On the Suitability of MPI as a PGAS Runtime

Jeff Daily Abhinav Vishnu Bruce Palmer Hubertus van Dam Darren Kerbyson
Pacific Northwest National Laboratory,
902 Battelle Blvd, Richland, WA 99352, USA
Email: {Jeff.Daily, Abhinav.Vishnu, Bruce.Palmer, HubertusJJ.vanDam, Darren.Kerbyson}@pnnl.gov

***Abstract*—Partitioned Global Address Space (PGAS) models are emerging as a popular alternative to MPI models for designing scalable applications. At the same time, MPI remains a ubiquitous communication subsystem due to its standardization, high performance, and availability on leading platforms. In this paper, we explore the suitability of using MPI as a scalable PGAS communication subsystem. We focus on the Remote Memory Access (RMA) communication in PGAS models which typically includes *get, put,* and *atomic memory operations*. We perform an in-depth exploration of design alternatives based on MPI. These alternatives include using a semantically-matching interface such as MPI-RMA, as well as not-so-intuitive interfaces such as MPI two-sided with a combination of multi-threading and dynamic process management. With an in-depth exploration of these alternatives and their shortcomings, we propose a novel design which is facilitated by the data-centric view in PGAS models. This design leverages a combination of highly tuned MPI two-sided semantics and an automatic, user-transparent split of MPI communicators to provide asynchronous progress. We implement the asynchronous progress ranks approach and other approaches within the Communication Runtime for Exascale which is a communication subsystem for Global Arrays. Our performance evaluation spans pure communication benchmarks, graph community detection and sparse matrix-vector multiplication kernels, and a computational chemistry application. The utility of our proposed PR-based approach is demonstrated by a 2.17x speed-up on 1008 processors over the other MPI-based designs.**

I. Introduction

Partitioned Global Address Space (PGAS) models such as Global Arrays [1], Unified Parallel C (UPC) [2], X10 [3] and Chapel [4] provide productive abstractions and high performance implementations of distributed data structures on modern high-end systems. As a result, PGAS models are becoming popular alternatives to traditional Communicating Sequential Processes execution models like the Message Passing Interface (MPI) [5], [6]. However, MPI is ubiquitous due to its high performance, standardization, and portability. The MPI standard has evolved to incorporate Remote Memory Access (RMA) operations, multi-threading support, non-blocking and sparse collective communication primitives, dynamic process management, and derived data types. The MPI specification matches well with the requirements of higher level solver libraries, scalable and productive PGAS programming models, and designing scalable applications directly.

The communication subsystems of PGAS models such as the Communication Runtime for Exascale (ComEx) [7], [8] and GASNet [2] primarily rely on network primitives to achieve the best possible performance. These communication subsystems have native design and implementations on many modern networks such as Cray Gemini [8], IBM Blue Gene/Q [7] and commodity clusters based on InfiniBand/Ethernet [9]. However, a native implementation of these communication subsystems is not always feasible. For example, the device layer below MPI may not be available for direct use by other libraries. This is the case for the communications interfaces of the K-Computer [10] and Tianhe-1A [11] supercomputers. In addition, early access (or even any access at all) to many of these systems is difficult and only available near the system acceptance period. This exacerbates the situation for many scientific applications which rely on these PGAS models [12], [13] and need a high performance implementation as soon as the system is production ready.

Most system acceptance specifications require MPI to be well tested and tuned for performance on many scientific applications. MPI send/receive (two-sided) semantics and collective communication primitives are heavily optimized with special hardware acceleration and low latency communication paths. Hence, it is natural to consider MPI two-sided primitives to be the optimal choice for designing PGAS communication subsystems. However, two-sided models require *implicit synchronization* which is a semantic mismatch with PGAS models. At the same time, there are other alternatives such as MPI-RMA which match semantically very well with the PGAS models, but suffer from severe performance degradation due to suboptimal implementations on high-end systems [14]. This leads to our problem statement. *What is the best way to design a PGAS communication subsystem given that MPI is our only choice?*

Specifically, this paper makes the following contributions:

- An in-depth analysis of design alternatives for a PGAS communication subsystem using MPI. We present four design alternatives: MPI-RMA (RMA), MPI Two-Sided (TS), TS with Multi-threading (MT), and TS with Dynamic Process Management (DPM).
- A novel approach using a combination of two-sided semantics and an automatic, user-transparent split of MPI communicators to act as asynchronous progress ranks (PR) for designing scalable and fast communication protocols.
- Implementation of TS, MT, and PR approaches and their integration with ComEx, the communication runtime for Global Arrays. We perform an in-depth evaluation on a spectrum of software including communication benchmarks, application kernels, and a full application, NWChem [12].

Our performance evaluation reveals that the proposed PR approach outperforms each of the other MPI approaches. We achieve a speedup of 2.17x on NWChem, 1.31x on graph community detection, and 1.14x on sparse matrix-vector multiply using up to 2K processes on two high-end systems.

This work has demonstrated that highly-tuned two-sided semantics are sufficient for implementing one-sided semantics in the absence of a native implementation. This result should continue to affirm system procurement requirements of optimized two-sided communication while suggesting that one-sided communication can be readily improved in the future using the existing MPI interface based on our proposed approach.

The rest of the paper is organized as follows: In section II, we present some background for our work. In section III, we present various alternatives when using MPI for designing ComEx. In section IV, we present our proposed design and present a performance evaluation in section V. We present related work in section VI and conclude in section VII.

II. Background

In this section, we introduce the various features of MPI and ComEx which influence our design decisions.

A. Message Passing Interface

MPI [5], [6] is a programming model which provides a Communicating Sequential Processes execution model with send/receive semantics and a Bulk Synchronous Parallel model with MPI-RMA. In addition, MPI provides a rich set of collective communication primitives, derived data types and multi-threading. In this section, we briefly present relevant parts of the MPI specification.

1) MPI Two-Sided Semantics: Send/receive and collective communication are the most commonly used primitives in designing parallel applications and higher level libraries. The two-sided semantics require an implicit synchronization between sender and receiver where the messages are matched using a combination of tag (message identifier) and communicator (group of processes). MPI allows a receiver to specify a wildcard tag (allowing it to receive a message with any tag) and a wildcard source (allowing it to receive a message from any source).

The send/receive primitives typically use *eager* and *rendezvous* protocols for transferring small and large messages, respectively. For high performance interconnects such as InfiniBand [9], Cray Gemini [8] and Blue Gene/Q [7], the eager protocol involves a copy by both the sender and the receiver, while large messages use a zero-copy mechanism such as Remote Direct Memory Access (RDMA).

2) MPI-RMA: MPI-RMA provides interfaces for *get, put,* and *atomic memory operations*. MPI-RMA 3.0 allows for explicit request handles, for request window memory to be allocated by the underlying runtime, and for windows that allocate shared memory. MPI-RMA provides multiple synchronization modes: *active target*, where the RMA source and target window owners participate in the synchronization and *passive target*, where only the initiator of an RMA operation is involved in synchronization. The availability of generic RMA operations and synchronization mechanisms makes MPI-RMA useful for designing PGAS communication subsystems. However, there are no known implementations of MPI-RMA 3.0 on high end systems (a reference implementation of MPI-RMA 3.0 within MPICH is available). The implementations of previous MPI specifications (such as MPI 2.0) are available. However, they perform poorly in comparison to native implementations as shown by Dinan *et al.* [14].

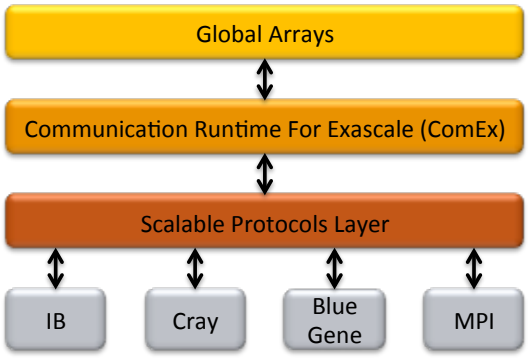

Fig. 1. Software Ecosystem of Global Arrays and ComEx. Native implementations are available for Cray, IBM, and IB systems, but not for Kcomputer and Tianhe-1A.

3) Multi-Threading: One of the most important features of MPI is supporting multi-threaded communication. MPI supports multiple thread levels (single, funneled, serialized, and multiple). The multiple mode is least restrictive and it allows an arbitrary number of threads to make MPI calls simultaneously. In general, multiple is the most commonly used threaded model in MPI. In the design section, we explore the possibility of using thread multiple mode as an option for PGAS communication.

B. Communication Runtime for Exascale (ComEx)

ComEx is a successor of the Aggregate Remote Memory Copy Interface (ARMCI) [15]. ComEx uses native interfaces for facilitating one-sided communication primitives in Global Arrays. As an example, ComEx has been designed to use Openfabrics Verbs (OFA) for InfiniBand [9] and RoCE Interconnects, Distributed Memory Applications (DMAPP) for Cray Gemini Interconnect [8], [16], and PAMI for x86, PERCS, and Blue Gene/Q Interconnects [7]. The specification is being extended to support multi-threading, group aware communication, non-cache-coherent architectures and generic active messages. ComEx provides abstractions for RMA operations such as get, put and atomic memory operations and provides location consistency [17]. Figure 1 shows the software ecosystem of Global Arrays and ComEx.

III. Exploration Space

In this section, we present a thorough exploration of design alternatives for using MPI as a communication runtime for PGAS models. We first suggest the semantically matching

choice of using MPI-RMA before considering the use of two-sided protocols. While considering two-sided protocols, the limitations of each approach are discussed which motivate more complex approaches. For the rest of the paper, the MPI two-sided and send/receive semantics are used interchangeably.

A. First Design: MPI-RMA

MPI-RMA 2.0 supports the one-sided operations *get, put* and *atomic memory operations* in addition to supporting *active and passive* synchronization modes. MPI-RMA 3.0 has several features which facilitate the design and implementation of scalable communication runtime systems for PGAS models. It allows non-blocking RMA requests, request-based transfers, window-based memory allocation, data type communication, and multiple synchronization modes. MPI-RMA 3.0 is semantically complete and suitable for designing scalable PGAS communication subsystems.

MPI-RMA has completely different semantics than the popularly used send/receive and collective communication interface. An important implication is that an optimal design of MPI-RMA needs a completely different approach than two-sided semantics. Since MPI-RMA has achieved low acceptance in comparison to two-sided semantics [14], most vendors choose to only provide a compatibility port due to resource limitations. At the same time, PGAS communication runtimes such as ComEx and GASNet [2] are tailored to serve the needs of their respective PGAS models. As an example, UPC [2] and Co-Array Fortran [18] need active messages for linked data structures which are not well supported by MPI-RMA. Similarly, Global Futures [19] - an extension of Global Arrays to perform locality driven execution - needs active messages for scaling and minimizing data movement.

MPI-RMA can be implemented either using native communication interfaces which leverage RDMA offloading, or by utilizing an accelerating asynchronous RMA thread in conjunction with send/receive semantics. Either of these cases require significant effort for scalable design and implementation. Dinan *et al.* have presented an in-depth performance evaluation of Global Arrays using MPI-RMA [14]. Specifically, Dinan reports that MPI-RMA implementations perform 40-50% worse than comparable native ports on Blue Gene/P, Cray XT5 and InfiniBand with NWChem. This observation implies that vendors are not providing optimal implementations on high-end systems. Unfortunately, although MPI-RMA is semantically complete as a backend for PGAS models, sub-optimal implementations require us to consider alternative MPI features for designing PGAS communication subsystems.

B. Second Design: MPI Send/Receive

MPI two-sided semantics are widely used in most parallel applications. These include point-to-point and collective communication. Their nearly ubiquitous use implies that these semantics are heavily optimized for a variety of scientific codes and co-designed with hardware for best performance. Hence, it is natural to consider two-sided semantics for designing scalable PGAS communication subsystems.

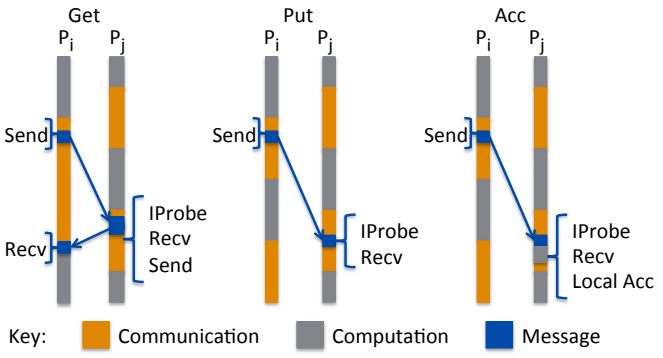

Fig. 2. One-Sided Communication Protocols using Two-Sided Communication Protocols in MPI. Protocols for Get, Put and Accumulate are on the left, middle, and right, respectively.

A possible design of ComEx using MPI send/receive semantics can be done by carefully optimizing RMA operations using MPI two-sided semantics. In this design, every process must service requests for data while at the same time performing computation and initiating communication requests on behalf of the calling process. As a result, this design is never allowed to make synchronous requests; all operations must be non-blocking. Otherwise, deadlock is inevitable. Furthermore, synchronization barriers and collective operations must also be non-blocking to facilitate progress while servicing requests. ComEx does not provide an explicit progress function, so progress can only be made when any other ComEx function is called. We consider design issues such as the above while mapping one-sided semantics onto two-sided semantics in the following sections.

1) Put/Accumulate Operations: In PGAS models like Global Arrays, blocking and non-blocking Put operations can be designed using MPI_Send and MPI_Isend primitives, respectively, issued from the source process. Due to the implicitly synchronous semantics of send/receive, the destination process must at some point initiate a receive in order to complete the operation. In the case of accumulate, the destination process must also perform a local accumulate after receiving the data. Figure 2 illustrates this design.

2) Get Operations: An MPI get operation can be designed as a *request to get + receive* operation at the initiator. The source of the get (the remote process which owns the memory from where the data is to be read) participates in the get operation implicitly by servicing the get request. A possible implementation would use a combination of MPI *probe, receive* and *send* in that order. Figure 2 illustrates this design.

3) Other Atomic Memory Operations: Atomic Memory Operations (AMOs) such as *fetch-and-add* are critical operations in scaling applications such as NWChem [12]. The AMOs are used, for example, for load balancing in these applications. AMOs can be implemented by a simple extension of the Get operation: In addition to servicing a get request, the remote process also performs an atomic operation on behalf of the

initiator. An additional MPI_Send needs to be initiated by the host of the target to provide the original value before the increment. The accumulate operations do not need to return the original value to the initiator.

4) Synchronization: ComEx supports location consistency with an *active mode* of synchronization [17]. A ComEx barrier is both a communication barrier (fence) as well as a control barrier e.g. MPI_Barrier. This can be achieved by using pair-wise send/receive semantics. Each process can exit a synchronization phase as soon as it has received the *termination* messages from every other process. While synchronizing, all other external requests are also serviced. It is important to note that each process needs to receive a termination message from every other process. A collective operation such as barrier/allreduce cannot be used for memory synchronization, since it does not provide pair-wise causality.

5) Collective Operations: ComEx does not attempt to reimplement the already highly-optimized MPI collective operations such as all reduce. However, since this design requires all operations to be non-blocking, entering into a synchronous collective operation would certainly cause deadlock. The two-sided design must then perform a collective communication fence in addition to a control barrier prior to entering an MPI collective.

6) Location Consistency: The location consistency semantics required in ComEx can be achieved by using the buffer reuse semantics of MPI - invoking a wait on a request handle can provide similar re-use semantics to ComEx as MPI. In addition, messages are ordered between all process rank pairings by using the same MPI tag for all communications, implicitly guaranteeing that a series of operations on the same area of remote memory are executed in the same order as initiated by a given process. Location consistency can be guaranteed in conjunction with the exclusively non-blocking requirement of this design by queuing requests and only testing the head of the queue for completion before servicing the next item in the queue.

7) Primary Issue: Communication Progress: The primary problem with MPI two-sided is the general need for communication progress for all operations, but especially for Get and FetchAndAdd primitives. PGAS models are frequently combined with non-SPMD execution models for load balancing and work stealing. In NWChem and certain graph algorithms, it is too prohibitive to predict the computation time of each task. Hence, it is important to provide a mechanism for asynchronous progress in addition to using MPI two-sided semantics.

For large put and accumulate messages requiring a rendezvous protocol, the sending process will not complete the transfer until the target process has initiated a receive. Unfortunately, the target process cannot make progress on requests unless it also has called into the ComEx library having made a request of its own. The performance of a compute-intensive large-message application would certainly degrade using this design, unless asynchronous progress could be made.

There are two main choices to facilitate communication progress: multi-threading and dynamic process management. In the next section, we discuss each of these alternatives in detail.

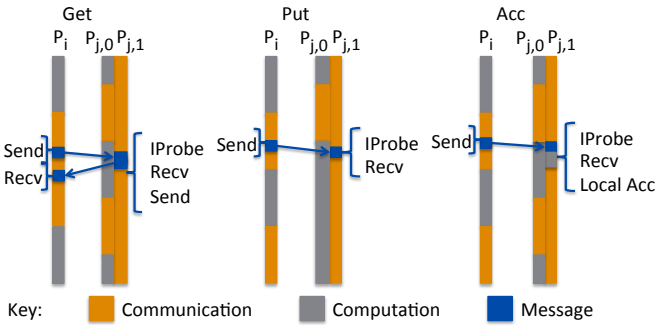

Fig. 3. One-Sided Communication Protocols using Two-Sided Communication Protocols in MPI with Multi-threading (MT). Protocols for Get, Put and Accumulate are on the left, middle, and right, respectively. Process P_i initiates a request to process P_j which is handled asynchronously by thread $P_{j,1}$.

C. Third Design: MPI Send/Recv with Multi-threading

Multi-threading support is a feature which allows multiple threads to make MPI calls with different threading modes. It is an important feature in the multi-core era to facilitate hierarchical decomposition of data and computation on deep memory hierarchies. Shared address space programming models such as OpenMP provide efficient support for multi-core/many-core architectures. MPI thread multiple mode allows invocation of MPI routines from any thread.

The computation model can be broadly classified in terms of *symmetric* and *asymmetric* work being performed by the threads. The symmetric model may require different thread support levels, depending upon algorithm design. As an example, a stencil computation can be performed using a thread multiple model (each thread reads/updates its individual edge) or thread serialized model (one thread coalesces reads/updates and sends them out as a sparse collective or individual point-to-point communication).

As an improvement over the previous send/receive design, progress is made using an asynchronous thread as shown in Figure 3. In our proposed design of ComEx on MPI multi-threading (MT), the asynchronous thread calls MPI_Iprobe after it has finished serving the send requests. We use a separate communicator each for communication between process-thread and thread-process. This reduces the locking overhead in the MPI runtime. However, even with this optimization, it is not possible to completely remove locking from the critical path.

Designing a communication runtime using MPI multi-threading is a non-trivial task. The primary reason is that the lock(s) used by the progress engine are abstracted (for performance portability), which results in non-deterministic performance observed with the MT design. Since the asynchronous thread is frequently invoking MPI_Iprobe (even on a separate communicator than the process thread), it has to

frequently relinquish the lock by using `sched_yield`. At the same time, if `sched_yield` is not used, the resulting performance is non-deterministic.

To eliminate the non-determinism as a result of locking in the critical sections, a possibility is to use *dynamic process management* which we explore in the next section.

D. Fourth Design: Dynamic Process Management

DPM is an MPI feature which allows an MPI process to spawn new processes dynamically. Using DPM, a new inter-communicator can be created which can be used for communication. An advantage of such an approach is that it alleviates a need to use multi-threading, and yet it provides asynchronous progress by spawning new processes.

A possible approach is to spawn a few (x) number of processes per node and to use them for asynchronous progress. The original and spawned processes would then attach to the same shared memory region in order for the spawned processes to make progress on behalf of the processes within its shared memory domain. This approach is very similar to the approach proposed by Krishnan *et al.* [20]

Unfortunately, dynamic process management is not available on most high-end systems. As an example, the Cray Gemini system used in our evaluation does not support dynamic process management even though the system has been in production use for two years. DPM requires support from the process manager. However, many implementations do not support dynamic process management since it is not commonly used in MPI applications. Due to a lack of available implementations of DPM, we do not evaluate this approach, although a design proposed by Krishnan *et al.* [20] would have been a useful comparison point.

IV. Approach: Progress Ranks

In this section, we present our proposed approach which addresses the limitations discussed in Section III. Specifically, the proposed approach uses the two-sided semantics (for performance reasons) and asynchronous progress by automatically and transparently splitting the world communicator. This allows a subset of processes to accelerate communication progress.

A. Basic Design

The PGAS models provide a notion of distributed data structures and load/store (get/put) on these structures by using array slice indices. A process does not address another process explicitly for communication since the meta-data management is handled automatically. This property of PGAS models has substantial impact on our proposed approach since it can be leveraged to automatically *split* the user level processes among ones which execute the algorithm and ones which provide the asynchronous progress. The data-centric view of the PGAS models facilitates this *splitting* without requiring any change in the application.

The proposed split of user-level processes facilitates the use of MPI two-sided semantics and the protocol processing by the *progress ranks (PR)*. The PR approach alleviates a need for guarding the critical sections by locks as is the case in the multi-threading approach. It also eliminates a dependency on MPI-RMA which requires an entirely separate design for best performance. Figure 4 shows the split of the processes in compute ranks and progress ranks. As shown in the figure, a simple configuration change would allow a user-defined number of progress ranks on a node - without any source code change in the application. The upcoming section provides details of our proposed approach, which is subsequently referred to as the Progress Rank (PR) based approach for rest of the paper.

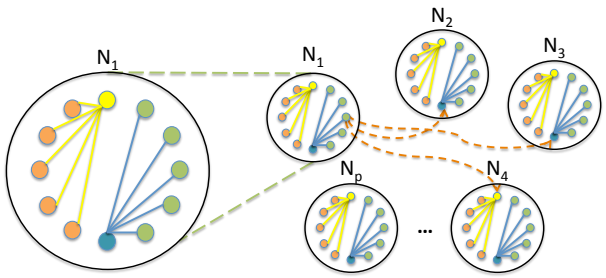

Fig. 4. Translation of communication operations in the PR approach. Left: A typical node with with two PR ranks (blue and yellow circles). Blue PR is responsible for performing RMA operations on the memory hosted by green user-level processes, Right: A user-level process communicates with PR ranks on other nodes for RMA requests. On-node requests are performed using shared memory.

B. Primary Details

Figure 4 shows the separation of data-serving processes in PR and user-level processes. The PR approach allows one to create a user-defined number of PR ranks to allow mapping with NUMA architectures and heterogeneous architectures (such as using an Intel Sandybridge and Intel KNF architecture together). A user-defined number of PR ranks also allows an application to allocate data structures with memory affinity. The figure shows a case where a specific instance of the PR approach uses two PR ranks (depicted by blue and yellow circles in this case).

The PR approach uses shared memory between the progress rank and the user-level processes within its shared memory domain, as shown in Figure 5. The same shared memory is also used for on-node communication to reduce the number of memory copies and eliminate superfluous communication with the progress rank. To minimize the space complexity, shared memory segments are created and destroyed on-demand. The cost of creation/deletion of shared memory segments is amortized since the distributed data structures (such as arrays) remain persistent for most applications. Inter-node communication is handled by redirecting the request to the progress rank corresponding to the target process on its node. In the following sections, we present communication protocols for facilitating RMA operations and also discuss space and time complexity analysis of the PR approach.

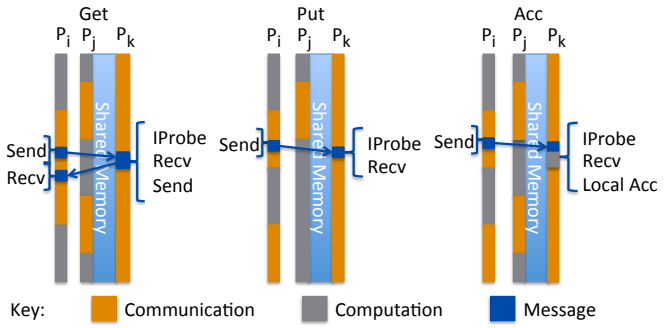

Fig. 5. One-Sided Communication Protocols using Two-sided Communication Protocols in MPI with Progress Ranks. Protocols for Get, Put, and Accumulate are on the left, middle, and right, respectively. Process P_i initiates a request to the progress rank P_k for the RMA targeting P_j. P_j and P_k reside within the same shared memory domain.

C. Complexity Analysis of Communication Protocols

The effectiveness of our approach is in its simplicity and its potential for near-optimal performance in comparison to other MPI ports. However, it is important to present the protocols for important communication primitives and present their space/time complexity.

Algorithm 1 shows the pseudocode executed by compute processes. It shows the protocols for each of the Get, Put, Accumulate and FetchAndAdd communication primitives. The $progress$ function is invoked as necessary to make progress on outstanding $send$ and $recv$ requests. Algorithm 2 shows the progress function executed by the *progress ranks*. The protocol processing is abstracted to hide the details such as a $LocalAccumulate$ function, which can use high performance libraries/intrinsics directly.

For small messages, the Put primitive is expected to use the eager MPI protocol which involves a copy by each of the sender and receiver. However, for large messages, a zero copy based approach is used in MPI with a rendezvous protocol. Hence, $T_{put} \approx m \cdot G$ where m is the message size and G is the inverse of the bandwidth, which is equivalent to the performance of the native ports. Using a similar analysis, T_{get} for large messages is expected to be similar to native ports. Small message Get transfer is impacted by the copies on both sides. The protocol for Get uses a combination of a send and receive on each side as shown in Algorithm 1's GET procedure and Algorithm 2's PROGRESS procedure. Hence: $T_{get} = T_{RDMAGet} + 4 \cdot \gamma$ where γ is the memory copy cost on each side. RDMA-enabled networks such as InfiniBand [9] and Gemini [16] provide a RDMA Get latency of $\approx 1\mu$s, hence the impact of γ can be non-trivial on the latency for small messages.

The FetchAndAdd operation is translated as an $irecv$ and $send$ on the initiator side with a PR rank needing to perform a $recv$ of the request, a local compute, and a send of the initial value back to the initiator. By using two-sided semantics, our approach cannot take advantage of hardware atomics on the NIC such as the ones available for Cray Gemini [16] and InfiniBand [9]. The $accumulate$ operations are bounded by the performance of the put operation and the $localacc$ function. For large accumulates, we expect the performance to be similar to a native port implementation since there are no known network hardware implementations of arbitrary size accumulates.

Algorithm 1: ComEx Routines

Input: source address s, target address d, message size m, target process r

Procedure PUT(s, d, m, r)
$r1 \leftarrow$ TranslateRankstoPR(r)
if $m < \delta$ **then**
  $buf \leftarrow$ InlineDataWithHeader(m)
  Send$(buf \ldots r1)$
else
  $buf \leftarrow$ PrepareHeader(m)
  Send$(buf \ldots r1)$
  Send$(d \ldots s)$
end if

Procedure GET(s, d, m, r)
$r1 \leftarrow$ TranslateRankstoPR(r)
$handle \leftarrow$ Irecv$(d \ldots r1)$
$buf \leftarrow$ PrepareHeader(m)
Send$(buf \ldots r1)$
Wait$(handle)$

Procedure ACC(s, d, m, r)
$r1 \leftarrow$ TranslateRankstoPR(r)
if $m < \delta$ **then**
  $buf \leftarrow$ InlineDataWithHeader(m)
  Send$(buf \ldots r1)$
else
  $buf \leftarrow$ PrepareHeader(m)
  Send$(buf \ldots r1)$
  Send$(d \ldots s)$
end if

Procedure FADD(s, d, m, r)
$r1 \leftarrow$ TranslateRankstoPR(r)
$buf \leftarrow$ InlineDataWithHeader(m)
Send$(buf \ldots r1)$
Recv$(d \ldots r1)$

D. Discussion: Comparison with GASNet

GASNet [2] is a communication subsystem for supporting UPC [2] and other languages and primarily relies on Active Messages [21]. GASNet supports a conduit based on Active Message MPI (AMMPI) which acts as the default port should a native conduit be unavailable on the system. While there is no published literature on AMMPI and GASNet over AMMPI, a look at the open source code provides several details of the implementation. There are two possible modes of execution in AMMPI: with and without the support of asynchronous threads. We argue that each of these modes reduce to one of the approaches proposed in this paper. The AMMPI approach without asynchronous threads relies on the communicating processes to make progress on the active messages which

Algorithm 2: ComEx PR Progress Routine

```
Input: source address s, target address d,
message size m, target process r

Procedure PROGRESS()
while running do
  flag ← Iprobe()
  if flag then
    header ← Recv()
    if header.messageType ==PUT then
      if IsDataInline(header) then
        CopyInlineData(header)
      else
        Recv(header.d ... header.r)
      end if
    else if header.messageType ==GET then
      Send(header.s ... header.r)
    else if header.messageType ==ACC then
      LocalAcc(header.d)
    else if header.messageType ==FADD then
      counter ← LocalFAdd(header.d)
      Send(counter ... header.r)
    end if
  end if
end while
```

is similar to our proposed MPI-TS approach. The AMMPI approach based on asynchronous threads requires one or more threads to make progress on MPI, requiring multi-threaded MPI implementation. As a result, the approach based on asynchronous threads reduces to the proposed MPI-MT implementation. However, GASNet does not provide a port similar to the proposed PR approach which we contend is the best possible approach among all MPI approaches.

We observe that an approach similar to the proposed PR approach is insufficient for supporting active messages because an active message must be executed in the address space of the target. Hence, an active message must be executed either by the target process directly or a thread which shares the same address space. However, this limitation is not problematic to our approaches since the RMA communication model - the primary requirement of PGAS models - can be effectively supported without active messages. Specifically, in the PR approach, the RMA requests from processes on other nodes are able to be served by a "progress rank" because the PGAS data structures use shared memory for processes on the same node.

V. Performance Evaluation

We present a performance evaluation of the approaches discussed in the previous section using a set of communication benchmarks, a graph kernel, a SpMV kernel, and full application with NWChem. Table I shows the various design alternatives considered in this paper and indicates whether they were considered for evaluation.

The TS implementation is not considered for evaluation because it is many orders of magnitude slower than rest of the implementations considered in this paper. As an example, even on a moderately sized system with NWChem, we noticed 10-15x performance degradation in comparison to the native approach. The TS approach requires explicit process intervention for RMA progress which makes it very slow in comparison to the other approaches. The DPM approach is not evaluated because dynamic process management is not supported on the high-end systems considered for evaluation in this paper. For the rest of the implementations, we have used process/thread pinning with no over-subscription.

TABLE I
Different Approaches Considered in this paper.

	Approach	Symbol	Implemented	Evaluated
1	Native	NAT	Yes	Yes
2	MPI-RMA	RMA	Yes [14]	Yes
3	MPI Two-sided	TS	Yes	No
4	MPI Two-sided + MT	MT	Yes	Yes
5	MPI Two-Sided + DPM	DPM	Yes [20]	No
6	MPI Progress Rank	PR	Yes	Yes

We used two systems for our performance evaluation:
NERSC Hopper is a Cray XE6 system with 6,384 compute nodes made up of two twelve-core AMD MagnyCours processors. The compute nodes are connected in a 3D torus topology with the Cray Gemini Interconnect. We used the default Cray MPI library on this system. This system is referred to as *Hopper* for rest of the performance evaluation.
PNNL Institutional Computing Cluster (PIC) is a 604 node cluster with each node consisting of two sixteen-core AMD Interlagos processors where the compute nodes are connected using a QLogic InfiniBand QDR network. We have used MVAPICH2 for the MPI library on this system. This system is referred to as *IB* for the rest of the performance evaluation.

A. Simple Communication Benchmarks

The purpose of the simple communication benchmarks is to understand the raw performance of communication primitives when the processes are well synchronized. Figures 6 and 7 show the Get communication bandwidth performance. As expected, NAT provides the best performance for Gemini and IB. The PR implementation on Hopper is based on MPI/uGNI, while the native implementation is based on DMAPP [8], so the difference in peak bandwidth for PR and NAT can be attributed to the use of different communication libraries. The RMA implementation provides sub-optimal performance on all message sizes in comparison to the NAT and PR implementations. On IB, the PR and RMA implementations perform similarly. The MT implementation uses MPI thread multiple mode and consistently performs sub-optimally in comparison to other approaches primarily due to lock contention. Similar trends are observed for Put communication primitives in Figure 8 with a drop at 8Kbytes for RMA, PR and MT implementations due to the change from eager to rendezvous protocol at that message size.

It is expected that the NAT implementation provides the best possible performance. For the rest of the sections, we only compare the performance of the MT, PR and RMA

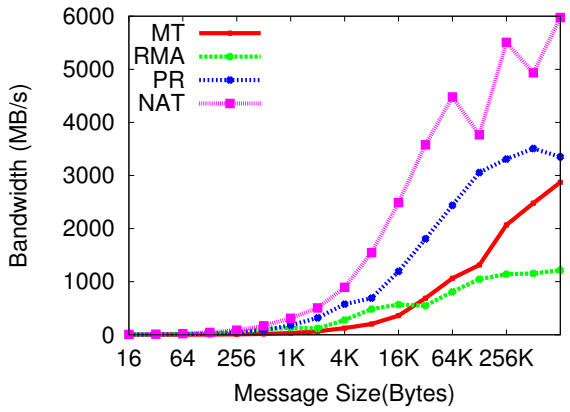

Fig. 6. Gemini, Get Performance

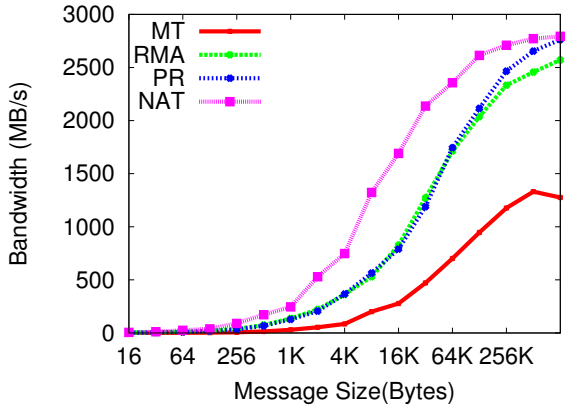

Fig. 7. IB, Get Performance

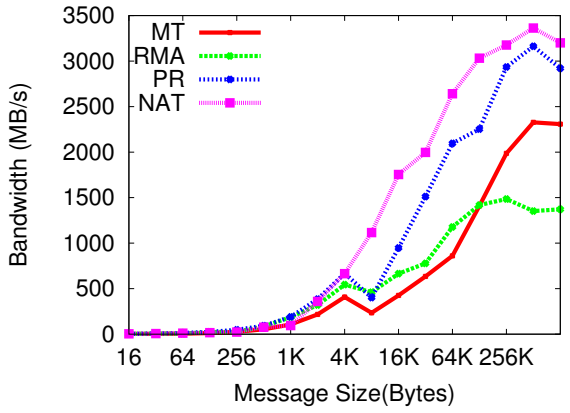

Fig. 8. Gemini, Put Performance

implementations since they provide fair comparison against each other.

B. Sparse Matrix Vector Multiply

SpMV ($A \cdot y = b$) is an important kernel which is used in scientific applications and graph algorithms such as PageRank. Here, we have considered a block CSR format of A matrix and a one-dimensional RHS vector (y), which are allocated

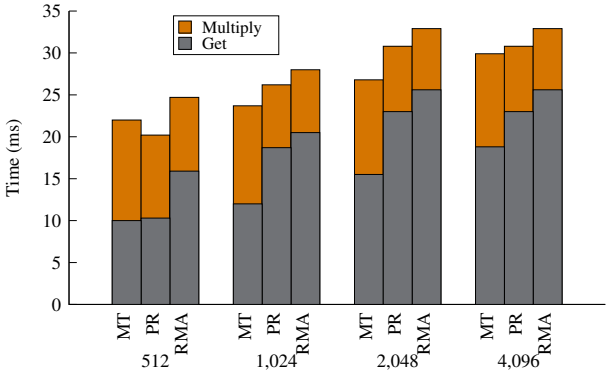

Fig. 9. SpMV, Hopper

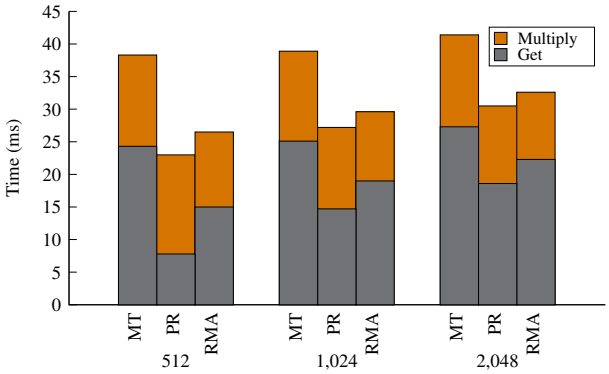

Fig. 10. SpMV, IB

and distributed using Global Arrays. The data distribution is *owner-computes*, resulting in local accumulates to b. The y is distributed evenly among processes. A request for non-local patches of y uses the *get* communication primitive. The sparse matrix used in this calculation corresponds to a Laplacian operator using a 7-point stencil and 3D orthogonal grid.

Figures 9 and 10 show the performance on Hopper and PIC systems, respectively. A weak scaling calculation mode is used with sizes varying from $(400 \cdot 400 \cdot 400)$ to $(800 \cdot 800 \cdot 800)$. The overall calculation is dominated by the *get* time. The MT port provides the best performance, with RMA being the worst among the three implementations. The MT port is executed using an asynchronous thread for every process, while the PR port uses a single rank per node for accelerating the RMA requests. A large majority of *get* requests are served from remote nodes, where a progress rank in the PR implementation needs to serve the requests of multiple y patches. The MT implementation has an asynchronous thread for every process, which reduces the *get* time in SpMV. However, for the IB system, the PR implementation performs the best among the three implementations. This is attributed to the sub-optimal implementation of the progress engine in MPI.

C. Graph Kernel: Triangle Counting

Triangle Counting (TC), among other graph algorithms, exhibits irregular communication patterns and can be implemented using PGAS models. In graphs with R-MAT structure

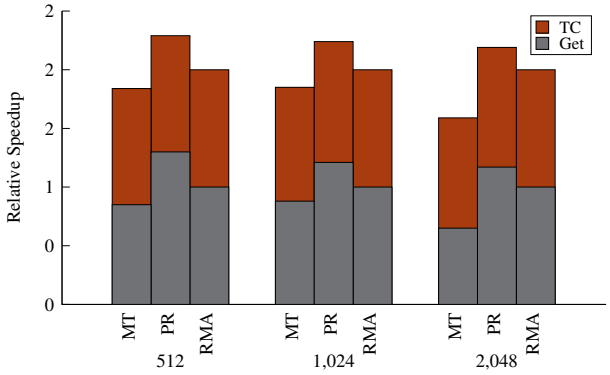

Fig. 11. Triangle Counting, IB

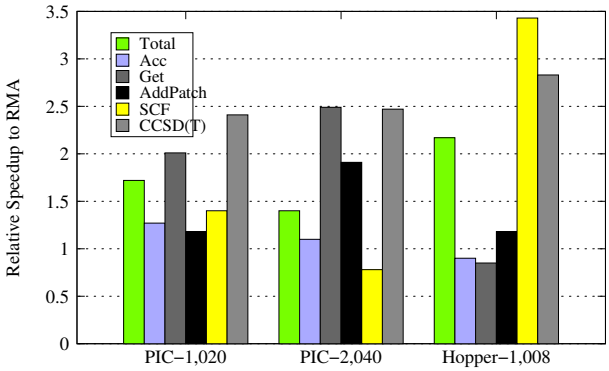

Fig. 12. NWChem CCSD(T) results for PR relative to RMA. MT did not finish execution in 1800s for 240, 1020 procs and 3600s for 2040 procs.

such as Twitter and Facebook, it is frequently important to detect communities. An important method to detect communities is by finding *cliques* in the graphs. Since CLIQUE is an NP-complete problem, a popular heuristic is to calculate cliques with a size of three which is equivalent to finding triangles in a graph. We show an example of community detection in natural (power-law) graphs, where the algorithm needs to calculate the number of triangles in a given graph. The edges are easily distributed using a compressed sparse row (CSR) format. The number of vertices are divided equally among the processes.

For TC, we allocate a CSR edge array using one-dimensional Global Arrays. The computation is distributed equally among processes where each process gets v/p number of vertices for computation. Figure 11 shows the speedup of get and $trianglecount$ on the IB system. The PR implementation provides a speedup of 1.31x, 1.21x and 1.17x on 512, 1024 and 2048 processes respectively. The speedup can be attributed to the asynchronous progress made in the PR port by the progress ranks. The implementation of the TC algorithm reuses the buffers for getting the neighbor list. This facilitates zero-copy transfer of the edge list, since most MPI libraries perform *lazy deregistration* of buffers for reuse.

The MT implementation performs poorly in comparison to the RMA and PR implementations. The MT implementation also suffers a slowdown in get communication, since it has to frequently use the sched_yield operation. However, the overall slowdown is worse, if the sched_yield operation is not used.

D. NWChem

We have evaluated the NWChem CCSD(T) and SCF modules respectively on the Hopper and PIC systems, in each case using naphthynes molecules. For 1020 processes on PIC, and 1008 processes on Hopper, we have used the cc-pvdz basis set which has 170 basis functions. For 2040 processes on Hopper, we have used the cc-pvtz basis set which has 380 basis functions. The MT implementation did not finish execution in its allocated time of 1800 seconds for any of the process counts. The MT implementation could not be run to completion due to a limited time allocation on these supercomputers. Hence, we compare the speedup of the PR approach relative to the RMA approach proposed previously by Dinan *et al.* [14]. Figure 12 shows the performance of NWChem on 1020 and 2040 processes with the CCSD(T) module on PIC and 1008 processes with CCSD(T) on Hopper. Relative speedups are calculated for the overall time, the SCF and CCSD(T) modules, and time-consuming functions such as Get, Accumulate and AddPatch.

The PR implementation provides a relative speedup to RMA of 1.72x on 1020 processes (PIC), 1.4x on 2040 processes (PIC) and 2.17x on 1008 processes (Hopper). The primary consumer of time in these calculations is CCSD(T), which provides a relative speedup of 2.41x, 2.47x and 2.83x. For each of the calculations CCSD(T) takes $\approx 80\%$ of the computation time. The get communication primitive provides a relative speedup of 2x and 2.5x in comparison to the RMA implementation on PIC. The SCF module provides a 3.2x speedup on Hopper, however, it is slightly slower than the RMA implementation on 2040 processes (PIC). Since CCSD(T) is the dominant module, the overall speedup is 1.4x. The overall speedup is slightly abated for each of the runs because NWChem performs intermediate I/O which performs similarly on all implementations.

E. Evaluation Summary

Our performance evaluation reveals that the proposed PR approach outperforms each of the other MPI approaches on a spectrum of evaluation criteria: communication benchmarks, community detection kernel in graphs, sparse matrix-vector multiply and a full application, NWChem. In a select few cases MPI-RMA did perform as good or slightly better, as was the case for get performance on the IB system and a few functions profiled within NWChem. The MT approach showed promise in the communication benchmarks, however its performance was stagnant for a real application even though other applications using multi-threaded MPI's thread multiple mode have been shown to scale well [22].

VI. Related Work

There have been a few efforts in using MPI as a communication target for PGAS models. We discuss them in this section. Bonachea *et al.* have presented the problems in using MPI as

a compilation target for PGAS languages with UPC as a case study [23]. However, the critique is only partially justified as Bonachea's argument does not take into account non cache-coherent architectures, which is the primary reason for the restrictions on conflicting memory accesses in MPI 2.0 RMA. Dinan *et al.* have presented an implementation of ARMCI using MPI-RMA [14]. They concluded that restrictions in the MPI-RMA 2.0 standard and their implementations lead to significant performance degradation in comparison to native ARMCI implementations on most platforms including InfiniBand, Blue Gene/P, and Cray Gemini Interconnect. Dinan's conclusion from the paper is a strong indication that while MPI-RMA provides a matching interface to ComEx, the search for an ideal PGAS runtime may not be provided by MPI-RMA. Hence, this paper is an important step to address the limitations. Gropp *et al.* have presented issues in designing a multi-threaded MPI implementation, however, they restrict the design to context-id allocation for communicators [24]. Balaji *et al.* have also presented approaches for fine-grained multi-threading in MPI [25]. Hoefler *et al.* have discussed the issues with multiple threads calling MPI_Probe and MPI_Recv together, which is not safe [26]. However, this issue is not applicable to our proposed design since only the asynchronous thread is involved in calling MPI_Probe and MPI_Recv.

VII. Conclusions

As the popularity of PGAS models continue to rise, it becomes more important that highly tuned communication subsystems are available to enable these models across a wide range of systems. This work has demonstrated that highly-tuned two-sided semantics are sufficient for implementing one-sided semantics in the absence of a native implementation. This result should continue to affirm system procurement requirements of optimized two-sided communication while suggesting that one-sided communication can be readily improved in the future using the existing MPI interface based on our proposed approach. This work narrows the performance gap between native and MPI-based runtimes for PGAS models and succeeds in making MPI-based runtimes for PGAS models an acceptable alternative when native implementations are not feasible to implement or readily available.

Acknowledgments

This material is based upon work performed under the Performance Health Monitoring for Large-Scale Systems project, supported by the U.S. Department of Energy (DoE) Office of Science (OS), Office of Advanced Scientific Computing Research. Additional support was provided by the eXtreme Scale Computing Initiative (http://xsci.pnnl.gov) of Pacific Northwest National Laboratory operated by Battelle for DoE under Contract DE-AC05-76RL01830. This research used resources of the National Energy Research Scientific Computing Center, which is supported by OS under Contract No. DE-AC02-05CH11231.

References

[1] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[2] P. Husbands, C. Iancu, and K. A. Yelick, "A Performance Analysis of the Berkeley UPC Compiler," in *ICS*, 2003, pp. 63–73.

[3] P. Charles *et al.*, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *OOPSLA*. ACM, 2005, pp. 519–538.

[4] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *JHPCA*, vol. 21, no. 3, pp. 291–312, 2007.

[5] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[6] A. Geist *et al.*, "MPI-2: Extending the message-passing interface," in *Euro-Par, Vol. I*, 1996, pp. 128–135.

[7] A. Vishnu, D. J. Kerbyson, K. Barker, and H. J. J. V. Dam, "Designing scalable pgas communication subsystems on blue gene/q." Boston: 3rd Workshop on Communication Architecture for Scalable Systems, 2013.

[8] A. Vishnu, J. Daily, and B. Palmer, "Scalable PGAS Communication Subsystem on Cray Gemini Interconnect." Pune, India: HiPC, 2012.

[9] A. Vishnu and M. Krishnan, "Efficient On-demand Connection Management Protocols with PGAS Models over InfiniBand," in *CCGrid*, 2010.

[10] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu, "The tofu interconnect," in *HOTI*, 2011, pp. 87–94.

[11] M. Xie, Y. Lu, L. Liu, H. Cao, and X. Yang, "Implementation and evaluation of network interface and message passing services for tianhe-1a supercomputer," in *HOTI*, 2011, pp. 78–86.

[12] M. Valiev *et al.*, "Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477 – 1489, 2010.

[13] Subsurface Transport over Multiple Phases, "STOMP," http://stomp.pnl.gov/.

[14] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays pgas model using mpi one-sided communication," in *IPDPS*, 2012, pp. 739–750.

[15] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems," in *Lecture Notes in Computer Science*. Springer-Verlag, 1999, pp. 533–546.

[16] A. Vishnu, M. ten Bruggencate, and R. Olson, "Evaluating the potential of cray gemini interconnect for pgas communication runtime systems," in *HOTI*, 2011, pp. 70–77.

[17] G. R. Gao and V. Sarkar, "Location consistency-a new memory model and cache consistency protocol," *IEEE Trans. Comput.*, vol. 49, pp. 798–813, 2000.

[18] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998.

[19] D. Chavarria-Miranda, S. Krishnamoorthy, and A. Vishnu, "Global futures: A multithreaded execution model for global arrays-based applications," in *CCGrid*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 393–401.

[20] M. Krishnan and V. Tipparaju, "Extending the MPI2 One-sided Model," in *HPC Asia*, 2009.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *International Symposium on Computer Architecture*, 1992, pp. 256–266.

[22] J. Daily, S. Krishnamoorthy, and A. Kalyanaraman, "Towards scalable optimal sequence homology detection," in *ParGraph*, 2012.

[23] D. Bonachea and J. Duell, "Problems with using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations," *JHPCA*, vol. 1, no. 1-3, pp. 91–99, 2004.

[24] W. D. Gropp and R. Thakur, "Issues in developing a thread-safe mpi implementation," in *PVM/MPI*, 2006, pp. 12–21.

[25] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Fine-grained multithreading support for hybrid threaded mpi programming," *JHPCA*, vol. 24, no. 1, pp. 49–57, 2010.

[26] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine, "Efficient mpi support for advanced hybrid programming models," in *EuroMPI*, 2010, pp. 50–61.