CAF Events Implementation Using MPI-3 Capabilities

Alessandro Fanfarillo
National Center for Atmospheric Research
Boulder, CO, USA
elfanfa@ucar.edu

Jeff Hammond
Intel Corporation
Portland, OR, USA
jeff.r.hammond@intel.com

ABSTRACT

MPI-3.1 is currently the most recent version of the MPI standard. It adds important extensions to MPI-2, including a simplified semantic for the one-sided communication routines and a new tool interface, capable of exposing performance data of the MPI implementation to users and libraries. These and other new features make MPI-3 a good candidate for being the transport layer of PGAS languages like Coarray Fortran.

OpenCoarrays, the free coarray implementation used by the GNU Fortran compiler, implements almost all Coarray Fortran 2008 and several Coarray Fortran 2015 features on top of MPI-3. Among the Fortran 2015 features, one of the most relevant for performance improvement is *events*; such a feature represents a fine grain synchronization mechanism based on a limited implementation of the well known semaphore primitives.

In this paper, we analyze two possible implementations of *events* using MPI-3 features and show how to dynamically select the best implementation, according to the capabilities provided by the MPI implementation. We also show how *events* can improve the overall performance by reducing idle times in parallel applications.

Keywords

MPI; PGAS; Coarray; Fortran

1. INTRODUCTION

Since the release of the MPI-2 standard in 2007, lots of changes have happened in the High Performance Computing (HPC) world. Massively parallel systems with over a million of cores are now a reality; Remote Direct Memory Access (RDMA) support in networks has become mainstream, allowing to overlap communication with computation and improving collective communication performance; single-core processors have disappeared and multi-threading programming has become very important for applications.

EuroMPI '16, September 25 - 28, 2016, Edinburgh, United Kingdom

ISBN 978-1-4503-4234-6/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2966884.2966916

In order to address these new challenges in HPC, the MPI-3 standard has introduced several changes. In particular, major changes have been made on the one-sided communication routines in order to overcome the limitations documented by Bonachea et al. [5]. Furthermore, new features like non-blocking and neighbor collectives, shared-memory programming and new functionalities for the inspection and manipulation of MPI control and performance variables have been added.

With the increasing availability of the RDMA support in computer networks, the so called Partitioned Global Address Space (PGAS) model has evolved in the last few years. The PGAS model is a parallel programming model that assumes a global memory address space logically partitioned, with a portion of the memory being assigned to a specific processor. The model attempts to combine (and get the best from) the Single Program Multiple Data (SPMD) approach, used in the distributed memory systems, and the semantic of shared memory systems. In the PGAS model, every process has its own memory address space but it can share a portion of its memory with other processes. Some languages that implement the PGAS model are: Coarray Fortran (CAF) [25, 26], Unified Parallel C (UPC) [30], Chapel [8], Fortress [2], X10 [9] and Global Arrays [24].

Several features introduced in the MPI-3 standard, in particular the new Remote Memory Access (RMA) support, have made MPI a good candidate for being used as transport layer for PGAS languages. OpenCoarrays [16], the free coarray Fortran implementation used by the GNU Fortran compiler, relies on MPI-3 to implement the coarray features specified in the Fortran 2008 and Fortran 2015 standard. Although the coarray operations usually have a good matching with the MPI one-sided routines, the lack of fully asynchronous support in MPI implementations and/or networks [7] represents a big limitation to the realization of a high-performing coarray implementation.

In this paper, we show how the MPI-3 features can be used to implement *events*: a fine-grain synchronization mechanism introduced in the Fortran 2015 standard, capable of reducing idle times in parallel applications and, consequently, improving the overall performance.

The rest of this paper is organized as follows: Section 2 introduces Coarray Fortran and the new features of the Fortran 2015 standard. Section 3 shows pros and cons of using MPI as transport layer for PGAS languages; in particular highlights the difficulties for having truly asynchronous support and how PGAS languages can benefit from the new features introduced in MPI-3. Section 4 describes a dynamic

selection algorithm used for choosing the best *events* implementation. Section 5 describes the test cases used for the performance evaluations of two *events* implementations. In Section 7, we report the results of our tests and show that an efficient implementation of *events* can brought a speedup up to 2x compared to other solutions. Finally, in Section 9 we provide our conclusions.

2. FORTRAN 2015

Coarray Fortran (also known as CAF) is a syntactic extension of Fortran 95/2003 which was proposed in the early 1990s by Robert Numrich and John Reid [25] and is now part of the Fortran 2008 standard (ISO/IEC 1539-1:2010) [26]. The main goal of coarrays is to allow Fortran users to create parallel programs without the burden of explicitly invoking communication functions or directives such as with MPI and OpenMP.

The coarray definition included in Fortran 2008 defines a simple syntax for accessing data on remote images, synchronization statements and collective allocation and deallocation of memory on all images. Although these features allow one to write a totally functional coarray program, they do not allow to express more complex and useful mechanisms for synchronization, images organization and failure management.

Technical Specification 18508 [21] proposes the following extensions to the coarray facilities defined in Fortran 2008:

- teams;
- failed images;
- events;
- new atomic and collective procedures.

Teams allows one to execute more effectively and independently parts of a larger problem by grouping the images into non-overlapping teams. A class of problems that can benefit of such feature is multiphysics codes (e.g., climate models).

Failed images provides a mechanism to identify what images have failed during the execution of a program. This obviously affects the resilience of programs running on large systems.

Events provide a convenient mechanism for ordering execution segments on different images without requiring that those images arrive at synchronization point before any is allowed to proceed. This feature implements a fine grain synchronization mechanism based on a limited implementation of the well known semaphore primitives. In this work, we show the potential of this feature and the challenges related to its efficient implementation using MPI.

Fortran 2008 does not provide intrinsic procedures for commonly used collective operations and provides only minimal support for atomic memory operations. Such procedures can be highly optimized for the target computational system, providing significantly improved program performance. A typical example of collective operation introduced by TS-18508 is *co_broadcast*. This intrinsic allows one to broadcast data from a source image to a group of images as one single command. In Fortran 2008, the only way to implement this operation is to run a do-loop on the source image and perform a "put" operation on each target image, one at a time. TS-18508 enriches the available set of atomic intrinsics (e.g., new *atomic_fetch_and_op* intrinsics).

All the features defined in TS-18508 are going to be part of the Fortran 2015 standard.

2.1 Coarray Synchronization Methods

As already mentioned in Section 2, *events* provide a fine grain synchronization mechanism, based on a limited implementation of semaphores, capable of improving parallelism in coarray applications. In order to understand how *events* can improve parallelism, we present the synchronization methods provided by Fortran 2008 and highlight the main differences.

A program that uses coarrays is treated as if it were replicated at the start of execution; each replication is called an image. Each image executes asynchronously and explicit synchronization statements are used to maintain program correctness.

A piece of code contained between synchronization statements is called *segment* and a compiler is free to apply all its optimizations inside a segment. On each image P, the statement execution order determines the segment order $(P_1, P_2, ...)$. A pair of segments P_i and Q_j are called *unordered* if P_i neither precedes nor succeeds Q_j. There are restrictions on what is permitted in a segment that is unordered with respect to another segment. In particular, unless the variable is atomic (or event), if a variable is defined in a segment on an image, it must not be referenced, defined or become undefined in a segment on another image unless the segments are ordered (for more details about segments and restrictions we suggest to read Metcalf et al. [23]).

The full list of image control statements of Fortran 2008 is:

- `sync all` statement;
- `sync images` statement;
- `lock` or `unlock` statements;
- `sync memory` statement;
- `allocate` or `deallocate` statements involving coarrays;
- `critical` or `end critical` statements;
- `end` or `return` statement that involves an implicit deallocation of a coarray;
- a statement that completes the execution of a `block` and results in an implicit deallocation of a coarray;
- `stop` or `end program` statement.

These image control statements synchronize with all or a part of the images composing a program.

For example, the `sync all` statement represents a barrier for all images, whereas `sync images(image-set)` performs a synchronization of the image that executes it with each of the other images in its image-set. Even though `sync images(image-set)` represents a more flexible way to synchronize images, it always implies a synchronization between execution flows of several images. This may lead to idle times on one or more images, waiting for a slower image to reach the synchronization statement. Figure 1 depicts a typical scenario where Image 1 needs an entire array A from Image 2. Because Image 1 terminates the computation earlier than Image 2, it will wait until Image 2 reaches the `sync images` statement.

In order to reduce this phenomenon as much as possible, Fortran 2008 defines a set of *atomic* subroutines capable of performing an action on a remote coarray variable instantaneously. This "atomic" behavior allows to break the segment partial ordering and, potentially, to improve the performance by reducing idle times. Unfortunately, *atomic*

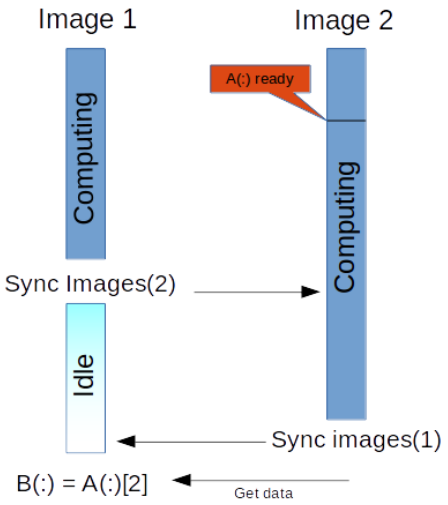


Figure 1: Get operation using sync images

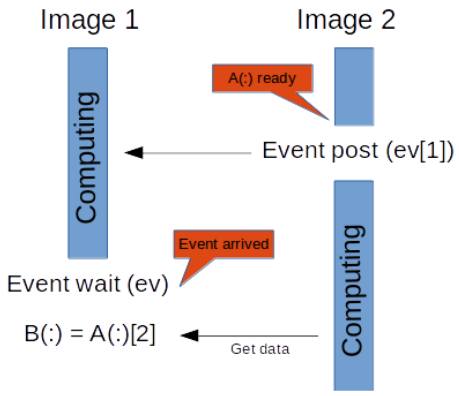


Figure 2: Get operation using events

subroutines (sometimes called *atomics*), as defined in Fortran 2008, have been revealed quite difficult to use correctly.

2.1.1 *Events*

Events represent the safer and more general implementation of *atomics*. An event coarray variable can be seen as a counter that can be incremented by any image, using the subroutine `event post`; this subroutine never blocks and should return as quick as possible. An image can wait for the event variable to reach a predefined value of posted events using the `event wait` subroutine; this blocking subroutine can be invoked only on local variables. Since an image may want to check the value of a local event variable without waiting, an `event_query` subroutine is also provided. The main difference of *events* from the general semaphores stands in the local applicability of the `event wait` and `event_query` subroutines; this restriction makes events safer, easier to use and highly performing. Figure 2 depicts the same scenario shown in Figure 1 using events. It is obvious that the `event_post` implementation must be as much as efficient and asynchronous as possible in order to obtain high performance.

3. MPI AS A PGAS TRANSPORT LAYER

The Message Passing Interface (MPI) execution model, thanks to its high performance, portability and standardization, is a de-facto standard in the High Performance Computing world and is installed and tuned on all supercomputers. The MPI standard has evolved from the initial version of 1994 and currently incorporates direct remote memory access (RMA) through one-sided functions, multithreading support, non-blocking and sparse collective communication operations and dynamic process management. Such new features make MPI-3 a good candidate for being the transport layer of PGAS languages [11]. In particular, the asynchronous communication required by the PGAS model can be easily implemented on top of the RMA one-sided functions of MPI-3. Although these operations map very well on the RDMA read and write operations provided by HPC network fabrics (like Cray*Gemini [33], IBM*Blue Gene/Q [34] and InfiniBand [35]), the synchronization models associated with the MPI one-sided operations are somewhat complicated. The MPI standard provides two synchronization modes: active and passive. In the active mode, the target process participates in the synchronization whereas, in the passive mode, the target process does not participate in the synchronization. In the latter case, all the processes accessing the memory exposed by a remote process have to synchronize amongst themselves, without participation of the target process. From the point of view of providing support for a PGAS language, the passive mode is the most suitable; in fact, it allows to overlap communication with computation, by reducing the synchronization penalty. Implementing passive MPI one-sided functions, even on network interfaces able to perform RDMA operations in hardware, may require the MPI implementation to check for transfer completion in order to *progress* the communication. In order to perform this check, the MPI library has to be called during the program execution. If the main program is busy doing other tasks (e.g., computation), the transfer cannot complete and it must be postponed until the target invokes the MPI library.

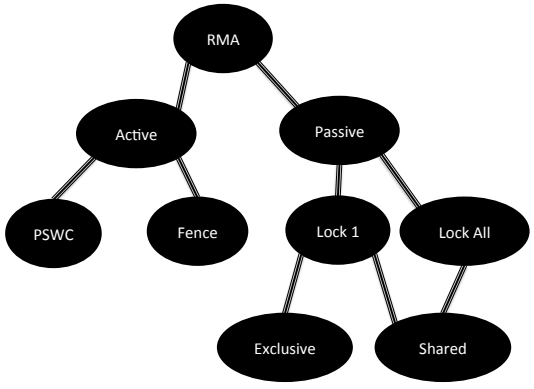


Figure 3: Taxonomy of MPI RMA synchronization motifs.

Asynchronous message progress is a very intricate and controversial topic in high-performance computing [6, 18, 19]. With the current available high-performance networks, there are essentially three strategies for asynchronous progress: programmer-directed progress, software-based progress (using threads, interrupts or processes, and hardware-based

progress. The manual progress gives complete control and responsability to the programmer for implementing message progress. A common solution for programmer-directed progress is to invoke functions like `MPI_Iprobe` that cause the implementation to enter the progress engine. Although this solution increases code complexity, it is a portable way to achieve asynchronous progress that does not entail the overheads associated with thread-based asynchronous progress within the MPI library, which is the most common implementation today.

Hoefler et al. [19] describe and analyze the thread-based approach. They conclude that thread-based progress based on polling (bypassing the OS) is beneficial only when separate computation cores are available for the progression thread. Using an interrupt-based approach (passing through the operating system) might be helpful in the case of oversubscribed nodes (the progress and user threads share the same core). In either case, passing through the operating system raises two concerns: 1) it seems unclear how big the interrupt latency and overheads are on a modern systems; 2) the scheduler has to schedule the progress thread right after the interrupts arrive so as to achieve asynchronous progress. This second issue can be faced by using real-time functionalities in the Linux kernel.

In [29], Si et al. propose to use dedicated communication processes (called ghost processes) for managing inter-node data transfers using two-sided communication. Once the data is received on the ghost process(es), it is delivered to the destination process using the MPI-3 shared memory capability. This mechanism ensures asynchronous progress on every process without incurring in any issue related to the use of threads.

Many hardware technologies have supported RMA-like features in hardware, and in these implementations, asynchronous progress does not require user- or software-agency. Many generations of Cray networks possess such features, including T3D [22], T3E [28], X1 [15] and X1E, Gemini [3] and Aries [1], as do IBM Blue Gene/P and Blue Gene/Q [10]. However, no known implementation to-date has supported *all* of the features of MPI RMA in hardware, which means that these cannot be true offload implementations. It is worth to mention that the Elan network hardware produced by Quadrics* [27] has been able to support all the MPI RMA features until the closure of the company in 2009.

The most common hardware features related to MPI RMA are remote read, write and some atomic operations, which are essential to PGAS programming models like SHMEM. Some of the features of MPI RMA that are less likely to be found implemented in hardware are floating-point atomic operations and noncontiguous buffers, both of which have been part of MPI RMA since MPI-2. Fortunately, conservative semantics make it possible to implement MPI RMA using a mixed implementation (hardware and software), at some loss of efficiency relative to a pure hardware implementation.

3.1 MPI Tool Information Interface

Understanding the performance issues of an MPI code is an operation that requires low-level information; for example, knowing how much time is spent in an `MPI_Recv` can help to understand whether the application suffers of poor load balancing or just high communication costs. Such a low-level information is usually hidden into the internal variables of the MPI implementation. For example, a typical information that can be useful to know (used also in this work) is *how many messages are in the Unexpected Message Queue waiting to be received?*.

With the new tools information interface introduced in MPI-3, MPI provides a standard way to access performance data contained inside the MPI implementation (called *performance variables*) and to internal variables that control the behavior of the implementation (called *control variables*). A typical *control variable* is the one that defines the threshold, associated with the message size, that decides whether a message should be sent using the eager or rendezvous protocol.

Although the performance variables are common to any MPI implementation (e.g., Unexpected Message Queue length), the MPI Forum does not specify a direct way to get the status of these variables. The reason is that such a low-level concepts are not MPI concepts; in other words, they are common but not necessary part of an implementation. For the case of the Unexpected Message Queue length variable, some MPI implementations may use a different approach to manage unexpected messages, such as rejecting and asking for a re-transmission instead of queuing. The intent of the MPI tools information interface is to enable an MPI implementation to expose implementation-specific details; for this reason is not possible to define variables that all MPI implementations must provide.

3.2 OpenCoarrays

OpenCoarrays [16] is an open-source software project for developing, porting and tuning transport layers that support coarray Fortran compilers. It targets compilers that conform to the coarray parallel programming feature set specified in the Fortran 2008 standard. It also supports several features proposed for Fortran 2015 in the draft Technical Specification TS-18508 "Additional Parallel Features in Fortran" [21]. OpenCoarrays uses a 3-clause BSD-style open-source license to facilitate its incorporation into free and proprietary compiler software and it is currently used by the GNU Fortran compiler. OpenCoarrays defines an application binary interface (ABI) that translates high-level communication and synchronization requests into low-level calls to a user-specified communication run-time library. This design decision liberates compiler teams from hardwiring communication-library choice into their compilers and it frees Fortran programmers to express parallel algorithms once, and reuse identical CAF source with whichever communication library is most efficient for a given hardware platform. At the time of this writing, OpenCoarrays covers almost all the Fortran 2008 coarray features, *events* and the collective/reduction and new *atomic* intrinsics belonging to the Fortran 2015 standard.

Since the first release of OpenCoarrays (August 2014), the widest coverage of coarray features was provided by a MPI based run-time library (LIBCAF_MPI). Because of the one-sided nature of coarrays, 99% of the run-time library uses MPI one-sided communication routines based on passive synchronization.

On the compiler side (GFortran), a coarray variable is represented by two entities: a) a token that stores the MPI window and b) a descriptor structure, that stores the address of the local variable and other information such as lower and upper bounds, rank and datatype. When a coarray state-

ment is found in the source code, the compiler translates the statement into an invocation to the right OpenCoarrays routine. During this operation, both token and descriptor are passed to OpenCoarrays.

Despite the good matching of coarray one-sided semantics and MPI one-sided routines, it should be noted that the behavior of some MPI routines may differ from the CAF counterpart. A typical example is the difference between `MPI_Get` and getting data from a remote coarray variable. For `MPI_Get`, the function call returns before the data arrives; the programmer can only assume that the operation has completed after a synchronization call (like `MPI_Win_Flush`). For coarrays, the Fortran semantics related to a variable assignment has to be respected; this means that the programmer can assume that the data has arrived as soon as the read operation returns.

3.3 Events using MPI

The most intuitive and straightforward implementation of *events* is the one based on Remote Memory Access (RMA) atomic operations and spin-locks. An event element is assumed to be a counter initialized to zero during the start-up phase; an invocation of `event post` is translated into an atomic increment of the target event variable. An invocation of `even wait` is translated into a spin-lock waiting for the counter (local event variable) to reach a predefined value. From now on we will refer to this approach as RMA-events.

The performance of this atomic-based implementation is strictly related to the performance of the MPI atomic operations and to the contention caused by several processes updating the same variable. In MPI 3, the atomic operations needed to implement this solution are represented by passive one-sided accumulate and fetch_and_op routines (`MPI_Accumulate` and `MPI_Fetch_and_op`).

As already explained in Section 3, asynchronous message progress is one of the most critical issues to address when MPI one-sided routines are used to implement PGAS functionalities. This is more than ever true when atomic functionalities are implemented on top of passive MPI one-sided routines, where latency is usually a critical factor. In case of a manual progress approach, the (passive) atomic operation completes only when the target invokes an MPI routine. Evidently, this is not a feasible way to implement a fine grain synchronization mechanism. Using a progress thread, based on interrupt or polling, seems to be a good alternative but unfortunately the latency introduced by the continuous polling or by the Operating System (interrupt) penalizes the performance. Finally, the offload MPI progress approach suffers of the lower performance of the embedded processors compared to CPUs. Summarizing, an event implementation based on one-sided atomic operations suffers of high latency and low performance due to MPI message progress mechanisms and contention on the event variable. A good way to provide low latency is to use two-sided communication; in fact, the data transfer is managed directly by the CPUs (that guarantees low latency) and MPI progress does not represent an issue anymore.

Because an image can post an event on a variable owned by a remote image asynchronously, the only way to ensure this behavior is to rely on the eager protocol, on the sender side, and on the Unexpected Message Queue (UMQ) on the receiver side. The eager protocol assumes that the receiver has enough space in its temporary buffers to store the message that is going to be sent. Such an assumption leads to minimal latency but holds only for short messages.

During an `event post()`, the image sends a small amount of data (2 integers) using the `MPI_Send` function to the remote image. Because of the small amount of data exchanged, the `MPI_Send` function (adopting the eager protocol) will not block and the data will be stored in a pre-allocated buffer on the receiver side. When an image invokes an `event wait` subroutine it spins in a while loop containing an invocation to the `MPI_Recv` routine; the loop terminates when the number of events directed to the target event variable is equal to the predefined value passed to the `event wait` subroutine. During this loop, the UMQ is examined and shrunk when a match with the predefined tag is found. From now on we will refer to this approach as P2P-events.

Even though this approach leads to minimal latency and truly asynchronous communication, it is based on the Unexpected Message Queue (unsafe practice) and implementation specific protocols (eager protocol). The UMQ is designed to manage situations where messages arrive but a corresponding receive has not been posted yet (thus there is not a buffer designated to it). When the application on the receiver side posts a `MPI_Recv`, the queue is traversed and, if a match is found, the message is copied into the specified buffer and then removed from the UMQ. If the receiver does not call `MPI_Recv`, the UMQ starts to grow without bounds until: 1) the memory occupied by the UMQ reaches a threshold specified by the MPI implementation or 2) all the RAM on the machine get occupied by the UMQ. In both cases, the program may crash or produce unexpected behaviors.

Summarizing, the RMA-events approach (that uses atomic MPI one-sided routines), leads to more stable results but also suffers of poor performance; on the other hand, the P2P-events approach (that uses a two-sided approach) leads to better performance but relies on unsafe mechanisms. Although relying on the UMQ is considered theoretically unsafe, it is practically safe as long as the queue status is kept under control.

In this work, we propose an *events* implementation based on both strategies; the selection of the best strategy is done dynamically, based on the support provided by the MPI implementation. In order to implement this algorithm safely, the control and performance variable described in 3.1 must be used.

4. DYNAMIC APPROACH SELECTION

As we already stated in Section 3.3, there are two main approaches to implement *events* in OpenCoarrays: RMA-events and P2P-events; both of them have pros and cons related to performance and stability. In this section we describe how to realize a dynamic selection of the *events* implementation, using the MPI Tool Information Interface capability provided by the MPI-3 standard.

4.1 Approach Selection

The idea is to start the execution using the P2P-events approach, in order to favor performance, and fallback to the RMA-events approach when the P2P-events reaches the limit or if the performance variables exposed by the MPI implementation do not allow to check the status of the UMQ and/or the eager protocol is not implemented.

In Algorithm 1 we sketch how the approach selection works during the execution of a coarray program.

```
1  Initialize RMA-events and P2P-events variables;
2  if UMQ perf var AND eager control var then
3  |  decide threshold for UMQ;
4  |  select P2P-events;
5  else
6  |  select RMA-events;
7  end
8  selection check;
9  while program not terminated do
10 |  if P2P-events then
11 |  |  if UMQ > threshold then
12 |  |  |  broadcast switch to RMA;
13 |  |  |  empty UMQ queue from events;
14 |  |  |  select RMA-events;
15 |  |  end
16 |  |  if switch_to_RMA received then
17 |  |  |  empty UMQ queue from events;
18 |  |  |  select RMA-events;
19 |  |  end
20 |  end
21 end
```

Algorithm 1: Approach selection

At line 1, the variables associated with both approaches, RMA and P2P, are initialized.

At line 2, OpenCoarrays checks whether the performance and control variables for UMQ and eager protocol are available or not. This control is performed during the OpenCoarrays initialization, before executing any other program instruction. Because the variable names are not standardized, OpenCoarrays should know a-priori these names for all possible MPI implementations. Although this represents a strong restriction, it should be noted that OpenCoarrays recommends to use a small subset of all MPI implementations (MPICH and MPICH derivatives like MVAPICH2). Converting the recommendation into a requirement would have a tolerable negative impact on the user community but a great improvement from the performance standpoint.

If the variables associated with UMQ and eager protocol are available, OpenCoarrays decides a threshold for the maximum UMQ length and selects the P2P-events approach. Otherwise, RMA-events is selected and the library will not select any other approach during the execution.

At line 8, right after the initial selection, OpenCoarrays checks that every image made a consistent decision about which approach to use; assuming to have a private variable with value 0 or 1 associated with RMA-events or P2P-events, this control can be easily performed with a `MPI_Reduce`. Note that every image executes this statements during the initialization.

The code contained between line 9 and line 21 represents the program execution. The control statement ranging from line 10 to 20 is intended to happen inside the OpenCoarrays library, anytime an OpenCoarrays routine is invoked. At line 10, the library checks whether the P2P-events approach has been selected; if this is the case, it reads the performance variable associated with the UMQ length using the MPI_T interface and checks if its value is greater than the predefined threshold (line 11). In this case (lines 12-15), the image that has noticed the condition warns all the other images that it is switching to the RMA-events approach (switch_to_RMA message); this communication is performed with an `MPI_Put` routine. Then, it empties the UMQ queue from the events already posted and not yet read. Finally, it switches to the RMA-events approach and it will never change approach again.

At line 17, the images check whether any other image has sent a switch_to_RMA message or not. If this is the case, each image empties the UMQ queue from the events already posted and selects the RMA-events approach.

Note that the scope of this algorithm is to fall back to a "safe" implementation when the application puts too much pressure on the UMQ. Even though switching to the RMA mode when the UMQ reaches the limit for the first time may appear too restrictive, the overhead introduced by switching back and forth from P2P and RMA may have a serious impact on the overall performance and may also increase the risk of loosing synchronization messages.

4.2 P2P-events in OpenCoarrays

The idea behind P2P-events is to implement an `event post` using a `MPI_Send` for sending a short message composed by only two integers. The first integer represents a unique identifier associated with the event variable, common to any image; the second one represents an offset, meaningful only when the coarray variable is an array. The fact that the message is only two integers long and the presence of the eager control variable, ensure that the `MPI_Send` completes immediately without blocking.

Because a coarray variable must exist and be allocated on every image, it is easy to generate a unique identifier for each event variable. In our implementation, Image 1 is in charge of keeping a counter that is incremented by one every time an event variable is created. Right after the generation of the ID, Image 1 performs a `MPI_Bcast` in order to propagate the unique ID to every other image. Right after the broadcast, every image stores the unique ID into a structure.

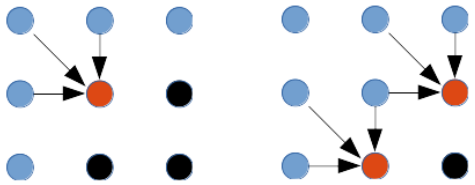
As we mentioned in Section 3.2, GNU Fortran manages coarray variables using: 1) a token used for accessing the remote coarray variable (it usually stores the MPI window); 2) a local variable used for common Fortran computation. An hypothetical implementation of P2P-event (without considering a RMA-based implementation), would store the structure containing the event ID into the token variable. Such a solution would generate a non-coherent situation with the other coarray variables that use the token as a MPI window.

In order to make the situation coherent and have both RMA- and P2P-event available, we decided to store the structure containing the unique ID and other information as a RMA window attribute attached to the MPI window used for the RMA-event implementation.

5. PARALLEL RESEARCH KERNELS

The Parallel Research Kernels (PRK) suite [12, 32, 31] focuses on providing a set of kernels that covers the most common patterns of communication, computation and synchronization encountered in parallel HPC applications. The suite is publicly available on GitHub[1] and currently provides parallel kernels written in a number of different programming models (OpenMP, MPI two-sided, MPI one-sided, MPI+OpenMP, UPC, SHMEM, Charm++, Grappa, Python, etc.).

[1]https://github.com/ParRes/Kernels



(a) Step 1 (b) Step 2

Figure 4: Instance of p2p_sync kernel

In order to show the potential of *events* and the impact of the different implementations, we decided to transform two kernels of the Parallel Research Kernels suite, already written in Fortran 2008 coarrays, to Fortran 2015.

5.1 Sync_p2p Kernel

The first kernel, called *sync_p2p*, implements a stencil code with a demanding data dependence that is typically resolved using a fine-grain software pipeline technique. A typical instance of this kernel is shown in Figure 4: in order to be computed, a component in position (i,j) requires data from the components in position (i-1,j), (i,j-1) and (i-1,j-1); as shown in Figure 4b, it is possible to compute in parallel several columns of the grid (pipeline among columns). A parallel example of this kernel is depicted in Figure 5, where Image 2 cannot start the computation on its second column because of the data dependency with Image 1. In this case, it is important to have a fine-grain synchronization mechanism capable to inform Image 2 that the data needed is ready to be taken.

In the Coarray Fortran 2008 (from now on CAF 2008) version already included in the PRK suite, the synchronization among images is implemented with `sync images` statements. This mechanism allows to the image that has invoked it to synchronize only with the set of images passed as argument. In a case like the one depicted in Figure 5 with 3 images involved, Image 2 would stop twice: one for synchronizing with Image 1 (where Image 2 is the "consumer") and one with Image 3 (where Image 2 is the "producer").

Events represent the most efficient mechanism for dealing with this sort of producer-consumer problems. The idea is to associate an event variable to each column of the grid; as soon as a "producer" (upper) image has completed the computation on its own column, it posts the event to the correspondent event variable on the "consumer" image. Because the `event post` routines is always non-blocking, the producer is free to continue the remaining computation. On the other hand, the "consumer" image waits for a single, specific, event related only to the data needed.

5.2 Stencil Kernel

The second kernel, called *stencil*, applies a data-parallel stencil operation to a two-dimensional array. It features multiple streams of regular but different strides on read, which benefits from efficient prefetching. This kernel represents one of the most common communication pattern in scientific computing: the halo exchange. In fact, many applications in high performance computing use domain decomposition techniques to distribute the work among different processing elements. To manage synchronization overheads, each decomposed domain is logically overlapped at the boundaries

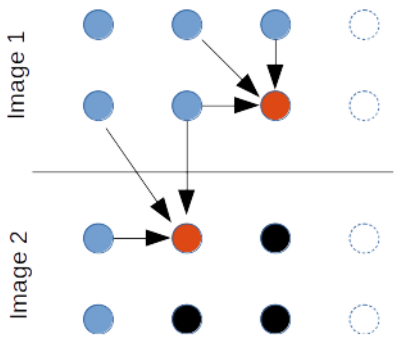


Figure 5: Parallel pipelined execution of p2p_sync kernel

and is updated with neighbor values before the computation proceeds. This update on the overlapped regions is called halo exchange.

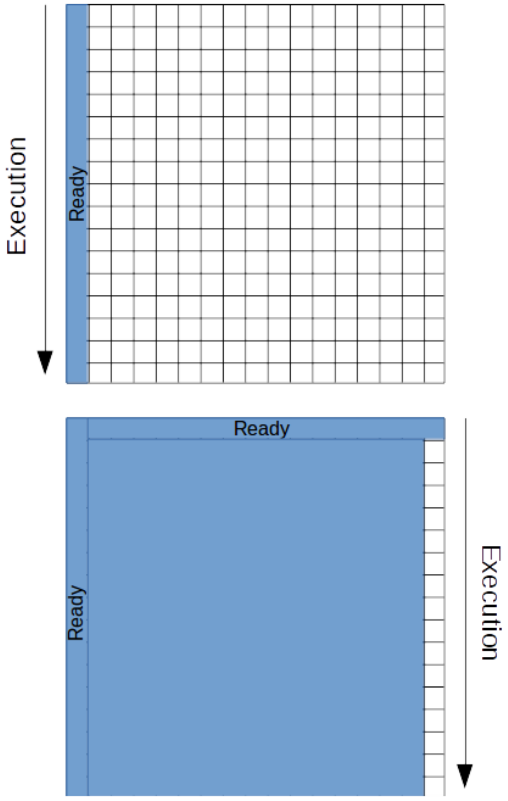


Figure 6: Data availability for halo exchange in stencil kernel

In the example reported in Figure 6, it is assumed that computation proceeds in a column-wise fashion; as soon as the first column has been computed, the halo exchange with the left neighbor can be performed. At the same way, as soon as the first row as been computed (after n-1 columns have been computed), the data is ready for the halo exchange with the upper neighbor.

Using CAF 2008, the only way to safely perform an halo exchange is to use a `sync all` statement or a `sync images` accompanied with a list of selected images, even though the data needed may be ready long before the synchronization

statements.

With *events* it is possible to completely remove any synchronization statement by looking at the algorithm as a produce/consumer problem. Assuming a "push" approach, where the process that produces the data pushes it to the neighbors' halo region, two event variables are needed for each side of the grid: one representing the data availability on the local halo region (from now on called *ready* variables) and another one representing the permission to overwrite the halo on the remote process (from now on called *consumed* variables).

During the data partitioning, the halo exchange is performed as part of the partitioning process and the events associated with the data availability are posted. At the beginning of the actual computation (usually a loop), the application performs an `event wait` on each *ready* variable. Once all the data is in the halo region (trivial for the first iteration) the computation starts. At the end of the computation, each image notifies the neighbors that the data contained in the halo region has been consumed by posting an event on the *consumed* coarray variables of the neighbors. Immediately after this notifications, each image waits on its own *consumed* variables. As soon as a halo region becomes available the data is sent using a coarray "put". Once the "put" has returned, the image posts an event on the *ready* variable of the correspondent neighbor.

The advantages brought by this approach are twofold: 1) the finer granularity of *events* allows to reduce idle times compared to an implementation based on *sync images*; 2) the "push" approach allows some computation/communication overlapping if the underlying network and MPI implementation support it (as in the case of Stampede).

6. EXPERIMENTAL PLATFORMS

Each reported test has been run on two supercomputers: Galileo and Stampede.

Galileo is a Tier-1 system operated by CINECA, the Italian supercomputing consortium. The compute nodes are equipped with two Intel®Xeon®E5-2630v3 (Haswell) processors (8 cores per chip, 2.40 GHz). The interconnect is Intel True Scale QDR InfiniBand⋆.

The TACC (Texas Advanced Computing Center) Stampede system is a 10 PFLOPS Dell⋆Linux⋆cluster based on 6400+ Dell PowerEdge server nodes, each outfitted with 2 Intel Xeon E5 (Sandy Bridge) processors. The nodes are connected with Mellanox⋆FDR InfiniBand technology in a 2-level (cores and leafs) fat-tree topology.

The results reported in Section 7 show how the underlying network influences the performance of the one-sided MPI routines.

7. RESULTS

In this section, we aim to show two kind of results: 1) how using *events* improves the performance of a parallel algorithms and 2) a performance comparison between the two approaches used for implementing *events* in OpenCoarrays. Note that the performance of Algorithm 1 has not been analyzed because the fallback from P2P to RMA can happen at most once.

The *sync_p2p* kernel is a communication latency bound problem: in the example depicted in Figure 5, computation on Image 2 cannot proceed unless Image 1 reaches the bottom of its grid. Figures 7 and 8 show the results of a weak scaling study which compares a version based on `sync images` with one based on *events*. It is clear that, on both machines, the approach based on `sync images` (called SYNC) has always lower performance than the P2P_EV approach based on events. Furthermore, the charts show the remarkable performance difference between RMA-events and P2P-events.

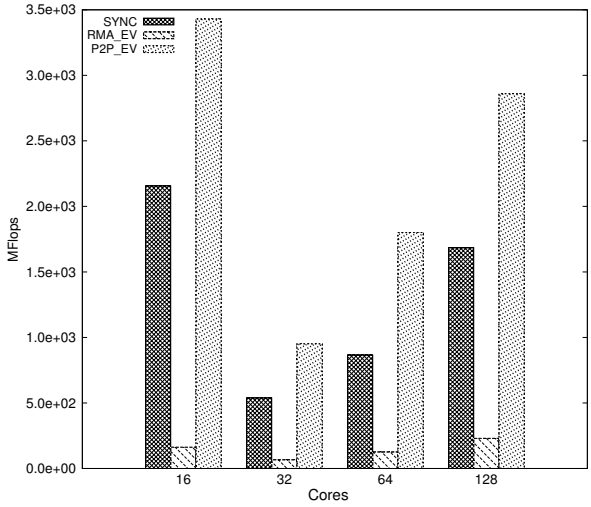


Figure 7: Sync_p2p kernel performance on Galileo

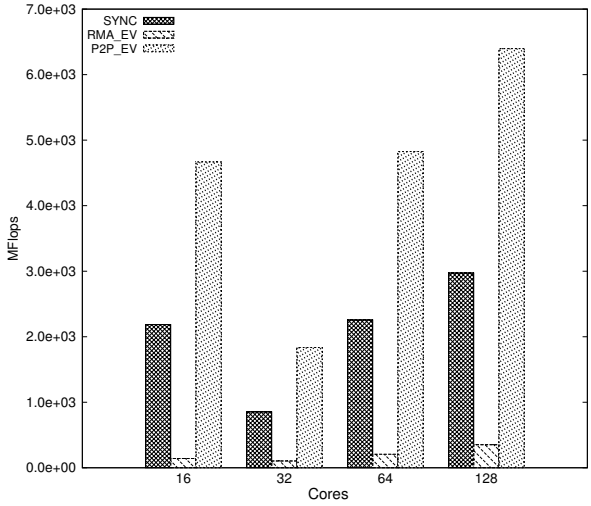


Figure 8: Sync_p2p kernel performance on Stampede

The *stencil* kernel described in Section 5.2 can get great benefits from the fine grain synchronization mechanism brought by *events*, in particular when the network interconnect provides native support for Remote Direct Memory Access (RDMA). Figures 9 and 10 show clearly how the P2P-events strategy leads to better results for any number of cores, on both supercomputers. Furthermore, the Mellanox Infiniband network installed on Stampede provides support for RDMA, allowing to hide communication costs during the "put" operation, with a resulting increase in overall performance.

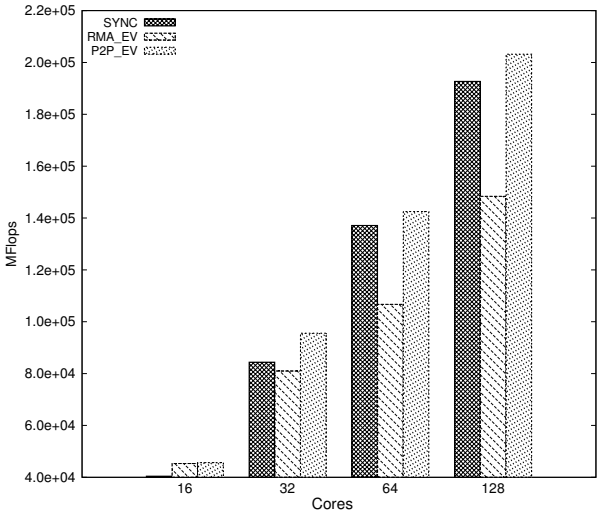


Figure 9: Stencil kernel performance on Galileo

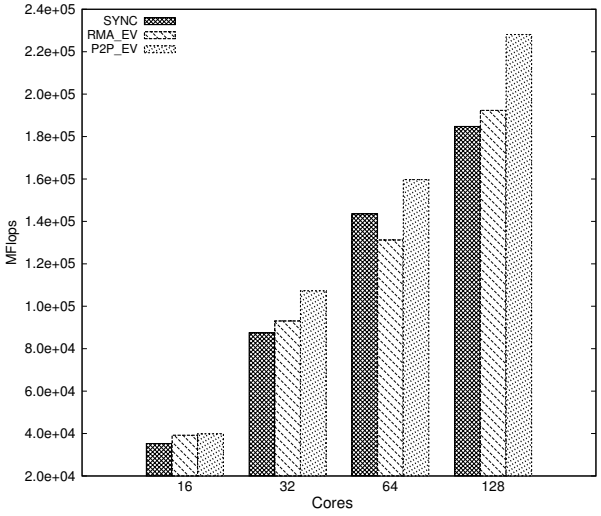


Figure 10: Stencil kernel performance on Stampede

8. RELATED WORK

The implementation of PGAS programming models using MPI RMA has been an active research and development activity, starting with ARMCI-MPI [13], which initially targeted MPI-2 but now supports MPI-3. Subsequent efforts targeted the more flexible MPI-3 semantics and included ports of OpenSHMEM [17] and Coarray Fortran [16, 20]. The recently developed DASH programming model also targets MPI-3 RMA as its communication runtime [36].

The Parallel Research Kernels have been used to evaluate the efficiency of one-sided communication for point-to-point communication in the context of proposed extensions to OpenSHMEM [14] and MPI RMA [4]. Recent studies have used the PRK suite to evaluate PGAS programming models [32] and a more general set of programming models [31]; these implementations provide a means for comparing Fortran coarrays to other PGAS models (SHMEM and UPC) as well as various forms of MPI.

9. CONCLUSIONS

In this work, we have shown how the new features introduced by the standard MPI-3 can be used by a PGAS communication library to implement fine grain synchronization mechanisms like CAF *events*. In particular, we have shown how to use the *control/performance variables* exposed by the MPI implementation (accessible through the MPI Tool Information Interface) to implement a synchronization mechanism based on the Unexpected Message Queue. This implementation of *events* has reported the best results for all tests, on every platform.

We have also shown the potential benefits brought by *events* and the impact of their implementation on the overall performance of typical scientific kernels.

10. ACKNOWLEDGMENTS

We gratefully acknowledge the support we received from the following institutions: CINECA for the access on Galileo for the project OpenCoarrays under the ISCRA grant program for 2016. The Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575, for the access on Stampede.

11. REFERENCES

[1] Cray XC series network. Technical Report WP-Aries01-1112, Cray, 2012.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The Fortress language specification. Technical report, Sun Microsystems, Inc., March 2008. Version 1.0.

[3] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. *High-Performance Interconnects, Symposium on*, 0:83–87, 2010.

[4] R. Belli and T. Hoefler. Notified access: Extending remote memory access programming models for Producer-Consumer synchronization. In *IPDPS*, Hyderabad, India, May 2015.

[5] D. Bonachea and J. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004.

[6] R. Brightwell and K. D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 298–305, New York, NY, USA, 2004. ACM.

[7] V. Cardellini, A. Fanfarillo, and S. Filippone. Overlapping communication with computation in MPI applications. Technical Report DICII RR-16.09, Università di Roma Tor Vergata, Feb. 2016. http://hdl.handle.net/2108/140530.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.

[10] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, Nov 2011.

[11] J. Daily, A. Vishnu, B. Palmer, H. van Dam, and D. Kerbyson. On the suitability of MPI as a PGAS runtime. In *21st International Conference on High Performance Computing (HiPC), 2014*, pages 1–10, Dec 2014.

[12] R. F. V. der Wijngaart and T. G. Mattson. The parallel research kernels. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, Sept 2014.

[13] 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*, pages 739–750, May 2012.

[14] J. Dinan, C. Cole, G. Jost, S. Smith, K. Underwood, and R. W. Wisniewski. Reducing synchronization overhead through bundled communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 163–177. Springer, 2014.

[15] T. H. Dunigan, Jr., M. R. Fahey, J. B. White III, and P. H. Worley. Early evaluation of the Cray X1. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 18–, New York, NY, USA, 2003. ACM.

[16] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. OpenCoarrays: Open-source transport layers supporting coarray Fortran compilers. In *PGAS*, PGAS '14. ACM, Oct. 2014.

[17] J. R. Hammond, S. Ghosh, and B. M. Chapman. *Implementing OpenSHMEM Using MPI-3 One-Sided Communication*, pages 44–58. Springer International Publishing, Cham, 2014.

[18] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. D. Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface*, volume 6305 of *LNCS*, pages 50–61. Springer, 2010.

[19] T. Hoefler and A. Lumsdaine. Message progression in parallel computing - to thread or not to thread? In *Proc. of 2008 IEEE Int'l Conf. on Cluster Computing*, pages 213–222, Sept. 2008.

[20] Intel. Distributed memory coarray Fortran with the Intel Fortran compiler for Linux: Essential guide, Nov. 2014.

[21] ISO/IEC/JTC1/SC22/WG5. TS 18508 additional parallel features in Fortran, Aug. 2015.

[22] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a new dimension for Cray Research. In *Compcon Spring '93, Digest of Papers.*, pages 176–182, Feb 1993.

[23] M. Metcalf, J. Reid, and M. Cohen. *Modern Fortran Explained*. Oxford University Press, Inc., New York, NY, USA, 4th edition, 2011.

[24] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[25] R. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[26] R. W. Numrich and J. Reid. Co-arrays in the next Fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, Aug. 2005.

[27] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, Jan. 2002.

[28] S. L. Scott and et al. The Cray T3E network: Adaptive routing in a high performance 3D torus, 1996.

[29] M. Si, A. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An asynchronous progress model for MPI RMA on many-core architectures. IPDPS '15, pages 665–676, May 2015.

[30] UPC Consortium. UPC language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[31] R. F. Van der Wijngaart, A. Kayi, J. R. Hammond, G. Jost, T. St. John, S. Sridharan, T. G. Mattson, J. Abercrombie, and J. Nelson. Comparing runtime systems with exascale ambitions using the Parallel Research Kernels. In *ISC High Performance*, pages 321–339, 2016.

[32] R. F. Van der Wijngaart, S. Sridharan, A. Kayi, G. Jost, J. R. Hammond, T. G. Mattson, and J. E. Nelson. Using the Parallel Research Kernels to study PGAS models. In *PGAS*. IEEE, 2015.

[33] A. Vishnu, J. Daily, and B. Palmer. Designing scalable PGAS communication subsystems on Cray Gemini interconnect. In *19th International Conference on High Performance Computing (HiPC), 2012*, pages 1–10, Dec 2012.

[34] A. Vishnu, D. Kerbyson, K. Barker, and H. van Dam. Building scalable PGAS communication subsystem on Blue Gene/Q. In *IPDPSW*, pages 825–833, May 2013.

[35] A. Vishnu and M. Krishnan. Efficient on-demand connection management mechanisms with PGAS models over InfiniBand. In *CCGrid*, pages 175–184, May 2010.

[36] H. Zhou, K. Idrees, and J. Gracia. Leveraging MPI-3 shared-memory extensions for efficient PGAS runtime systems. In *Euro-Par*, pages 373–384, 2015.