Notified Access in Coarray Fortran

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

Davide Del Vento
National Center for Atmospheric Research
Boulder, Colorado
ddvento@ucar.edu

ABSTRACT

With the increasing availability of the Remote Direct Memory Access (RDMA) support in computer networks, the so called Partitioned Global Address Space (PGAS) model has evolved in the last few years. Although there are several cases where a PGAS approach can easily solve difficult message passing situations, like in particle tracking and adaptive mesh refinement applications, the producer-consumer pattern, usually adopted in task-based parallelism, can only be implemented inefficiently because of the separation between data transfer and synchronization (which is usually unified in message passing programming models).

In this paper, we provide two contributions: 1) we propose an extension for the Fortran language that provides the concept of Notified Access by associating regular coarray variables with event variables. 2) We demonstrate that the MPI extension proposed by foMPI for Notified Access can be used effectively to implement the same concept in a PGAS run-time library like OpenCoarrays.

CCS CONCEPTS

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

KEYWORDS

MPI, PGAS, Coarray, Notified Access

ACM Reference format:
Alessandro Fanfarillo and Davide Del Vento. 2017. Notified Access in Coarray Fortran. In *Proceedings of EuroMPI/USA '17, Chicago, IL, USA, September 25–28, 2017,* 7 pages.
DOI: 10.1145/3127024.3127026

1 INTRODUCTION

In recent years computer architectures have changed significantly: heterogeneous hardware, high level of parallelism, higher failure rates and the lack of globally cache coherent systems will compel users to write parallel applications in a more flexible way. In this environment, task-based parallelism can be very effective because it allows the programmer to cope efficiently with heterogeneous hardware, dynamic workloads and failures.

The producer-consumer communication pattern is widely adopted in high performance parallel applications using any form of halo exchange or task-based parallelism. All producer-consumer communications requires two basic steps: 1) data transmission and 2) synchronization. In the message passing model, both steps are provided by the receive operation. Remote Memory Address (RMA) programming schemes, which include most Partitioned Global Address Space (PGAS) languages, separate data transfer and synchronization into different primitives. For the producer-consumer communication pattern, this approach is inefficient because it requires at least three message transactions on the critical path. This inefficiency has been deeply analyzed and tackled by Belli and Hoefler in [2]; we report their finding in Section 2.

To work around this inefficiency, we propose to add the concept of Notified Access provided by [2] in coarray Fortran (CAF) by associating a coarray event variable to a regular data coarray variable. This approach would provide minimal impact on the already existing Fortran 2015 features and it would not be complex to use for end users. In Section 3 we explain why *events* are suitable for implementing Notified Access in coarray Fortran and how their implementation in OpenCoarrays can be easily adapted to fit the Notified Access requirements.

As a proof of concept, we implemented the Notified Access in OpenCoarrays [7] on top of the strawman Interface for Notified Access for MPI proposed by Belli and Hoefler in [2] provided in *foMPI_NA*. The implementation of Notified Access in OpenCoarrays on top of *foMPI_NA* will be described in Section 4.

Then, in Section 5, we compare the performance of the Fortran Notified Access implementation we are proposing in this paper with the regular Fortran 2015 events, for a pipelined stencil called *Sync_P2P* (from the Intel Parallel Research Kernels [5]). We also compare the performance of our Fortran Notified Access with regular Fortran 2015 events and MPI two-sided version of a regular stencil kernel applied on a Structured Grid, which represents the common *halo exchange* communication pattern. We show that our implementation of Notified Access in Coarray Fortran based on *foMPI_NA* provides good performance.

Finally, in Section 7, we report our conclusions and future work.

2 FOMPI WITH NOTIFIED ACCESS

In [2], Belli and Hoefler examine the inefficiencies of the one-sided communication functions when applied on the producer-consumer communication pattern. Many producer-consumer communication patterns need one message for synchronization, which implies additional network transactions. For example, for a remote put there will be one message for the actual data transfer, one for communicating the remote completion and one for the explicit synchronization. For a remote get, there will be two messages for the actual data transfer and one for synchronization. See [2] for details.

To tackle this inefficiency, Belli and Hoefler propose to extend the RMA programming models with a new mechanism called *Notified Access* which allows the target process to detect when a transfer is completed without additional messages.

Notified Access adds a remote completion notification to any remote access. The target process can use this notification for synchronizing local or remote accesses to the buffer. The interpretation of the notification depends on the action: if the notified access is a read then the notification indicates that the data was copied and the buffer can be overwritten; if the notified access is a write then the notification indicates that the data was committed to memory and can be read. The origin can mark accesses with a notification or without, i.e., not all accesses have to trigger a remote notification.

While Notified Access is independent of a particular programming model, Belli and Hoefler propose an interface for the Message Passing Interface (MPI). They implemented the new interface for Notified Access using the open source foMPI (Fast One Sided MPI) [9] which supports the full MPI-3.0 One Sided interface. The foMPI library is based on Cray DMAPP [1, 19] and XPMEM [23] APIs for inter- and intra-node communications, respectively.

The extended foMPI-NA [1] uses the uGNI API [1] that provides direct access to Cray's Fast Memory Access (FMA) and Block Transfer Engine (BTE) mechanisms. Using both mechanisms it is possible to directly notify the completion of a RDMA operation to the target process.

2.1 MPI Interface for Notified Access

To extend MPI with Notified Access, Belli and Hoefler introduce a notified variant for each communication operation in MPI RMA. Each new function has an additional integer tag argument. In Listing 1 we show the C interface for `MPI_Put_notify`.

Listing 1: Interface MPI_Put_notify

```
int MPI_Put_notify(void *origin_addr, int origin_count,
             MPI_Datatype origin_type, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_type, MPI_Win win,
             int tag)
```

MPI request objects are used for notification at the target side. The requests are initialized explicitly with the function `MPI_Notify_init` and are not automatically freed.

Listing 2: Interface MPI_Notify_init

```
int MPI_Notify_init(MPI_Win win, int src_rank, int tag,
                    int expected_count,
                    MPI_Request *request)
```

`MPI_Notify_init` initializes a request for notification and binds it to a specific MPI window with notification count, tag and source. The returned MPI request object can be used with the usual MPI test and wait functions. A request completes after `expected_count` matching notified accesses have been performed. Matching is performed in order and it is defined through source and tag and the wild-cards `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are supported. If a request is completed, the returned MPI status object includes the information of only the last matching notified access. In Listing 2 we show the C interface for `MPI_Notify_init`.

The requests initialized with `MPI_Notify_init` should be freed with `MPI_Request_free` because they are intended to be *persistent requests*. Before each use, requests have to be started with `MPI_Start`. Once started, request completion can be progressed and completed with the normal test and wait operations.

Because many of today's networks do not support hardware message matching, the matching mechanism for notified access is implemented in software. Note that the data movement is still fully performed in hardware and only the processing of the light-weight notification in software.

[1] Available at http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI_NA/

3 FORTRAN 2015

Coarray Fortran (also known as CAF) is a set of features of the Fortran 2008 standard (ISO/IEC 1539-1:2010) [16] that make Fortran a Partitioned Global Address Space programming language.

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 [13] proposes, among several new extensions to the coarray facilities defined in Fortran 2008, *Events*.

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. All the features defined in TS-18508 have been approved by the standard committee and will be part of the Fortran 2015 standard.

3.1 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 `event post` statement; this routine 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` statement; this blocking routine can be invoked only on local variables. The statement takes two arguments: 1) the event variable and 2) the number of events to wait for. Once the routine returns the event variable is set to zero.

Since an image may want to check the value of a local event variable without waiting, an `event_query` statement is also provided. This routine takes two arguments: 1) the event variable and 2) the number of events that have been posted so far, which will be the output of the routine. When the routine returns, the internal counter of the event variable is never changed. An example of `event wait` and `event_query` is shown in Listing 3.

Listing 3: Event wait and query

```
  event wait(ev,until_count = count)
  call event_query(ev,count = count)
```

The main difference between *events* and the general semaphores stands in the local applicability of the `event wait` and `event_query` routines; this restriction makes events safer, easier to use and highly performing.

3.2 OpenCoarrays

OpenCoarrays [7] 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" [13]. 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 hard-wiring 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*, the collective/reduction and new *atomic* intrinsics belonging to the Fortran 2015 standard and an experimental version of *failed images*.

Since the first release of OpenCoarrays (August 2014), the widest coverage of coarray features was provided by a MPI-3 based run-time library (LIBCAF_MPI). Because of the one-sided nature of coarrays, the vast majority of the run-time library uses MPI-3 one-sided communication routines based on passive synchronization [11].

Because foMPI routines differ from the standard MPI routines only for the prefix library name (e.g. `MPI_Win_create` vs. `foMPI_Win_create`), it is straightforward to transform the MPI version of OpenCoarrays in foMPI.

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 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.

4 IMPLEMENTING NOTIFIED ACCESS USING EVENTS

The concept of Notified Access described in Section 2, can be implemented in coarray Fortran by binding an event variable with the coarray variable that contains the data.

For a coarray "put" operation, every time the coarray variable gets updated by a remote process using a "put", an `event post` gets automatically triggered on the associated event variable to communicate that the variable got updated. For a coarray "get" operation, every time the coarray variable is read by a remote process using a "get", an `event post` gets triggered on the associated event variable to communicate that the variable has been read and it can be safely overwritten. Because the coarray "put" operation provides the highest opportunity for overlapping communication with computation, in this work we will focus only on the notified "put" operation. In fact, the Fortran syntax for "put" does not require the actual data movement to be completed before the end of the segment, whereas for get the data movement must be completed before the call returns. However we think that notified access for coarray "get" operations has potential and we plan to explore this solution in a future work.

Because the Fortran 2015 standard does not provide a way to connect an event with a regular coarray variable, an easy-to-use extension would be the `event attach` routine. A possible signature of this routine is shown in Listing 4.

Listing 4: Proposed event attach signature

```
subroutine event attach (event_variable,coarray_variable,
                         [stat=statvar])
```

The `coarray_variable` represents the coarray variable that contains the data whereas `event_variable` represents the event variable that should be signaled when an image performs a remote access on `coarray_variable` using a "put".

Because a coarray variable can be a scalar or array, static or dynamic, and of intrinsic or derived type, the `event attach` routine should also allow the user to specify a certain range of the coarray variable to bind with an event. For the purpose of this paper we provide motivation and proof-of-concept for an implementation of Notified Access in CAF. The complete definition of the semantic of the `event attach` subroutine is beyond the scope of this paper and left as a future work.

4.1 Implementation in OpenCoarrays using foMPI_NA

The first step to take in order to implement Notified Access in OpenCoarrays is writing the `event attach` function. As shown in Listing 4, the function takes two compulsory arguments and one optional. The function inserts the coarray data and event variables in a linked list in order to keep track of this association, using the structure shown in Listing 5. The `notified_ev` field is initialized to NULL and it is needed to keep track of the `foMPI_Request` associated with the notification event. The function `event attach` is local, non-blocking and does not require any communication.

Listing 5: Attached events structure

```
  struct attached_events
  {
    caf_token_t ev;
    caf_token_t var;
    void *notified_ev;
    struct attached_events *next;
  };
```

Secondly, the `event wait` and `event_query` routines must be adapted to the new mechanism. The `event wait` statement takes two arguments: 1) the event variable and 2) the number of events to wait for. Because foMPI_NA requires to declare the number of notifications to receive in the `foMPI_Notify_init` function (see Listing 2, argument `expected_count`), and performs the wait using the regular `foMPI_Wait` function, we implemented initialization AND wait directly inside `event wait`. In case a notification gets posted before the invocation of `foMPI_Notify_init` on the destination, foMPI_NA is able to take care of the unexpected notifications. In Listing 6 we report an example of how `event wait` can be implemented using the Notified Access support provided by foMPI_NA.

Listing 6: Event wait based on NA

```
if(var_attached)
{
  notified_ev = (foMPI_Request *)
                    malloc(sizeof(foMPI_Request));
  tmp_ev->notified_ev = notified_ev;
  foMPI_Notify_init(*var_attached, foMPI_ANY_SOURCE,
                   ev_id, until_count, notified_ev);
  foMPI_Start(notified_ev);
  foMPI_Wait(notified_ev, &status);
  foMPI_Request_free(notified_ev);
}
else
{
  // Regular event wait implementation here
}
```

Currently, in OpenCoarrays regular events are implemented in two alternative ways: 1) using the MPI RMA atomic operations; 2) using point-to-point MPI operations and the unexpected message queue. In [8], Fanfarillo and Hammond describe pros and cons of the two approaches and show the performance differences. Both approaches are incompatible with the implementation of `event wait` based on Notified Access (NA) and thus the library must provide support for both, NA-based and non NA-based solutions, as shown in the else branch of Listing 6.

This coexistence of different implementations for `event wait` and `event query` introduces a limitation: the event cannot be managed as a regular event variable AND a notified access request at the same time. Namely, it is not possible to perform an `event post` on an event attached to a variable. However, arguably this limitation is not really a problem, and in fact forces the programmer to a clearer separation of concerns. Should explicit "event post"s be needed, they should use a separate event variable and not the same that is posted automatically with Notified Access.

5 RESULTS

Each of the following test has been run on Swan, a small Cray XC40 composed by a variety of compute nodes; for the purpose of our investigation we used the nodes equipped with 2 Broadwell 22-core Intel Xeon at 2.2 GHz with 128 GB of RAM DDR4-2400. For all the test cases we allocate 32 images on each compute node.

5.1 Sync_P2P Kernel

The first kernel, called *sync_p2p*, is taken from the Parallel Research Kernels (PRK) suite [5, 21, 22]. The suite 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[2] is publicly available on GitHub and

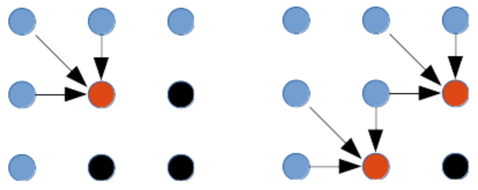

Figure 1: Instance of p2p_sync kernel

currently provides parallel kernels written in a number of different programming models (OpenMP, MPI two-sided, MPI one-sided, MPI+OpenMP, CAF, UPC, SHMEM, Charm++, Grappa, Python, etc.).

The kernel *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 1: 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 1b, 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 2, 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, lightweight synchronization mechanism capable to inform Image 2 that the data needed is ready.

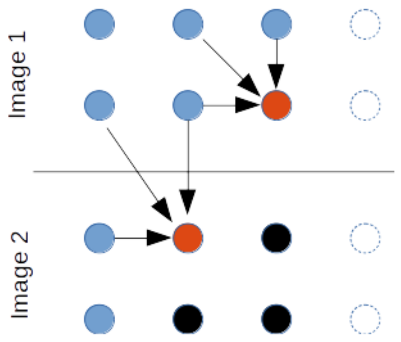

Figure 2: Parallel pipelined execution of p2p_sync kernel

In the 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 2 but 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.

In [8], Fanfarillo and Hammond convert the Fortran 2008 version of the kernel to Fortran 2015, associating 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

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

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.

In this work, we slightly modify the Fortran 2015 version of *sync_p2p* by attaching the event variable related with a column of the grid with the column itself.

The modification results in a simpler, more clear and less error prone source code, because the explicit event posts are removed, and therefore cannot be misunderstood, forgotten or misplaced by an inexperienced programmer. In order to test the performance of the new approach, we run a strong scaling test on a fixed grid size of 32768x32768 elements. We tested the approach based on Notified Access and the regular Fortran 2015 version based on *events* using 4 different event implementations.

In Figure 3, the Notified Access bars represent the new mechanism proposed in this paper and described in Section 4, which has been implemented on top of *foMPI_NA*. The *P2P* bars represent the point-to-point algorithm presented in [8] to implement *events* on top of MPI two-sided and the Unexpected Message Queue. *RMA foMPI* represents an event implementation based on the MPI RMA atomic operations provided by foMPI_NA. *RMA* and *RMA_AMO* represent an event implementation based on the MPI RMA atomic operations provided by the standard Cray MPI implementation based on DMAPP (MPICH-7.5.3). For *RMA*, *RMA_AMO* and *P2P*, we set the environmental variable `MPICH_RMA_OVER_DMAPP` equal to 1 in order to use the DMAPP version for RMA. This setting ensures asynchronous progress for the MPI RMA operations using a thread-based approach.

For the *RMA_AMO* we set the environmental variable `MPICH_RMA_USE_NETWORK_AMO` to 1. This variable set the use of network Atomic Memory Operations (AMOs) for selected MPI operations (like the ones used to implement *events*).

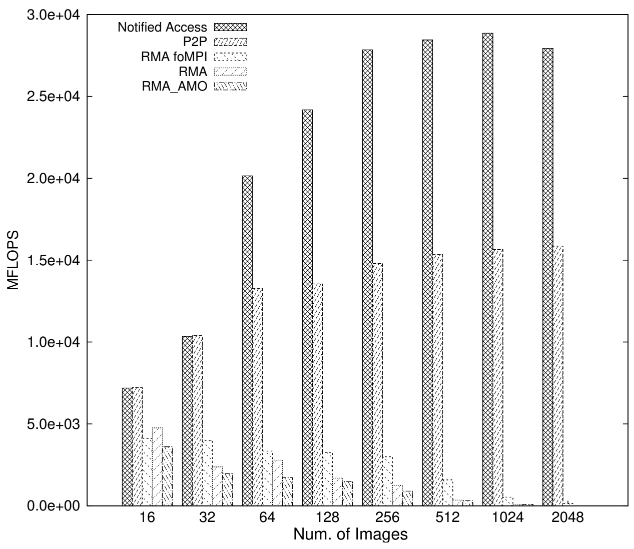

Figure 3: Performance of p2p_sync kernel

For this particular test case, which exposes a producer-consumer communication pattern, the solution based on Notified Access is always better than any other mechanism, dramatically at large scale. This is the case because the kernel is synchronization bound and the *foMPI_NA* library is very effective in limiting the overhead of the synchronizations. Because of the semantics of Notified Access, it would be impossible to exploit these benefits without extending the Fortran language. The performance achieved by our prototype demonstrates that the proposed *event attach* Fortran extension could effectively use *foMPI_NA* to improve the performance of the synchronizations, as shown in Figure 3.

5.2 Structured Grid Kernel

Structured grids problems are very common in scientific computing. Data is arranged in a regular multidimensional grid (most commonly 2D or 3D), and the computation proceeds as a sequence of grid update steps. At each step, all points are updated using values coming from a small neighborhood around each point (stencil). Each processor can be statically assigned to a contiguous subgrid, and can perform each update step locally and independently of other nodes. Each node only has to communicate and synchronize with neighboring nodes on the grid, exchanging data from the boundary of their sub-grids at the end of each step. This kernel represents one of the most common communication pattern in scientific computing: the halo exchange. To manage synchronization overheads, each decomposed domain is logically overlapped at the boundaries and is updated with neighbor values before the computation proceeds. This update on the overlapped regions is called halo exchange.

Because the structured grid is usually uniformly distributed across the images, the computation takes almost the same time on all processes. In this situation, the message passing programming model leads to good performance because the synchronization among processes is implicit in the halo exchange, which occurs almost at the same time because of the homogeneous data partitioning. PGAS languages, and in particular CAF, are penalized because they require explicit synchronization calls at the end of the communication phase.

The real power of PGAS languages relies on the one-sided semantics and the possibility of overlapping communication with computation. In our test case, a global 2D grid is partitioned among processes assigning columns as uniformly as possible; the halo exchange is required only for the right and left neighbors. The computation is performed in a column-wise fashion starting from the leftmost column. Using CAF, it is possible to send the first column to the left neighbor with a put operation while computing the remaining columns. The synchronization can be implemented using an event variable for the left and right halo region on each process. Listing 7 shows how this overlapping can be achieved using *events*.

Listing 7: Structured Grid using Events

```
do i=1,NR
T(i,1) = 0.25 * ( Told(i+1,1)+Told(i-1,1)+
                  Told(i,2)+Told(i,0) )
enddo

right_halo(:)[prev] = T(:,1)

do j=2,NCL
```

```
    do i=1,NR
    T(i,j) = 0.25 * ( Told(i+1,j)+Told(i-1,j)+
                      Told(i,j+1)+Told(i,j-1) )
    enddo
  enddo

  event post(ready_right[prev])
  left_halo(:)[next] = T(:,NCL)
  event post(ready_left[next])
```

In this scenario, Notified Access are not as critical as for the previous test case, but they can still produce results that are competitive with MPI two-sided, even for this particular communication/computation pattern, which is very well suited for MPI two-sided.

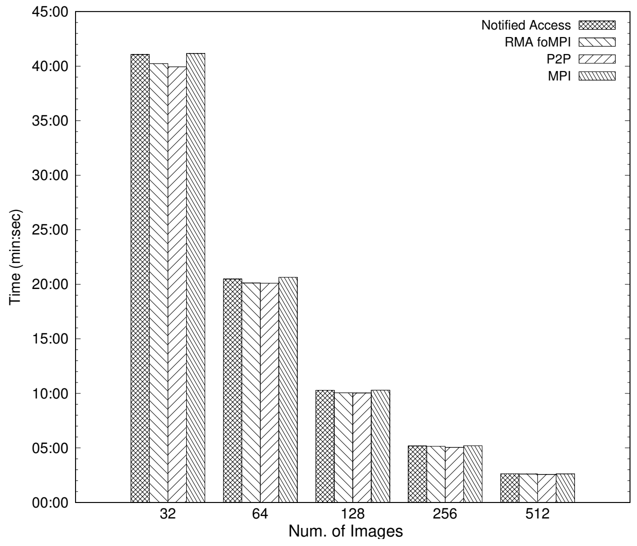

Figure 4: Performance of Structured Grid kernel

In Figure 4, we show the time spent for applying a five-point stencil on a square grid with side 65536 points wide, using coarray with Notified Access (Notified Access), coarray with events using the MPI RMA provided by foMPI (RMA foMPI), coarray with events implemented on top of MPI two-sided and Unexpected Message Queue (P2P) and pure MPI two-sided (MPI). In the latter case, there is no opportunity to overlap communication with computation because the data transfer is performed at the end of the computation phase using MPI_ISend and MPI_IRecv routines. The effect of the overlapping is much more evident when the number of cores is small (more time spent in computation). The version based on coarrays, implemented on top of the RMA atomic operations of foMPI provides the best performance.

The Notified Access performance is penalized because the operation itself takes more time that a regular "put" operation and the benefit brought by the overlapping gets hidden by the higher cost imposed by the function. Furthermore, the performance of foMPI using XPMEM strongly depends on the number of processes allocated on the same node.

6 RELATED WORK

The concept of put-with-notify and its potential for dealing with producer-consumer problems, using PGAS languages, has been studied for years.

Jose el al. [14] propose and implement extensions to OpenSHMEM [4], such as non-blocking put, and non-blocking put-with-notify. Similarly, Dinan et al. [6] propose a SHMEM extension that utilizes capabilities present in most high performance interconnects (e.g. communication events) to bundle synchronization information together with communication operations.

Unified Parallel C (UPC) [20] is another PGAS parallel programming model, that provides capabilities similar to SHMEM. The current UPC language provides similar synchronization routines as SHMEM, with the addition of split- phase barriers and locks. In [3], Dan Bonachea proposes the concept of semaphores to UPC and defines the `upc_memput_signal` and `upc_memput_signal_async` to implement the concept of put-with-notify.

The Aggregate Remote Memory Copy Interface library (ARMCI) [15] also provides a put-with-flag operation, that attaches a flag variable update with data transfer. In this mechanism, the origin of a remote write waits for the completion of the write and then notification message. This approach delays the notification to the target process for a round trip time of the network. Similarly, the GASPI [10, 18] PGAS library provides a write-and-notify operation, that bundles an event notification with data movement. In both cases, the notification is performed by a write, rather than an atomic update. Thus, for algorithms that require many synchronizations, many flag and event variables would be needed.

LAPI [17] provides a special data type called *counter variable*. A counter variable can be used to count message completion at the target. LAPI also provides a function to wait until the value of the counter reaches a specified value. The GET and PUT functions of LAPI have two arguments for the counters, one is a remote counter for target completion, and the other is a local counter for initiator completion.

Hori et al. [12] propose to associate a synchronization flag with memory regions. A notification on the flag gets triggered automatically when the memory region associated with it gets accessed. Although very promising, this approach cannot be supported efficiently on most of today's RDMA networks.

7 CONCLUSIONS AND FUTURE WORK

In this paper we show how the Notified Access mechanism proposed in [2] can be implemented in coarray Fortran and how it can benefit certain communication patterns like producer-consumer.

In order to implement Notified Access in CAF, we decided to introduce one simple language extension called `event attach` able to connect a coarray variable containing data with a coarray event variable. By doing so, every time the coarray data variable gets accessed by a remote process, an event gets automatically posted on the associated event variable. The definition of the `event attach` has been left incomplete because the focus of this paper is more on the potential of Notified Access and their implementation. We plan to explore a good definition of `event attach` in a future work.

For a synchronization bound test case like *sync_p2p* (described in Section 5.1), the implementation of Notified Access in CAF based on the strawman MPI interface proposed in [2] and implemented in *foMPI_NA*, provides the best performance compared to all the other event-based alternatives.

For a test case like the structured grid stencil described in Section 5.2, the implementation of Notified Access is penalized because the operation itself takes more time that a regular "put" operation, and the benefit brought by the communication/computation overlapping gets hidden by the higher cost imposed by the function. Even in this scenario where the traditional message passing model is well suited, Notified Access provides competitive performance.

We have noticed that the performance of data transfer and notification, based on XPMEM for shared memory in foMPI, are sometimes influenced by the number of processes allocated on the compute node. Currently, there are two possible alternatives already implemented in foMPI: one based on the usual `memcpy` operation and another based on SSE instructions. As a future work, we plan to explore other ways to perform the memory copy using XPMEM, like using non-temporal store instructions or optimizing the copy for a specific architecture.

ACKNOWLEDGMENTS

The authors would like to thank Cray Inc. for providing system resources to carry out this research.

REFERENCES

[1] 2013. *Using the GNI and DMAPP APIs.* Technical Report S-2446-5002. Cray. http://docs.cray.com/books/S-2446-5002/S-2446-5002.pdf

[2] R. Belli and T. Hoefler. 2015. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *2015 IEEE International Parallel and Distributed Processing Symposium.* 871–881. DOI: https://doi.org/10.1109/IPDPS.2015.30

[3] D Bonachea. 2012. Proposal for extending the UPC libraries with explicit point-to-point synchronization support. *Lawrence Berkeley National Lab, Tech. Rep* (2012).

[4] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10).* ACM, New York, NY, USA, Article 2, 3 pages. DOI: https://doi.org/10.1145/2020373.2020375

[5] R. F. Van der Wijngaart and T. G. Mattson. 2014. The Parallel Research Kernels. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE.* 1–6. DOI: https://doi.org/10.1109/HPEC.2014.7040972

[6] James Dinan, Clement Cole, Gabriele Jost, Stan Smith, Keith Underwood, and Robert W. Wisniewski. 2014. *Reducing Synchronization Overhead Through Bundled Communication.* Springer International Publishing, Cham, 163–177. DOI: https://doi.org/10.1007/978-3-319-05215-1_12

[7] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. 2014. OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers. In *PGAS (PGAS '14).* ACM, Article 4, 11 pages.

[8] Alessandro Fanfarillo and Jeff Hammond. 2016. CAF Events Implementation Using MPI-3 Capabilities. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016).* ACM, New York, NY, USA, 198–207. DOI: https://doi.org/10.1145/2966884.2966916

[9] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13).* ACM, New York, NY, USA, Article 53, 12 pages. DOI: https://doi.org/10.1145/2503210.2503286

[10] Daniel Grünewald and Christian Simmendinger. 2013. The GASPI API specification and its implementation GPI 2.0. In *7th International Conference on PGAS Programming Models*, Vol. 243.

[11] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.* 2, 2, Article 9 (June 2015), 26 pages. DOI: https://doi.org/10.1145/2780584

[12] A. Hori, J. Lee, and M. Sato. 2011. Audit: New Synchronization for the GET/PUT Protocol. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum.* 747–753. DOI: https://doi.org/10.1109/IPDPS.2011.217

[13] ISO/IEC/JTC1/SC22/WG5. 2015. TS 18508 Additional Parallel Features in Fortran. (Aug. 2015).

[14] Jithin Jose, Sreeram Potluri, Hari Subramoni, Xiaoyi Lu, Khaled Hamidouche, Karl Schulz, Hari Sundar, and Dhabaleswar K. Panda. 2014. Designing Scalable Out-of-core Sorting with Hybrid MPI+PGAS Programming Models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14).* ACM, New York, NY, USA, Article 7, 9 pages. DOI: https://doi.org/10.1145/2676870.2676880

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

[16] Robert W. Numrich and John Reid. 2005. Co-arrays in the Next Fortran Standard. *SIGPLAN Fortran Forum* 24, 2 (Aug. 2005), 4–17. DOI: https://doi.org/10.1145/1080399.1080400

[17] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. 1998. Performance and experience with LAPI-a new high-performance communication library for the IBM RS/6000 SP. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing.* 260–266. DOI: https://doi.org/10.1109/IPPS.1998.669923

[18] Christian Simmendinger, Mirko Rahn, and Daniel Gruenewald. 2015. *The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures.* Springer International Publishing, Cham, 17–32. DOI: https://doi.org/10.1007/978-3-319-10626-7_2

[19] Monika ten Bruggencate and Duncan Roweth. 2010. DMAPP: An API for One-Sided Programming Model on Baker Systems. *Cray Users Group (CUG), Tech. Rep.* (2010).

[20] UPC Consortium. 2005. *UPC Language Specifications, v1.2.* Tech Report LBNL-59208. Lawrence Berkeley National Lab. http://www.gwu.edu/~upc/publications/LBNL-59208.pdf

[21] Rob F. Van der Wijngaart, Abdullah Kayi, Jeff R. Hammond, Gabriele Jost, Tom St. John, Srinivas Sridharan, Timothy G. Mattson, John Abercrombie, and Jacob Nelson. 2016. Comparing Runtime Systems with Exascale Ambitions Using the Parallel Research Kernels. In *ISC High Performance.* 321–339. DOI: https://doi.org/10.1007/978-3-319-41321-1_17

[22] Rob F. Van der Wijngaart, Srinivas Sridharan, Abdullah Kayi, Gabriele Jost, Jeff R. Hammond, Timothy G. Mattson, and Jacob E. Nelson. 2015. Using the Parallel Research Kernels to study PGAS models. In *PGAS.* IEEE.

[23] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. 2003. The SGI Altix TM 3000 global shared-memory architecture. Technical Whitepaper, Silicon Graphics. (2003).