Investigating Scenario-conscious Asynchronous Rendezvous over RDMA

Judicael A. Zounmevo and Ahmad Afsahi

Department of Electrical and Computer Engineering, Queen's University

Kingston, ON, Canada

{judicael.zounmevo, ahmad.afsahi}@queensu.ca

Abstract— **In this paper, we propose a light-weight asynchronous message progression mechanism for large message transfers in Message Passing Interface (MPI) Rendezvous protocol that is scenario-conscious and consequently overhead-free in cases where independent message progression naturally happens. Without requiring a dedicated thread, we take advantage of small bursts of CPU to poll for message transfer conditions. The existing application thread is parasitized for the purpose of getting those small bursts of CPU. Our proposed approach is only triggered when the message transfer would otherwise be deferred to the MPI wait call; and it allows for full message progression, achieving 100% overlap. It does not add to the memory footprint of the applications, and is effective in improving the communication performance of most of the applications studied in this paper.**

I. INTRODUCTION

Non-blocking point-to-point operations represent an important performance improvement strategy allowing communication to be overlapped with computation. With the availability of Remote Direct Memory Access (RDMA) technology in modern interconnects such as InfiniBand [1], iWARP [6] and Myrinet [3], independent message progression became considerably easier. Still, the start of data transfer, and consequently the communication completion, may need further library calls even with the presence of RDMA [13].

We establish in general that a sufficient condition for an all-time effective autonomous communication progress requires three mechanisms: 1) A *carrousel* mechanism, meant to ferry messages from one node to another without any external propelling force. RDMA, by itself, only represents the carousel; 2) A *watchdog* mechanism to check the availability of messages or transfer conditions at either end of the carrousel; and 3) A *trigger* mechanism to kick off the carrousel. This includes dropping/picking up messages on/from the carrousel if applicable.

Message progression improvement efforts can be classified in two categories: 1) protocol improvements, and 2) the use of asynchronous communication progress [9]. Protocol improvement approaches, the redesign or rearrangement of control messages, fall short of covering all scenarios. As for the asynchronous communication progress, it happens when the triggering of the message transfer is not provoked by the application thread that issued the communication. The simplest and only known embodiment of this approach so far is by means of a dedicated thread. That thread can either poll or wait on an interrupt.

In this work we study the feasibility of approaching asynchronous message progression without resorting to a thread. In situations where a CPU core cannot be dedicated to the thread for polling, we seek to provide to the interrupt-based approach an alternative that is overhead-free when no improvement is possible. We seek to parasitize the existing application thread with the watchdog so as to periodically poll for MPI [2] Rendezvous message transfer conditions. We have implemented the proposed approach in MVAPICH, as it possesses an asynchronous Rendezvous implementation [10] that we intend to compare our proposal with. The micro-benchmark results confirm that our proposal incurs no or less overhead compared to the interrupt-based threading approach, and is on par on message progression and communication-computation overlapping. The application results show that our method is substantially lighter on memory footprint. They also show that our proposed approach is more adequate in dealing with the communication wait time and performance for applications that use blocking calls and most of the applications that use non-blocking calls.

The rest of this document is organized as follows. Section 2 presents the related work and builds the motivations behind this proposal. In Section 3, we describe our design objectives and implementation on Linux. Section 4 presents our experimental evaluations. Section 5 concludes and gives future directions.

II. RELATED WORK AND MOTIVATIONS

The work in [14] falls in the category of protocol improvement, and so do [11][12]. The main hindrance for overlapping in presence of RDMA is the lack of timely fulfilment of the trigger. Protocol improvement proposals try to fix this situation by making sure that the trigger is activated before any computation starts. They do not provide a watchdog; they try to do without instead. They do so by either redefining the message transfer initiator (receiver vs. sender), redesigning the set of handshaking control messages, or by doing both. Protocol improvement approaches are unable to improve message progression for

messages bearing MPI_ANY_SOURCE when the Rendezvous is receiver-initiated [12][14]. They are also usually more complex and consequently more synchronizing than the default protocols [11][12]. This last drawback is exacerbated especially when the protocol contains a mix of sender-initiated and receiver-initiated sub-protocols [12][14]. Small et al. [15] propose a profile-driven mechanism to select the best protocols based on the relative timing of the calls of send/receive/wait routines. However, prior profiling is hardly a guarantee because the same relative timings usually do not hold across executions due to various nondeterministic host events and system noise.

Polling by means of a thread is characterized by a fast response time but it is viable only when spare CPU cores are available; implying that the parallelism level available to the application must be halved [9]. The use of interrupts [8][16] [10] notifies a waiting helper thread for message arrival. The usual criticism directed to interrupts relates to their relatively high latency. With InfiniBand and OFED [5], there is also the impossibility of selectively generating the interrupts only in cases where the message progression would not naturally happen. There are three possible receive scenarios, namely, 1) blocking receive; 2) non-blocking receive with the *Ready To Send* (RTS) [12] control message coming before the receive call is issued; and 3) non-blocking receive with RTS coming after the receive call exits. Scenario 3 is the only case where the progression thread is required in order to prevent the message transfer from being deferred to the wait call. The interrupt happens at the receive-side but it is triggered from the send-side by setting a flag in the header of the RTS control message. Since the sender can predict neither the arrival time of the receiver nor the blocking-nature of the receive call, it always sets the interrupt flag. As a result, the receive-side progression thread gets waken up for every receive operation. Triggering the progress thread equals incurring 1) an interrupt cost, 2) a context switch when the thread wakes up, and 3) either a lock to get into the progress engine, or another interrupt(signal) to the application thread to execute the progress engine [10]. That succession of events associated with each waking of the progression thread is too expensive to allow them to happen in situations where they deterministically have no chance of producing any improvement. Furthermore, an x86_64 thread adds a fixed 10MB to the resident set size of Linux applications, making the helper thread method a bit inefficient with respect to memory footprint.

III. Design objectives and Implementation

A. Design Objectives

The watchdog can only be polling, otherwise the application thread that we seek to parasitize would have to block; preventing any possibility of overlapped computation and communication. Furthermore, the task of checking control message arrivals is easier with RDMA Read. With its back-and-forth control message scheme, RDMA Write would require the watchdog to operate at both sides for each Rendezvous.

The RDMA Read-based Rendezvous protocol is already known to allow independent progress and overlapping when the RTS message reaches the receiver before MPI_Irecv is issued [12][15][16]. However, when the receiver does not see the RTS message, the overlapping opportunity is missed. The application thread must then trigger the message transfer in the wait call after the completion of any ongoing computation. We designate by *Application Execution Flow* (AEF) the flow made of the default instructions of the application thread; and by *Parasite Execution Flow* (PEF) the flow made of the instructions we are sneaking into the existing application thread at the middleware-level. Before any MPI_Irecv is issued, the thread only has AEF. If it was not active, PEF is spawned by the next MPI_Irecv that misses its RTS. PEF then periodically executes the watchdog until RTS is found and the RDMA Read transfer started (in Fig. 1). Any subsequent MPI_Irecv that executes before PEF dies just adds a progress request to the queue of currently watched receive requests. PEF dies when the last expected RTS is found or when any of the wait family of routines is called for the last pending request. There is only a single PEF in the application thread no matter the number of pending non-blocking receives. For a single pending Irecv, PEF is active for at most the duration of the possible overlapping period. The overlapping period ends when any of the wait family of routines is called for the receive request. It also ends when the transfer completes; even if wait is not yet called.

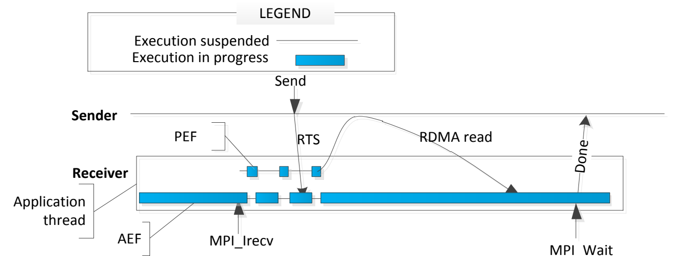

Fig. 1: Parasite execution flow-based message progression at receiver-side for point-to-point communications

B. Design Realization

An asynchronous means is required to disrupt the flow of AEF at a desired frequency. The use of timers appears to be an immediate candidate. The timer would periodically preempt AEF to transfer the CPU to PEF. Then, when done, PEF willingly retransfers the CPU to AEF. A SIGALRM-based [7] signal delivery can be singled out as a potential candidate to fulfil both the goals of disruption and periodic polling event.

The PEF to AEF transition bears no issue because it is not preemptive. However, the preemptive nature of the AEF to PEF transition can raise data access concerns but we still

manage to keep the disruption mechanism lock-free. First, there is no programmer-accessible data access issue because PEF is disabled when AEF, i.e., the thread itself, is about to call the progress engine, which is the only critical section of interest. There is therefore no risk of AEF being preempted when inside the progress engine. A similar mechanism is already used in the thread-helped Rendezvous of MVAPICH. As for non-programmer-accessible data access issues, as far as Linux signals are concerned, they relate to the internal buffers used by the C standard I/O functions as well as data manipulated by dynamic memory allocations. However, there is no reasonable motive why dynamic memory allocation would be required in the progress engine. Plus, even if stream functions such as printf are needed in the progress engine for tracing purpose, they can be deferred by resorting to buffering while inside PEF and then flushing while outside.

We provide the following environment variables to control the AEF/PEF alternations. A period *ppef_period* to specify the time between two consecutive turn taking of PEF; a phase *ppef_phase* to specify the delay before PEF takes turn the first time after a new request is queued; a frequency decay *ppef_freq_decay* to specify a multiplicative factor of the period after each turn taking; a turn limit *ppef_max_turns_per_req* to specify the maximum number of turn taking after which PEF gives up progressing the most recent request. The tuning space defined by the afore-mentioned parameters is very large. Without a prior knowledge of the application at hand, tuning is thus a difficult task, and the subject of our future study. As a result, we propose a two-step rule of thumb that serves as the default set of parameters. The first step is optimistic. It assumes that the application is well-behaved; meaning that the peers are balanced enough to exhibit only small delays that a small value of the phase (e.g. 2μs or 5μs at most) should help servicing. If RTS does not arrive in the time frame of the phase, the rule enters its second step where it becomes more and more pessimistic after each PEF turn that does not hit. The pessimistic step is realized by a period that is multiplied by a frequency decay after each turn. The pessimistic step gives up a bit of reactiveness each time in order to limit the overhead imposed on the receiver by very late senders. On our test system, the rule uses a phase of 2μs, a period of 10μs and a frequency decay of 2. This rule generates a PEF turn taking time sequence that is reset every time a new request is posted.

IV. Experimental Evaluation

Our experimental setup is a four-node InfiniBand cluster. Each node is equipped with two quad-core 2GHz AMD Opteron 2350 processors and 8GB of RAM. The network devices are Mellanox ConnectX QDR cards and switches. The Eager/Rendezvous protocol threshold is 9180 bytes. The tests compare the MVAPICH-ASYNC method, which implements the interrupt-thread approach, with our proposed PEF method. RGET designates the default RDMA Read-based Rendezvous in MVAPICH.

A. Micro-benchmark Results

Both the interrupt-thread and PEF methods operate exclusively at receive-side and over RDMA Read. The micro-benchmarks directly focus on receiver-side results because it is already established that message progression naturally occurs at sender-side with RDMA Read [10]. Fig. 2 shows the latency overhead of MVAPICH-ASYNC and PEF compared to RGET. Barriers are used to force arrival orders for non-blocking tests. The scenarios “blocking receive” (Fig. 2-a) and “non-blocking receive, sender arrives first” (Fig. 2-b) show that the overhead of PEF is on average 0μs while MVAPICH-ASYNC exhibits on average 10μs and 20μs per message transfer respectively. The scenario-consciousness of PEF justifies its absence of overhead in these scenarios where MVAPICH-ASYNC is triggered even though there is no need and no room for improvement. When the receiver is non-blocking and early compared to RTS (Fig. 2-c), a progression help is required to avoid deferring the message transfer to the wait call. Fig. 2-c shows that while PEF exhibits an overhead of approximately 5μs MVAPICH-ASYNC exhibits on average 12μs.

Due to space limitations, the message progression and communication-computation overlapping graphs (Fig. 3) show only the results of interest; i.e., the receiver-side performances when RTS is late. We show message progression for two message sizes, 64KB and 512 KB in Fig. 3-a and Fig. 3-b respectively. We insert a synthetic computation between MPI_Irecv and MPI_Wait to see if the communication latency grows proportionally to the computation length. The computation length is increased from 50μs to 1000μs by 50μs increments. On all the graphs in Fig. 3-a and Fig. 3-b, both mechanisms keep the latency approximately constant and independent from the inserted computation; meaning that they all allow full message progression. Since RDMA Read by itself is not progressing when the receiver comes first, the latency associated with RGET grows endlessly with the computation length. In order to avoid dwarfing the other constant curves, RGET is not presented in Fig. 3-a and Fig. 3-b.

We use the communication-computation overlapping algorithm described in [13] to generate the overlapping performances depicted in Fig. 3-c. We show MVAPICH-RGET in this graph because it does not hamper the readability of the other curves. The figure shows that MVAPICH-RGET yields no overlapping. PEF and MVAPICH-ASYNC both yield close to 100% overlapping. The overlapping formula is more accurate for larger messages for which the transfer duration dominates the communication time. This inaccuracy explains why MVAPICH-RGET shows nonzero overlapping for 16KB and 32KB and why the PEF and thread curves are not superimposed for those messages.

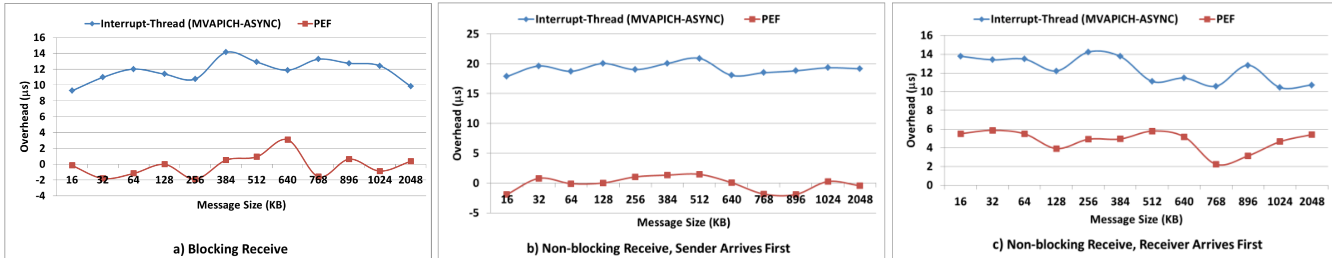

Fig. 2: Receiver-side latency overhead of interrupt-based threading (MVAPICH-ASYNC) vs. PEF. (MVAPICH-RGET is the reference)

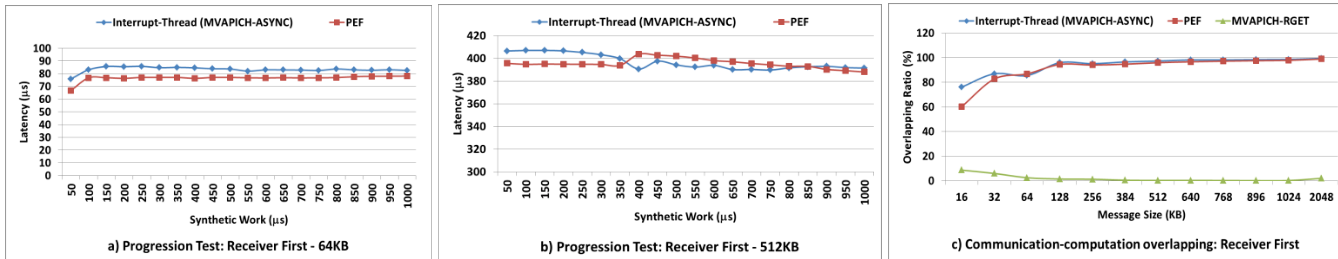

Fig. 3: Receive-side message progression and communication-computation overlapping

B. Application-Level Evaluation

For our tests purpose, it is not easy to force an arrival order-based scenario in applications. Nevertheless, we can present the impact of both PEF and the interrupt-thread mechanism on blocking receives. The results, using the NAS [4] LU application is shown in Fig. 4. The name of the program run for each test is specified with the class and the number of ranks appended in that order to the application name. The results are shown for 4 and 32 ranks, respectively to mimic the case where there is no shared-memory communication and the case where the system is loaded with the maximum possible number of processes. NAS LU, because it is based on blocking receives, needs no improvement from any of the mechanisms. The interrupt-thread (MVAPICH-ASYNC) mechanism degrades the communication by more than 8% for 4 ranks and by more than 22% for 32 ranks. In comparison, PEF shows a degradation of less than 0.2% and 0.9% for 4 and 32 ranks respectively. Since PEF is actually not triggered at all in this scenario, those tiny percentages are more tributary to noise than actual penalty. From the trend observed in Fig. 4, we can conjecture that the interrupt-thread mechanism would deal larger overheads to larger jobs; exacerbating the issue at larger scales.

Fig. 5-a and Fig. 5-b show the wait and communication improvements respectively for NAS BT, CG, MG and SP. BT and SP can only be executed over square numbers of processes; and can reach a maximum of 25 ranks on our 32-core cluster. We first observe that both mechanisms yield similar performances for some of the applications; namely, BT.B.4, CG.B.4, CG.C.32, SP.B.4 and SP.C.25. We also observe that for BT.C.25, the thread shows better results while PEF performs better for MG.B.4 and MG.C.32. Once again, the key observation remains the absence of a general trend of PEF paying its scenario-consciousness by underperforming compared to the interrupt-thread mechanism.

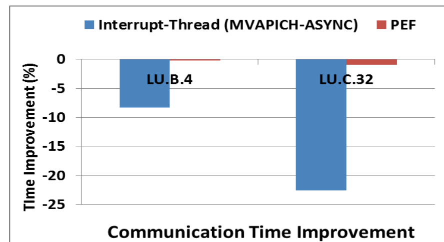

Fig. 4: Effect of each mechanism on a blocking receive-based application

In general, it is important to remind that the actual non-blocking communication progression is handled by RDMA. The two means being compared here just trigger the transfer. As a consequence, when the RTS lateness is reasonable, non-blocking communication progression performance while RDMA is available tends to be a two-state variable swinging between a maximum possible performance and zero. This explains well the behaviour observed for BT.B.4, CG and SP in Fig. 5-a and Fig. 5-b. However, application behaviours can sometimes create some differences in the performances as shown by BT.C.25, MG.B.4 and MG.C.32. If RTS arrives mostly early, MVAPICH-ASYNC would tend to perform poorly compared to PEF. PEF on the other hand would tend to be outperformed by MVAPICH-ASYNC when RTS mostly exhibits slightly large delays. In these cases, PEF tend to be less reactive than MVAPICH-ASYNC. Note that very large delays are a performance killer, no matter the Rendezvous mechanism used, as they end up voiding the reactiveness advantage as well. In

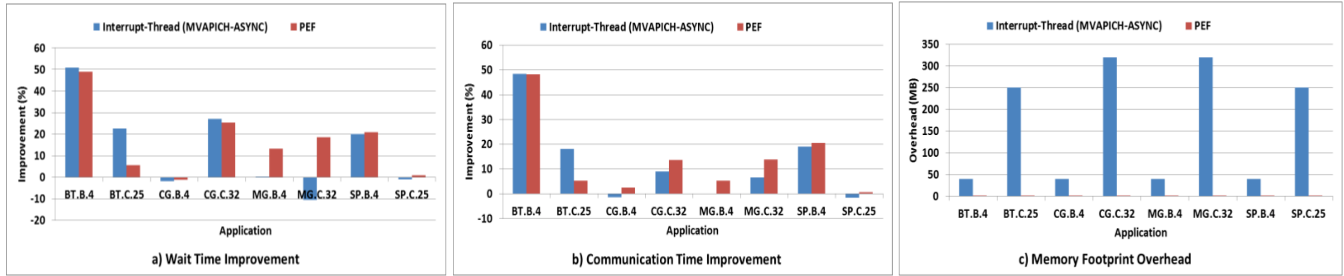

Fig. 5: Application Benchmarks for non-blocking receives

particular, very large delays, when they exist, tend to be the dominant communication latency component; making the actual transfer time insignificant.

Finally Fig. 5-c shows the memory footprint overhead of each mechanism. The overhead is how much each mechanism adds to the resident set size of each application when compared to MVAPICH-RGET. For all the applications, PEF shows an insignificant overhead, reaching 128KB in the worst case for 32 ranks. In comparison, the interrupt-thread mechanism shows more than 320MB in the worst case for 32 ranks. As expected on any x86_64 Linux platform, Fig. 5-c shows an overhead of approximately 10MB per rank for the interrupt-thread mechanism. Once again PEF seems to be the better mechanism when scalability is a concern.

V. Conclusion And Future Work

We realize our proposal by having a Parasite Execution Flow coexisting with the Application Execution Flow in the same thread. PEF hosts the control message polling in a SIGALRM-based signal handler. While the interrupt-thread mechanism of asynchronous Rendezvous is triggered from the sender-side; PEF is entirely controlled by the receiver itself. For having a precise knowledge of the need for progression help, our mechanism can thus be triggered only when it is required. As a consequence and unlike its interrupt-thread counterpart, every overhead that the PEF mechanism imposes to the application is justifiable. Plus, PEF proves to be substantially lighter than the interrupt-thread mechanism for memory footprint overhead. On top of allowing such a control over the overheads and the memory footprint it adds to the HPC jobs, the PEF proposal proves to be on average as efficient as the interrupt-thread mechanism in terms of performance improvement. To the best of our knowledge, this proposal is also the first asynchronous message progression approach that requires neither a specialized hardware, nor a dedicated separate thread. For future work, we intend to replace the signal delivery mechanism by a special-purpose disruption mechanism by altering the operating system. We also seek to test the impact of our proposal on larger systems.

Acknowledgment

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation and Ontario Innovation Trust. We would like to thank Mellanox Technologies for the resources.

References

[1] InfiniBand Trade Association, http://www.infinibandta.org/index.php.
[2] MPI Forum, http://www.mpi-forum.org/.
[3] Myrinet, http://www.myri.com/.
[4] NAS Parallel Benchmarks, http://www.nas.nasa.gov/resources/software/npb.html.
[5] OpenFabrics Alliance, http://www.openfabrics.org/.
[6] RDMA Consortium, http://www.rdmaconsortium.org.
[7] The GNU C Library Reference Manual, http://www.gnu.org/software/libc/manual/.
[8] G. Amerson and A. Apon, "Implementation and design analysis of a network messaging module using virtual interface architecture," *Proceedings of the 2004 IEEE International Conference on Cluster Computing,* 2004, pp. 255-265.
[9] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?," *Proceedings of the 2008 IEEE International Conference on Cluster Computing,* 2008, pp. 213-222.
[10] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman and D. K. Panda, "Lock-free asynchronous Rendezvous design for MPI point-to-point communication," *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* Dublin, Ireland, 2008, pp. 185-193.
[11] S. Pakin, "Receiver-initiated message passing over RDMA networks," *Proceedings of the 2008 IEEE International Parallel & Distributed Processing Symposium,* 2008.
[12] M. J. Rashti and A. Afsahi, "A speculative and adaptive MPI Rendezvous protocol over RDMA-enabled interconnects," *International Journal of Parallel Programming,* vol. 37, 2009, pp. 223-246.
[13] M. J. Rashti and A. Afsahi, "Assessing the ability of computation/communication overlap and communication progress in modern interconnects," *Proceedings of the 2007 Annual IEEE Symposium on High-Performance Interconnects,* 2007, pp. 117-124.
[14] M. Small and X. Yuan, "Maximizing MPI point-to-point communication performance on RDMA-enabled clusters with customized protocols," *Proceedings of the 2009 International Conference on Supercomputing,* Yorktown Heights, NY, USA, 2009, pp. 306-315.
[15] M. Small, Z. Gu and X. Yuan, "Near-optimal Rendezvous protocols for RDMA-enabled clusters," *International Conference on Parallel Processing,* 2010, pp. 644-652.
[16] S. Sur, H. Jin, L. Chai and D. K. Panda, "RDMA read based Rendezvous protocol for MPI over InfiniBand: design alternatives and benefits," *Proceedings of the 2006 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* New York, New York, USA, 2006, pp. 32-39.