred eagerly.

5.3 One-Sided Communication

The evaluation of one-sided SmartInterrupts was based on similar criteria as its two-sided counterpart. For all micro-benchmarks, two types of communication schemes were tested, pair-wise communication and one-to-many communication. In both schemes, all the

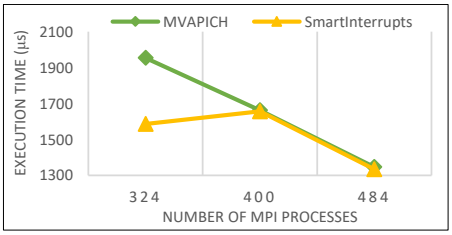


Figure 9: NAS SP results, Class E

MPI processes on one node were designated as the origins and the MPI processes on the other node were designated as the targets. In the pair-wise scheme, each RMA window object is associated with exactly one origin on one node and exactly one target on the other node, and that origin communicates exclusively with its target. In the one-to-many scheme, each window object is associated with one origin on one node and all the targets on the other node. All the origins communicate with all the targets but no two origins share the same window.

The micro-benchmarks were designed with Fence, GATS and exclusive-lock based epochs. These epochs were tested with two different RMA operations, MPI_Put and MPI_Get. For Fence and GATS, the experiments showed similar results for both the targets and the origins. We present the results for the targets only. This is because the target cannot close its epoch until all its origins have closed their own epochs, causing it to suffer the same unproductive waits as its origins.

As mentioned earlier, SmartInterrupts is implemented on top of NewRMA. Therefore, the results of NewRMA was used as the baseline to evaluate SmartInterrupts. SmartInterrupts' asynchronous progression performance was tested against polling based asynchronous progression, which is supported by NewRMA. For convenience, they are referred to in this paper as NewRMA, NewRMA-Async and SmartInterrupts. Casper is available to be used as an extension over existing MPI implementations, however, it could not be experimentally compared with SmartInterrupts because it was found to be incompatible with NewRMA.

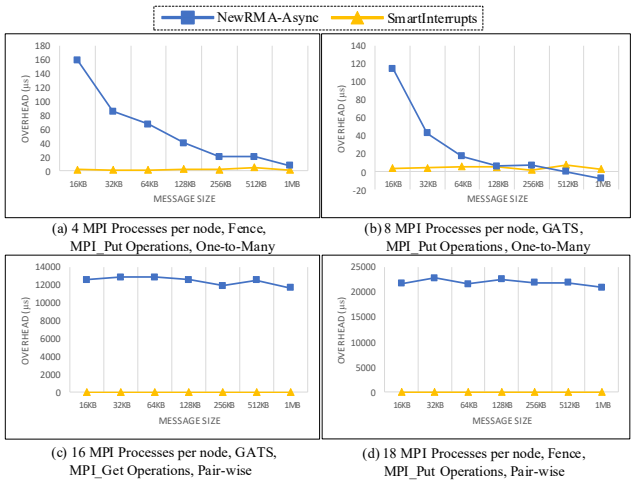
5.3.1 Latency Overhead. Figure 10 shows the algorithms used to measure the latency overhead, using messages in the range of 16KB to 1MB. In the pair-wise communication, the loop at the origin, between lines 4-6, iterates only once because the origin only communicates with one other target. For this and all the other micro-benchmarks, the algorithms were executed 1000 times and results of the first 100 iterations were discarded to account for cache warm-up. The average duration of the next 900 epochs was first recorded locally and then averaged across all the targets.

Since both the asynchronous approaches, SmartInterrupts and NewRMA-Async, work on top of NewRMA, the average epoch durations of NewRMA was chosen as the baseline to measure the latency overheads. The latency overhead is the difference between the epoch durations of an asynchronous message progression approach and NewRMA. Figure 11 shows the latency overheads of SmartInterrupts and NewRMA-Async for different combinations of RMA synchronizations, process counts, and communication schemes. SmartInterrupts shows a negligible amount of overhead in all the

```
ORIGIN:
1.  MPI_Barrier(MPI_COMM_WORLD)
2.  Measure Start Time
3.  MPI_Win_fence/MPI_Win_start
4.  Foreach( target in Targets){
5.      MPI_Put/MPI_Get
6.  }
7.  (Overlap only) Synthetic Work
8.  MPI_Win_fence/MPI_Win_complete
9.  Measure Stop Time
```

```
TARGET:
1.  MPI_Barrier(MPI_COMM_WORLD)
2.  (Overlap only) Small constant
    delay to delay the opening of the
    exposure epoch
3.  Measure Start Time
4.  MPI_Win_fence/MPI_Win_post
5.  MPI_Win_fence/MPI_Win_wait
6.  Measure Stop Time
```

Figure 10: Latency overhead and Fence/GATS overlap micro-benchmark



(a) 4 MPI Processes per node, Fence, MPI_Put Operations, One-to-Many

(b) 8 MPI Processes per node, GATS, MPI_Put Operations, One-to-Many

(c) 16 MPI Processes per node, GATS, MPI_Get Operations, Pair-wise

(d) 18 MPI Processes per node, Fence, MPI_Put Operations, Pair-wise

Figure 11: One-sided latency overhead for Fence and GATS

results. However, the same cannot be said for NewRMA-Async's performance. With 4 or 8 MPI processes per node, NewRMA-Async suffers no oversubscription so its overhead becomes comparable to SmartInterrupts for larger messages. With 16 MPI processes per node, NewRMA-Async suffers oversubscription on at least 12 CPU cores, leading to large overheads.

5.3.2 Communication/Computation Overlap. Unlike two-sided communication, RMA communication necessitates complex synchronizations among multiple peers and may involve multiple RMA messages in the same epoch. This makes it difficult to quantitatively measure the overlap. However, as shown in [22, 29], overlap can be well appreciated through qualitative means. To qualitatively analyze the overlap, the algorithms described in Figure 10 are executed in two phases. In the first phase, the average epoch durations are recorded with no delay at Line 2 of the target and no synthetic work at the origin. In the next phase, the target is delayed at Line 2 and the synthetic work is set to the epoch duration that was recorded for the largest message in phase one. The algorithms are then executed and the average epoch durations are recorded again. Finally, the recorded results are plotted as curves in a graph. If the curve of an approach is below the baseline, then that is an indication of communication/computation overlap. An ideal curve is a straight line with a slope of zero and an intercept that is equal to the synthetic work.

Figure 12 shows the results of the GATS and Fence overlap micro-benchmarks with different combinations of MPI processes

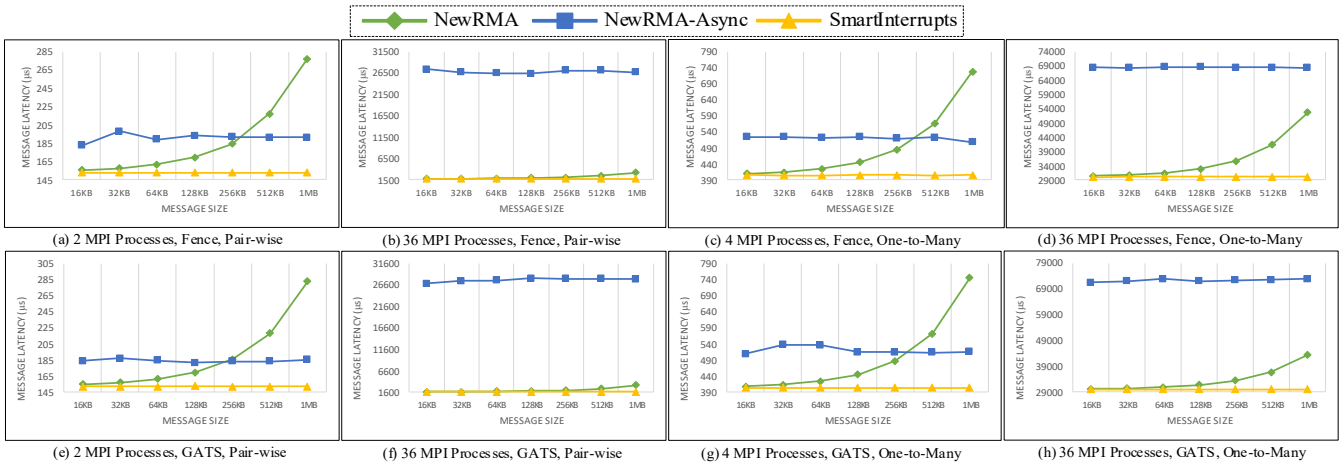


Figure 12: One-sided overlap results for Fence and GATS

and communication schemes. The number of helper processes per node and the type of RMA operation are constant throughout all the results. Space limitations prevent us from presenting all the results; however, these results are a fair representation of the observations. During the experiments, the results with MPI_Put and MPI_Get were found to be identical, hence we only present the results with MPI_Put. Also, it was found that a single helper process per node was enough to achieve the ideal amount of overlap in all scenarios. In fact, the intercepts of SmartInterrupts' lines always remained in the proximity of 0.0001% of their ideal values. The results of NewRMA are in line with expectations. It does not offer any asynchronous progression, so the communications and computations end up being serialized. The results of NewRMA-Async are always worse than that of SmartInterrupts. Overlap is observed when there is no over-subscription but at high process counts, over-subscription dominates and leads to worse latency results than both NewRMA and SmartInterrupts.

To study the overlap in case of exclusive lock epochs, we designed a micro-benchmark requiring only three processes, two of which are designated as the origins. The algorithms executed by Origin0 and Origin1 are described in Figure 13 and the code at the target only consists of an MPI_Barrier. This micro-benchmark simulates a scenario in which Origin1 has been granted the lock, but it cannot progress its messages since it is involved in a computation. The algorithms are executed in two phases. In the first phase, Origin0 is absent, so there is no need to delay Origin1 (line 2). Also, the synthetic work is set to zero and the epoch durations are recorded. In the next phase, Origin0 is activated and Origin1's synthetic work is set to the epoch duration that was recorded for the largest message in phase one. The results from Origin1 are displayed in Figure 14. They are similar to the Fence and GATS overlap results and can be interpreted in the same way.

5.3.3 Memory Footprint Analysis. SmartInterrupts' memory footprint was found to be negligible and hovered around 60 KB per process, for any number of MPI processes.

ORIGIN0:

```
1. MPI_Barrier(MPI_COMM_WORLD)
2. MPI_Win_lock
3. Small delay to ensure that
   Origin1 enters the synthetic work
   without obtaining the lock.
4. MPI_Win_unlock
5. Measure Stop Time
```

ORIGIN1:

```
1. MPI_Barrier(MPI_COMM_WORLD)
2. Small delay to ensure that
   Origin0 gets the lock first
3. Measure Start Time
4. MPI_Win_lock
5. MPI_Put/MPI_Get
6. Synthetic Work
7. MPI_Win_unlock
8. Measure Stop Time
```

Figure 13: Exclusive lock communication/computation overlap micro-benchmark

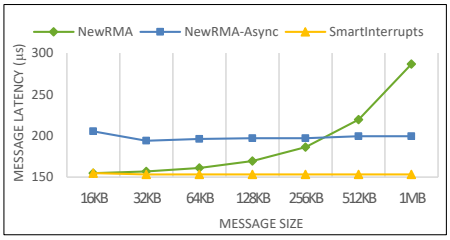


Figure 14: Overlap results for exclusive lock epochs

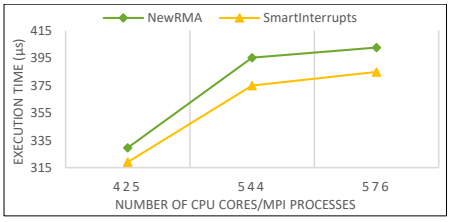


Figure 15: LU results with 1 helper process per node

5.3.4 Application Performance Results. As shown in Figure 15, SmartInterrupts was evaluated with the NAS Lower Upper (LU) decomposition application and compared against NewRMA with strong scaling. From the micro-benchmark results, it is evident that NewRMA-Async does not lead to any performance improvements,

even without oversubscription. Hence, it was not used in evaluating SmartInterrupts' application performance. For our studies, the size of the matrices was kept constant at 32768x32768, and we experimented with 425, 544 and 576 processes, excluding one core per node for the helper process. The results show consistently lower execution times for SmartInterrupts, with performance improvements of 3.1%, 5.1% and 4.4% for 425, 544 and 576 MPI processes, respectively. This suggests that SmartInterrupts can indeed make a difference in a real-world scenario.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose SmartInterrupts, which is a hybrid polling and interrupt based asynchronous message progression technique for both point-to-point and RMA communications for RDMA enabled networks. The progression is performed by interrupt threads which are triggered into action by helper processes when the message matching parameters are met. Each MPI process has its own interrupt thread and several of these interrupt threads may be associated with a single helper process. The micro-benchmark evaluation of both designs was done for the latency overhead, communication/computation overlap, and memory footprint. Additionally, the scalability analysis of the point-to-point design was also performed. The results were compared with those of default MVAPICH2 with no asynchronous progression and with its polling based asynchronous progression. We showed that the overhead incurred by SmartInterrupts is negligible and that it can achieve the ideal amount of overlap in most scenarios. Also, our approach is scalable and casts an insignificant amount of memory footprint. Finally, using applications such as NAS SP and LU decomposition, we showed the effectiveness of SmartInterrupts in practical scenarios.

In future, we would like to extend SmartInterrupts' support to multi-threaded and mixed-mode MPI applications. Although the resource consumption of SmartInterrupts is practically insignificant, we plan to devise a mechanism to dynamically control the number of helper processes, so that the cores occupied by them can be used for other purposes, if certain MPI processes cannot benefit from them.

ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and Canada Foundation for Innovation. We thank the HPC-AI Advisory Council for the computing resources to conduct this research.

REFERENCES

[1] 2021. InfiniBand Trade Association. http://www.infinibandta.org/
[2] 2021. NAS Parallel Benchmarks (NPB). https://www.nas.nasa.gov/publications/npb.html
[3] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. 2010. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 2010 Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. 180–186.
[4] A. Denis. 2014. ioman: a Generic Framework for Asynchronous Progression and Multithreaded Communications. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 276–277.
[5] A. Denis. 2015. pioman: a Pthread-Based Multithreaded Communication Engine. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 155–162.
[6] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos. 2015. The BXI Interconnect architecture. In *IEEE 23rd Annual Symposium on High-Performance Interconnects (HotI)*.
[7] T. Gysi, J. Bär, and T. Hoefler. 2016. dCUDA: Hardware Supported Overlap of Computation and Communication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[8] T. Hoefler and A. Lumsdaine. 2008. Message Progression in Parallel Computing - To Thread or not to Thread?. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster)*. IEEE, 213–222.
[9] W. Jiang, J. Liu, H.-W. Jin, D. Panda, D. Buntinas, R. Thakur, and W. Gropp. 2004. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In *Euro PVM/MPI, Lecture Notes in Computer Science*, Vol. 3241. 68–76.
[10] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. K. Panda. 2008. Lock-Free Asynchronous Rendezvous Design for MPI Point-to- Point Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Vol. 5205 of Lecture Notes in Computer Science. Springer, 185–193.
[11] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. 2004. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of the 2004 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*.
[12] MPI 2021. Message Passing Interface. Retrieved March 20, 2021 from http://www.mpi-forum.org
[13] MPICH 2021. High-Performance Portable MPI. Retrieved February 27, 2021 from http://www.mpich.org
[14] MVAPICH 2021. MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. Retrieved March 24, 2021 from http://mvapich.cse.ohio-state.edu/
[15] OFI 2021. OpenFabrics Alliance. Retrieved April 20, 2021 from https://www.openfabrics.org
[16] OPEN MPI 2021. Open Source High Performance Computing. Retrieved April 10 2021 from https://www.open-mpi.org/
[17] S. Pakin. 2008. Receiver-initiated Message Passing over RDMA Networks. In *Proceedings of the 2008 IEEE International Parallel Distributed Processing Symposium (IPDPS)*. 1–12.
[18] M. J. Rashti and A. Afsahi. 2007. Assessing the Ability of Computation/Communication Overlap and Communication Progress in Modern Interconnects. In *15th Annual IEEE Symposium on High- Performance Interconnects (Hot Interconnects)*. 117–124.
[19] M. J. Rashti and A. Afsahi. 2009. A Speculative and Adaptive MPI Rendezvous Protocol over RDMA-enabled Interconnects. *International Journal of Parallel Programming* 37, 2 (2009), 223–246.
[20] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. Panda. 2018. Efficient Asynchronous Communication Progress for MPI without Dedicated Resources. In *EuroMPI*.
[21] M. Si and P. Balaji. 2017. Process-Based Asynchronous Progress Model for MPI Point-to-Point Communication. In *19th IEEE International Conference on High Performance Computing and Communications (HPCC)*.
[22] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. 2015. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. 665–676.
[23] M. Small and X.Yuan. 2009. Maximizing MPI Point-to-point Communication Performance on RDMA-enabled Clusters with Customized Protocols. In *Proceedings of the 2009 International Conference on Supercomputing (ICS)*. ACM.
[24] S. Sur, H. Jin, L. Chai, and D. K. Panda. 2006. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 32–39.
[25] R. Thakur, W. Gropp, and B. Toonen. 2005. Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication. *International Journal of High Performance Computing Application (IJHPCA)* 19 (2005), 119–128.
[26] A. Wagner, H.-W. Jin, D. Panda, and R. Riesen. 2004. NIC-based Offload of Dynamic User-defined Modules for Myrinet Clusters. In *IEEE International Conference on Cluster Computing (Cluster)*. 205–214.
[27] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. 2013. Asynchronous MPI for the Masses. arXiv:1302.4280.
[28] J. A. Zounmevo and A. Afsahi. 2011. Investigating Scenario-conscious Asynchronous Rendezvous over RDMA. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (Cluster)*. IEEE, 542–546.
[29] J. A. Zounmevo and A. Afsahi. 2014. Intra-Epoch Message Scheduling to Exploit Unused or Residual Overlapping Potential. In *EuroMPI 2014*. 13–19.
[30] J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi. 2014. Nonblocking Epochs in MPI one-sided Communication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 475–486.