Persistent Coarrays: Integrating MPI Storage Windows in Coarray Fortran

Sergio Rivas-Gomez
sergiorg@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

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

Sai Narasimhamurthy
sai.narasimhamurthy@seagate.com
Seagate Systems UK
Havant, United Kingdom

Stefano Markidis
markidis@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

The inherent integration of novel hardware and software components on HPC is expected to considerably aggravate the Mean Time Between Failures (MTBF) on scientific applications, while simultaneously increase the programming complexity of these clusters. In this work, we present the initial steps towards the integration of transparent resilience support inside Coarray Fortran. In particular, we propose *persistent coarrays*, an extension of OpenCoarrays that integrates MPI storage windows to leverage its transport layer and seamlessly map coarrays to files on storage. Preliminary results indicate that our approach provides clear benefits on representative workloads, while incurring in minimal source code changes.

CCS CONCEPTS

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

KEYWORDS

Persistent Coarrays, Coarray Fortran, MPI Storage Windows

ACM Reference Format:
Sergio Rivas-Gomez, Alessandro Fanfarillo, Sai Narasimhamurthy, and Stefano Markidis. 2019. Persistent Coarrays: Integrating MPI Storage Windows in Coarray Fortran. In *26th European MPI Users' Group Meeting (EuroMPI 2019), September 11–13, 2019, Zürich, Switzerland*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3343211.3343214

1 INTRODUCTION

With the emergence of deep learning and data-centric workloads on HPC, drastic hardware and software changes are expected to be featured on upcoming major supercomputers [13, 25]. These changes mostly aim to address the recent demands of the scientific community. For instance, novel CPU architectures are likely to integrate data formats that use tensors with a shared exponent [6, 21], maximizing the dynamic range of the 16-bit floating point data format for training and inference on convolutional neural networks.

While these technological breakthroughs can provide multiple advantages in terms of performance and power consumption (e.g., 5–10× using just twice the power of current large-scale HPC clusters [33]), it has also been largely debated that the presence of billions of hardware components and several levels of software stack will also represent an increment in number of hardware and software failures [4, 12, 28]. In addition, the increase in concurrency, estimated between 100–1000× in comparison [8], will further aggravate this problem.

To overcome some of these limitations, recent supercomputers feature a variety of Non-Volatile RAM (NVRAM) memories with different performance characteristics, next to conventional hard disks and DRAM [24, 27] (Figure 1). Compared to traditional parallel file systems, the cost of accessing these memory tiers can provide tremendous opportunities for fault-tolerance [19]. Nonetheless, allocating and moving data in such systems often require the use of different programming interfaces to program separately memory and storage. Hence, the inherent heterogeneity poses additional constraints due to the programming complexity, making it difficult for scientific applications to take advantage of these developments.

In this work, we set the initial steps towards the integration of transparent resilience support inside Coarray Fortran (CAF). In particular, we present the concept of *persistent coarrays*, a mechanism that provides implicit storage support with barely minimal source code changes. By extending the transport layer implementation of OpenCoarrays [11], we take advantage of MPI storage windows [31] to map coarrays to files and enable direct access to a diverse range of memory and storage technologies. Furthermore, persistent coarrays represents a vital contribution to the recovery mechanism of `failed images` [12], opening the opportunity for seamless fault-tolerance in the near-term future. Initial performance results indicate that the penalty of persistent coarrays is negligible when performing remote memory operations compared to non-resilient implementations. Moreover, we also illustrate that local storage can improve approximately up to 2× the performance obtained with traditional parallel file systems.

The contributions of this work are the following:

- We present the concept of persistent coarrays to provide transparent resilience support in Coarray Fortran.
- We integrate the concept of MPI storage windows [31] inside OpenCoarrays to extend its MPI-based transport layer.

EuroMPI 2019, September 11–13, 2019, Zürich, Switzerland
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7175-9/19/09…$15.00
https://doi.org/10.1145/3343211.3343214

- We evaluate and compare the performance of our implementation under representative workloads, and compare the results with non-resilient implementations.
- We provide further insight into how this approach could be integrated into future revisions of the Coarray Fortran standard, including three potential API alternatives.

The paper is organized as follows. We provide an overview of Coarray Fortran and present the design and implementation details of persistent coarrays based on MPI storage windows in Section 2. The experimental setup and performance results of the EPCC CAF Microbenchmark suite [18] and Intel's Parallel Research Kernels (PRK) [35] are presented in Section 3. We extend the discussion of the results and provide further insights in Section 4. Related work is described in Section 5. Lastly, Section 6 summarizes our conclusions and outlines future work on this topic.

2 PERSISTENT COARRAYS

Coarray Fortran (CAF) originated as a syntactic extension of Fortran 95, which eventually became part of the Fortran 2008 standard in 2010 (ISO/IEC 1539-1:2010) [26, 30]. The main objective of CAF is to simplify the development of parallel applications without the burden of explicitly invoking communication primitives or directives, such as those available in the Message-Passing Interface (MPI) [23] or OpenMP [7].

The programming model of CAF is based on the Partitioned Global Address Space (PGAS) model, in which every process is able to access a portion of the memory address space available on other processes using shared-memory semantics. A program that uses CAF is treated as a replicated entity, commonly referred as `image`. This is similar to the concept of "rank" in MPI terminology. An `image` is assigned a unique index, that is represented by a number between 1 and the number of images (inclusive). In order to identify a specific `image` at runtime or the total number of images, Fortran provides the `this_image()` and `num_images()` functions.

Each `image` executes asynchronously until the programmer explicitly synchronizes it through one of the synchronization mechanisms available in the standard. For instance, the "`sync all`" statement serves as a barrier and guarantees the synchronization of the different `images`[1]. Any variable can be declared as *coarray* inside an `image`, which can be represented by a scalar or array, static or dynamic, and of intrinsic or derived type. Applications access a coarray object on a remote `image` using square brackets "`[index]`". An object with no square brackets is considered local to the process.

In this context, extending the concept of coarray to become persistent requires no major changes into the Fortran standard. In particular, while the specification mostly describes the semantics and respective functionality to interact between coarrays located on different `images`, it does not restrict the supporting technology where the specific coarray is pinned to (e.g., NVDIMM [19, 24]). Moreover, the different synchronization mechanisms, already available in the Fortran standard, provide an opportunity to guarantee data consistency with storage as necessary. As a consequence, the integration of resilient coarrays inside CAF is expected to become feasible in the near-term future.

[1]Source code contained between synchronization points represents a `segment`. Here, only `get` operations are guaranteed to conclude before the synchronization barrier.

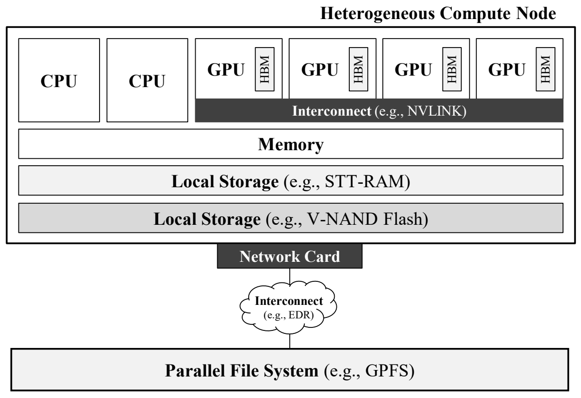

Heterogeneous Compute Node
CPU
CPU
GPU HBM
GPU HBM
GPU HBM
GPU HBM
Interconnect (e.g., NVLINK)
Memory
Local Storage (e.g., STT-RAM)
Local Storage (e.g., V-NAND Flash)
Network Card
Interconnect (e.g., EDR)
Parallel File System (e.g., GPFS)

Figure 1: Compute nodes of upcoming supercomputers feature local storage technologies. The diagram is inspired by the hardware configuration of Summit (ORNL) [33, 36].

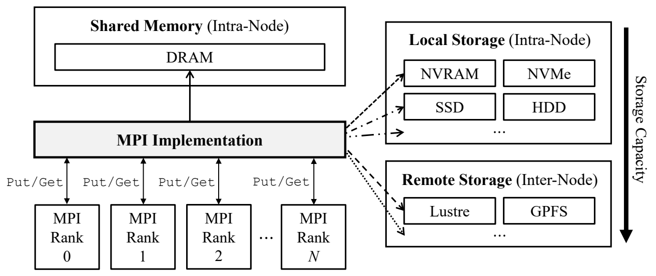


Figure 2: MPI storage windows provides seamless access to a diverse range of memory and storage technologies through the MPI one-sided communication model [31].

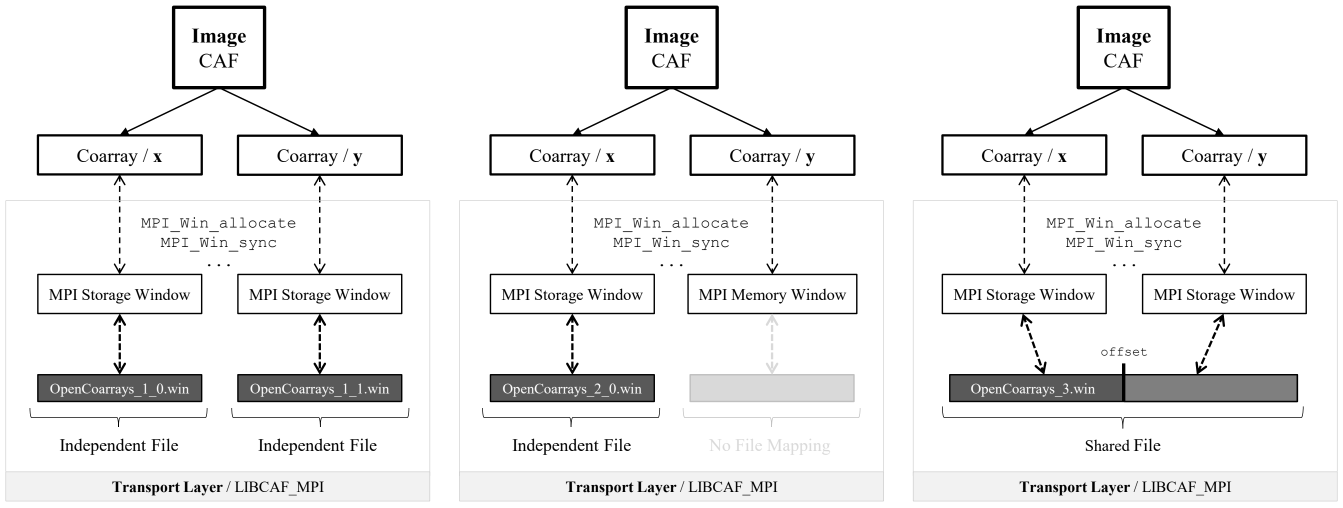
In this section, we present the design and implementation details of persistent coarrays, an extension that seamlessly allows the integration of fault-tolerance mechanisms on CAF-based applications. Thereafter, we provide further insight into how this approach could be adopted by the Fortran community and illustrate a source code example that demonstrates its usage.

2.1 Design and Implementation

We design and implement persistent coarrays as an extension inside the OpenCoarrays communication library for CAF compilers[2] [11]. The implementation extends the release version v2.6.3 and consists of approximately 100 lines of code changes in comparison. It also supports the same standardized CAF functionality of the original.

OpenCoarrays is a collection of open-source transport layers that assist CAF compilers by translating the communication and synchronization requests into specific primitives from other communication libraries (e.g., OpenSHMEM [5]). The GNU Fortran (GFortran) compiler has integrated OpenCoarrays since the GNU Compiler Collection (GCC) v5.1.0 release. The front end of GFortran is designed to remain agnostic about the actual transport layer employed in the communication. By default, GFortran uses the

[2]https://github.com/sourceryinstitute/OpenCoarrays/tree/MPISWin



(a) File-per-process Configuration **(b) Mixed Persistent + Non-Persistent** **(c) Shared-file Configuration**

Figure 3: By integrating the concept of MPI storage windows inside the transport layer of OpenCoarrays, we can enable seamless access to storage to create the concept of persistent coarrays in CAF. For instance, different processes might map their coarrays into individual or shared files, and might be able to combine persistent and non-persistent coarrays in the future.

`LIBCAF_MPI` transport layer of OpenCoarrays, which is based on the MPI one-sided communication model [14, 16]. Here, coarrays are represented as MPI memory windows and accessed using `put` / `get` internally through the passive target synchronization of MPI-3 [23].

We extend the MPI-based transport layer of OpenCoarrays to integrate the concept of MPI storage windows [31]. This on-going effort proposes to raise the level of programming abstraction by using MPI one-sided communication and MPI windows as a unified interface to access a diverse range of memory and storage technologies (Figure 2). MPI windows provide a familiar interface that can be used to program data movement among hybrid memory and storage subsystems, allowing to map MPI windows to files. Thus, coarrays can immediately benefit from this integration and transparently expose resilient variables inside `images`. The use of persistent coarrays could considerably reduce the programming complexity on HPC applications, while providing support for different fault-tolerance mechanisms, such as checkpoint-restart [4, 17].

Existing CAF applications compiled with GFortran and the extended OpenCoarrays implementation, will seamlessly convert their traditional coarrays (i.e., memory-based) into persistent coarrays (i.e., storage-based). In most cases, no specific source code changes are required. We also extend the synchronization statements defined in the CAF specification to automatically enforce consistency between main memory and storage. This is accomplished by internally calling `MPI_Win_sync` with the MPI storage window associated to each coarray. Further information is provided in Section 2.2.

For compatibility reasons with the Fortran standard, persistent coarrays are configured through several environment variables. If a specific version of OpenCoarrays does not support persistent coarrays, the environment variables are simply ignored and applications use traditional coarrays in memory instead.

These are the new variables introduced in our implementation:

- **`PCAF_ENABLED`**. If set to "`true`", it enables support for persistent coarrays in the application. Otherwise, the coarrays will be allocated in memory (default).
- **`PCAF_PATH`**. Defines the path where the files corresponding to each coarray are mapped. This allows to support different storage targets (e.g., local or remote storage). The name of each file is automatically generated using the `image` index, allowing to remap the content between executions.
- **`PCAF_PREFIX`**. Determines the prefix to be used for each coarray file. By default, "`OpenCoarrays_`" is assigned.
- **`PCAF_SHAREDFILE`**. If set to "`true`", it allows the use of a single shared file that will contain the coarrays of a given `image`. Otherwise, a file per coarray is created (default).
- **`PCAF_UNLINK`**. If set to "`true`", removes the associated files after execution (i.e., useful for out-of-core with coarrays).
- **`PCAF_SYNCFREQ`**. Specifies the synchronization frequency limit expected after each synchronization. The aim is to prevent frequent synchronizations with storage (see Section 2.2).
- **`PCAF_SEGSIZE`**. Defines the segment size utilized in the specific MPI storage window where the coarray is mapped. By default, the current page-size available at runtime is set.
- **`PCAF_STRIPESIZE`**. Sets the striping unit to be used for the mapped file (e.g., stripe size of Lustre). This hint has no effect on existing files and/or local storage.
- **`PCAF_STRIPECOUNT`**. Sets the number of I/O devices that the mapped file should be striped across (e.g., OSTs on Lustre). This hint has no effect on existing files and/or local storage.

In order to allocate a persistent coarray inside a certain CAF `image`, we create a dedicated MPI storage window and configure it through *performance hints* provided to the `MPI_Win_allocate` call.

```fortran
...
! Declare two coarrays and other variables
real, dimension(10), codimension[*] :: x, y
integer :: num_img, img

! Retrieve the index and the number of images
img = this_image()
num_img = num_images()

! Perform certain operations accross images
x(2) = x(3)[7] ! Get value from image 7
x(6)[4] = x(1) ! Put value on image 4
x(:)[2] = y(:) ! Put a complete array on image 2

! Enforce synchronization point, which guarantees
! a data synchronization with storage as well
sync all

! Remote array transfer, with ind. synchronization
if (img == 1) then
    y(:)[num_img] = x(:)
    sync images(num_img)
elseif (img == num_img) then
    sync images([1])
endif
...
```

Listing 1: Source code example that illustrates how to use coarrays in Fortran. The source code will use persistent coarrays if compiled with our version of OpenCoarrays.

The MPI hints determine the location of the mapping to storage and define other hardware-specific settings. We use the environment variables described to configure some of these settings. For instance, applications that use CAF can continue to allocate coarrays in memory by avoiding to set the environment variable PCAF_ENABLED, or by setting it to "false". To enable persistent coarrays, it is enough to define the aforementioned variable with a "true" value instead. The rest of the described environment variables are optional and will strictly depend on the particular use-case where persistent coarrays is integrated. By default, the active path is used to store the different files that support the coarrays.

The current implementation of MPI storage windows is based on the use of the memory-mapped file I/O mechanism of the OS [3]. Target files from an MPI storage window are first opened, mapped into a virtual address space, and then associated with the MPI window. Even though a better implementation of the mapping mechanism of MPI storage windows is expected to be available soon, we found unexpected issues in our preliminary evaluations and decided to focus on the stable release[3].

Figure 3 summarizes the integration of several persistent coarrays with their correspondent MPI storage windows. The first image contains two persistent coarrays that are mapped into the virtual memory space and assigned into two separate files on storage. After the mapping is established, the MPI storage windows implementation is responsible for migrating data from memory to storage, and vice-versa. The second image is similar to the previous one, but allocating the coarray y in memory instead (see Section 2.4). In addition, the figure illustrates one of the CAF images sharing a file through the PCAF_SHAREDFILE environment variable setting. The offset of each persistent coarray follows the program order and is calculated internally (i.e., it cannot be currently configured).

[3]https://github.com/sergiorg-kth/mpi-storagewin

2.2 Considerations for Data Consistency

The flexibility of MPI storage windows can introduce several challenges in regards to data consistency. First and foremost, local or remote operations are *only* guaranteed to affect the memory copy of the window inside the target image. This fact implies that the semantics of the MPI one-sided operations only ensure completion of the local or remote operations inside main memory[4] (i.e., the storage status might be undefined). The semantics for the operations are similar to the "public + private" copies of the window in MPI-2 [16]. Thus, we use MPI_Win_sync to enforce data consistency on each mapped image, as necessary. Read operations, on the other hand, will still trigger data accesses to storage automatically.

Using persistent coarrays, applications are required to perform an explicit synchronization of the images when consistency of the data within the storage layer has to be preserved. For this purpose, we extend part of the available synchronization primitives in CAF:

- **Collective synchronization**. Enforces a barrier-like synchronization, in which all the processes ensure RDMA completion (as in CAF) plus a synchronization with storage. This is accomplished with "sync all".
- **Individual synchronization**. Enforces individual synchronization between groups of images, without involving the rest. This is accomplished with "sync images([indices])".

Applications can choose whether to always synchronize with storage during each synchronization point (default), or to limit the amount of data synchronizations performed to improve the overall performance. The environment variable PCAF_SYNCFREQ determines the interval of time in milliseconds in which a persistent coarray is considered outdated. For instance, setting a value of 500 implies that the synchronization of the persistent coarrays will only take effect at least after every 500ms. This setting is inspired by the vm.dirty_expire_centisecs provided by the OS[5].

Lastly, it is important to take into account that Fortran provides locks, critical sections, atomic intrinsics, and events. Using these operations, applications can guarantee data consistency even when multiple images are accessing the same target coarray.

2.3 Using Persistent Coarrays

Listing 1 illustrates a source code example using persistent coarrays. Here, the application begins by declaring two coarrays x and y, with 10 floating-point values each. The use of codimension[*] requests the compiler to define these as CAF coarrays, instead of traditional Fortran arrays. The brackets determine the layout and distribution of the images, which is set by default in this case.

Thereafter, the example retrieves the index and also the number of images, similar to retrieving the rank and the size of the MPI_COMM_WORLD communicator in MPI [16]. Multiple operations are then performed across the images, using the square brackets

[4]This limitation does not strictly apply when using NVDIMM memories [24], that are expected to be accessible through RDMA when widely available.
[5]https://www.kernel.org/doc/Documentation/sysctl/vm.txt

"`[index]`" syntax to specify the target image for the `put` / `get` operation being conducted. A global synchronization is also requested, which will guarantee not only that the data has been transferred across the different `images`, but that the persistent coarray is synchronized with storage. Finally, the source code performs a remote array transfer and shows how to enforce an individual synchronization between two pair of `images`.

Note that the example is also valid for traditional coarrays (i.e., the support for persistent coarrays is provided at compiler level).

2.4 Expected API for Persistent Coarrays

The presented implementation in OpenCoarrays seamlessly allows to convert existing coarrays into persistent coarrays. Despite the simplicity of our approach, we also consider that it imposes several constraints. For instance, it is currently not possible to differentiate individually between persistent and non-persistent coarrays.

Even though the design of a specific API for persistent coarrays has been considered out of the scope of this work, we provide below further insight about how the concept could pave its way into a generalized API that might even influence the Fortran standard in the future. We consider three different alternatives:

Keyword in Fortran. Extending the declaration of a coarray to become persistent could be possible by the integration into the standard of a "`persistent`" keyword, provided during the declaration of the coarray. The main limitation of this approach is that it would require changes into the CAF specification in the Fortran standard. Furthermore, the path must still be configured by other means (e.g., through environment variables or a data-placement runtime).

```
real, persistent :: a(n)[*] ! Simplified declaration
real, dimension(n), persistent, codimension[*] :: a
```

Compiler Directives. Instead of forcing changes into the Fortran standard, an alternative could be to define a mechanism using directives. The support for this approach is at compiler level, and provides the opportunity to configure the persistent coarray (e.g., path). The declaration of the coarray would be almost identical.

```
!dir$ persistent, path, ...
real, dimension(n), codimension[*] :: a ! No changes
```

Library API. Alternatively, OpenCoarrays could expose specific functionality to extend the definition of a coarray to become persistent (e.g., similar to the mechanism used by DMAPP [34]). The main advantage of this option is that it does not depend on the Fortran standard or the compiler, as long as OpenCoarrays is the transport layer for CAF. In addition, the path is individually configured for each coarray file (i.e., instead of a base path for all the files).

```
real, dimension(n), codimension[*] :: a ! No changes
...
call opencoarrays_config(a, "persistent", "path", ...)
```

We plan to examine the possibilities of these and other approaches in future work on this topic.

3 EXPERIMENTAL RESULTS

In this section, we illustrate the performance of persistent coarrays using a reference testbed of the National Center for Atmospheric Research (NCAR). This testbed allows us to demonstrate the implications of our approach on upcoming clusters with local persistency support, that also combine traditional parallel file systems. The specifications of the testbed are described below:

- **Casper** is a cluster with 28 heterogeneous compute nodes equipped with Intel Xeon Gold 6140 (Skylake) processors running at 2.3GHz. The group used in our tests contains two sockets with 18 cores and a total of 384GB DRAM per node. Storage is provided through 2TB of local NVMe SSD drives per node, alongside a GPFS parallel file system with approximately 15PB. The network uses Mellanox VPI EDR InfiniBand dual-port interconnect. The OS is CentOS 7 with Kernel v3.10.0-693.21.1.el7.x86_64 . The applications are compiled with `GCC` v7.3.0 and OpenMPI 3.1.4. The transport layer is provided by OpenCoarrays v2.6.3 with our extension.

The figures reflect the standard deviation of the samples as error bars. We use the PMPI-based [20] implementation of MPI storage windows inside OpenCoarrays for deployment reasons on Casper. We set the default values for most of the I/O settings of the library. In addition, we also use the default GPFS settings, assigning a block size of 8MB and a sub-block of 16KB. The evaluations are conducted on different days and timeframes, to account for the interferences produced by other users on the cluster. Note that, after this section, we continue and extend the discussion on the obtained results.

3.1 EPCC CAF Microbenchmark Suite

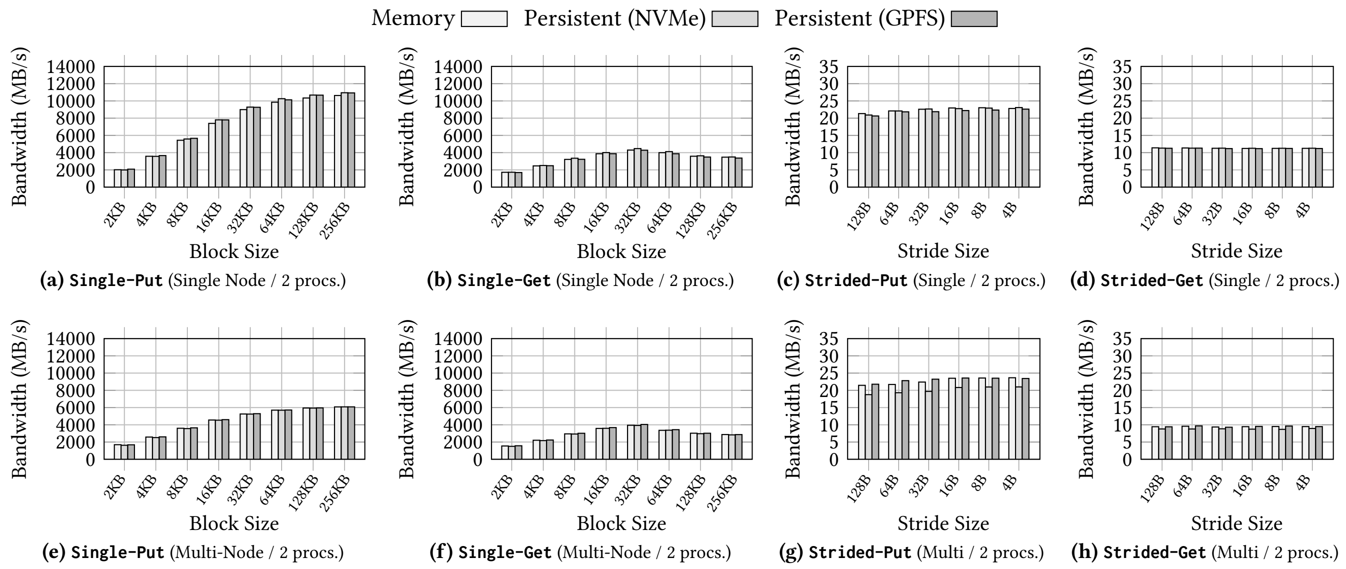
We first verify that the integration of MPI storage windows in the transport layer of OpenCoarrays does not incur in additional overheads when performing remote memory operations. This can be useful to existing CAF-based applications that aim to integrate resiliency support, but do not necessarily enable it by default.

For this purpose, we use the Fortran Coarray Microbenchmark suite from the University of Edinburgh (EPCC) [18]. This open-source project[6] focuses on a small set of low-level operations, such as `put`, `get`, `strided put`, `strided get`, and typical communication patterns, like halo exchange. Their performance is then measured in terms of bandwidth and latency in isolation, using double-precision floating-point values.

Every basic operation is analyzed in two different scenarios: single point-to-point and multiple point-to-point. We focus on the former, in which a given CAF `image` interacts with another one located inside the same node (i.e., intra-node communication), and also outside the node (i.e., inter-node communication). In this scenario, there is no network contention to expect. To prevent interferences on the results, we avoid to enforce a storage synchronization.

Figure 4 illustrates the measured bandwidth of traditional coarrays, as well as persistent coarrays that use local storage (NVMe) and the parallel file system (GPFS) available on Casper. The evaluations conduct single and multi-node evaluations, using two different

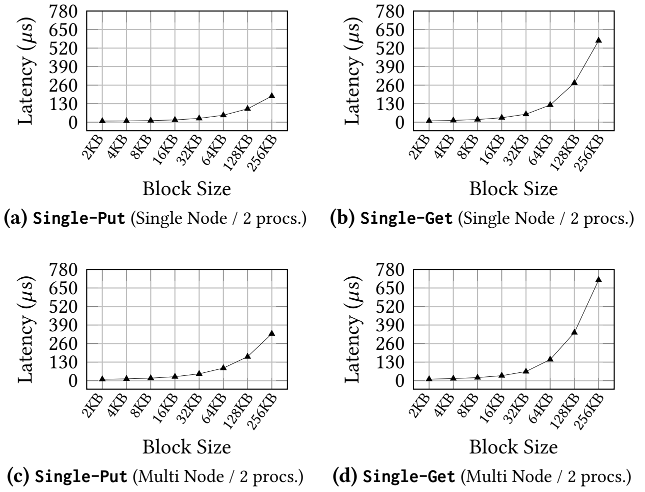
[6]https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-co-array-fortran-micro



(a) `Single-Put` (Single Node / 2 procs.) (b) `Single-Get` (Single Node / 2 procs.) (c) `Strided-Put` (Single / 2 procs.) (d) `Strided-Get` (Single / 2 procs.)

(e) `Single-Put` (Multi-Node / 2 procs.) (f) `Single-Get` (Multi-Node / 2 procs.) (g) `Strided-Put` (Multi / 2 procs.) (h) `Strided-Get` (Multi / 2 procs.)

Figure 4: Measured bandwidth of traditional coarrays in memory, as well as persistent coarrays on local storage (NVMe) and a parallel file system (GPFS) using the EPCC CAF Microbenchmarks. The tests run on 2 different processes located on a single node (a-d) and on two nodes (e-h) of Casper, and evaluate both single and strided `put` / `get` operations.



(a) `Single-Put` (Single Node / 2 procs.) (b) `Single-Get` (Single Node / 2 procs.)

(c) `Single-Put` (Multi Node / 2 procs.) (d) `Single-Get` (Multi Node / 2 procs.)

Figure 5: Measured latency of persistent coarrays on local storage (NVMe) using the EPCC CAF Microbenchmarks. The tests run on 2 processes located on a single node (a-b) and on two nodes (c-d) of Casper.

processes / `images`. The results indicate that no relevant performance differences are observed when conducting `put` / `get` operations, including when using strided accesses. The use of persistent coarrays even reflects slightly better results compared to traditional coarrays, featuring approximately a 4% improvement on average. We assume that this side-effect is due to how the implementation of MPI storage windows allocates memory in comparison with the default memory allocator provided by the OpenMPI implementation used in our experiments.

On the other hand, Figure 5 shows the latency of the previous evaluation, illustrating in this case only the results for persistent coarrays that use local storage (NVMe). From the results, we observe that `put` operations generally feature lower latency and higher-bandwidth, even for large data transfers. However, `get` operations incur in additional overheads, degrading the access latency exponentially as the block size increases. For instance, for a block size of 256KB using two different nodes of Casper, the latency of a `get` operation is 2.14× slower in comparison. Nonetheless, we must note that the results for traditional coarrays and persistent coarrays in a parallel file system do not differ[7].

3.2 Parallel Research Kernels

We now evaluate persistent coarrays using the Parallel Research Kernels (PRK) from Intel Corporation [35]. This collection of kernels covers some of the most common patterns of communication, computation, and synchronization encountered in parallel HPC applications. Thus, we use PRK to understand the implications of persistent coarrays under representative workloads.

The CAF implementation of PRK[8] provides three computational kernels. We pick the most relevant two. The first kernel is `Stencil`, that applies a data-parallel stencil operation to a two-dimensional array. The second kernel is `Transpose`, that stresses communication and memory bandwidth with regular, unit strided `reads`, and non-unit strided `writes` (and vice versa). Compared to the previous evaluation, in this case we enforce storage synchronizations during the execution of the aforementioned kernels. Thus, we try to understand the possible performance considerations of persistent coarrays with full storage support.

[7]For illustration purposes, we decided to provide only the results for persistent coarrays on local NVMe storage.

[8]https://github.com/ParRes/Kernels/tree/master/FORTRAN

Figure 6 illustrates the performance of traditional coarrays in memory, as well as persistent coarrays that use local storage (NVMe) and the parallel file system (GPFS) available on Casper. The evaluations utilize a grid dimension of 8192×8192, and perform 10 iterations. The results illustrate that the use of local storage for persistent coarrays can be beneficial. For 128 processes (4 nodes), the performance improves 2.1× in comparison when observing the `Transpose` kernel. Moreover, due to the problem size, the limited amount of I/O operations required for persistent coarrays makes it produce identical results compared to coarrays in memory.

4 DISCUSSION

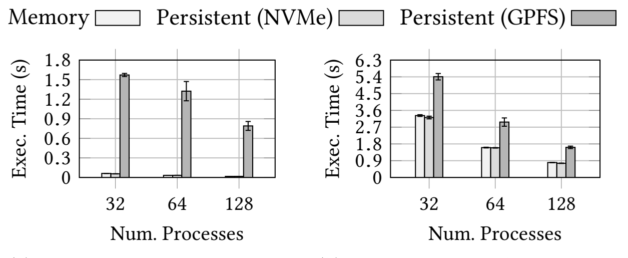
We further extend the discussion concerning the results given in the previous section and additional observations.

Local Storage Performance. The trend for large-scale supercomputer design is diverging from the traditional compute-only approach, to hybrid solutions that aim to reduce data movement inside the cluster [25]. Local storage, such as the promising NVDIMMs [15, 19], is integrated as a scratch tier that is part of the storage hierarchy of the cluster. Despite using NVMe-based drives, our results on Casper illustrate that the use of persistent coarrays on local storage provides enhanced performance compared to parallel file systems, like GPFS. Consequently, this fact opens opportunities for neighbour checkpoints (i.e., `images` replicate the state between nodes using remote operations over persistent coarrays) or seamless out-of-core execution (i.e., `images` exceed main memory transparently).

Importance of the MPI implementation. The default implementation of MPI storage windows is based on the memory-mapped I/O mechanism of the OS, which imposes major constraints on certain HPC applications [31]. In the context of the Sage2 EU-H2020 project [25], we are developing a User-level Memory-mapped I/O (uMMAP-IO) library that provides enhanced control of the data. Figure 7 illustrates preliminary results running on Cori from NERSC [2]. The tests use a STREAM-inspired microbenchmark[9] [22] designed to stress the I/O subsystem. The results show promising performance improvements with MPI storage windows based on uMMAP-IO in comparison to MMAP-IO on different workloads, with up to 5× increased throughput on average when using the Burst Buffer (not supported by MMAP-IO) and up to 2.5× using Lustre. Thus, we expect that future implementations of MPI storage windows would dramatically improve the performance of persistent coarrays.

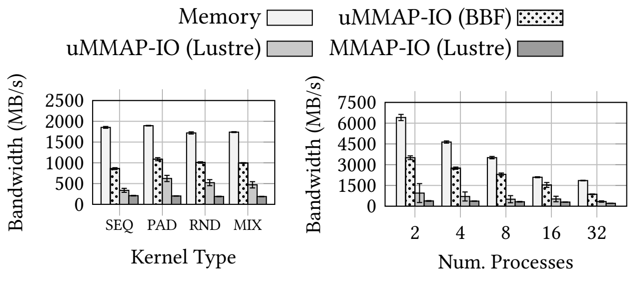
Opportunities for fault-tolerance. Despite not discussing specific mechanisms for fault-tolerance in this work, we observe that persistent coarrays can provide a tremendous opportunity to enable transparent resilience support on HPC applications. Generalizing the API of persistent coarrays, as previously described, would allow applications to alter the mapping of their respective coarrays and recover after a failure. Moreover, the concept of "`failed images`" [12, 29], recently introduced to the Fortran 2018 standard, could also provide the necessary tools to recover `images` that utilize persistent coarrays. This would only require subtle source code modifications on already existing applications that use CAF to parallelize their code. We plan to explore this opportunity in future work.

[9] https://github.com/sergiorg-kth/mpi-storagewin/tree/master/benchmark



(a) `Stencil` Kernel Evaluation **(b) `Transpose` Kernel Evaluation**

Figure 6: Performance of traditional coarrays in memory, as well as persistent coarrays on local storage (NVMe) and a parallel file system (GPFS) using reference PRK kernels. The tests run on 1, 2 and 4 nodes of Casper with a grid of 8192.



(a) Kernel Eval. (32 procs.) **(b) Weak Scaling** (SEQ / Ind. BW)

Figure 7: Measured bandwidth of MPI memory windows and MPI storage windows using mSTREAM on a single-node of Cori [2], and varying the implementation type.

5 RELATED WORK

Designing a fault resilient system can be done at different levels of the software stack. The general-purpose technique most widely used in HPC relies on checkpointing and rollback recovery: parts of the execution are lost when processes are subject to failures, and the fault-tolerant protocol, when catching such errors, uses past checkpoints to restore the application in a consistent state, and recomputes the missing parts of the execution.

For the purpose of this work, a runtime library is in charge of performing the checkpointing by using the new concept of persistent coarrays. The checkpointing is implemented almost at compiler level. In [9], the authors propose a similar approach to the one proposed in our work; however, their work focuses strictly on MPI one-sided whereas our work focuses more on the integration of the MPI Storage Windows [31] with the coarray semantics. The concept of compiler-based transparent fault-tolerance treated in this work is not new, and presented in [32] by Rodriguez et al.

Because the checkpointing mechanism is hidden by the compiler and runtime library, the way the data is stored can be changed transparently by the user (by using an environment variable). In [1, 10], the authors show that a combination of regular DRAM and SSD can be used to avoid writing all the checkpoints on disk, without impacting the possibility of recovering from failures. We plan to explore the implementation of such hybrid solution in future work.

6 CONCLUSION AND FUTURE WORK

The inherent increase in number of hardware and software components on HPC, will likely aggravate the amount of hardware and software failures faced by scientific applications [4, 28]. In this paper, we have presented the concept of persistent coarrays, an extension to the coarray specification of Coarray Fortran that provides seamless resilience support. Thus, the approach enables the integration of fault-tolerance mechanisms for CAF in the future.

Preliminary results have demonstrated that no performance penalties are introduced by the integration of MPI storage windows with OpenCoarrays when performing remote memory operations. In addition, the performance of persistent coarrays using local storage provides clear advantages compared to the use of traditional parallel file systems, like GPFS.

As future work, we plan to explore the integration of persistent coarrays as part of the recovery mechanism of `failed images` [12] on HPC applications. In addition, we also expect to evaluate the integration of the user-level memory-mapped I/O mechanism that is currently being developed for MPI storage windows [31].

ACKNOWLEDGMENTS

The experimental results were performed on resources provided by the National Center for Atmospheric Research (NCAR) in the United States, as well as resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231 (Project ID: OpenCoarrays_FT).

Part of this work was also funded by the European Commission through the Sage2 project (Grant agreement no. 800999). More information at http://www.sagestorage.eu.

REFERENCES

[1] Nilmini Abeyratne, Hsing-Min Chen, Byoungchan Oh, Ronald Dreslinski, Chaitali Chakrabarti, and Trevor Mudge. 2016. Checkpointing Exascale Memory Systems with Existing Memory Technologies. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. ACM, New York, NY, USA, 18–29. https://doi.org/10.1145/2989081.2989121

[2] Katie Antypas, Nicholas Wright, Nicholas P Cardo, Allison Andrews, and Matthew Cordery. 2014. Cori: A Cray XC Pre-exascale System for NERSC. *Cray User Group Proceedings. Cray* (2014).

[3] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management.* O'Reilly.

[4] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. 2009. Toward Exascale Resilience. *The International Journal of High Performance Computing Applications* 23, 4 (2009), 374–388.

[5] 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.* ACM, 2.

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering* 1 (1998).

[8] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. 2011. The International Exascale Software Project Roadmap. *The International Journal of High-Performance Computing Applications* 25, 1 (2011).

[9] Piotr Dorożyński, Paweł Czarnul, Artur Malinowski, Krzysztof Czuryło, Łukasz Dorau, Maciej Maciejewski, and Paweł Skowron. 2016. Checkpointing of parallel MPI applications using MPI one-sided API with support for byte-addressable non-volatile RAM. *Procedia Computer Science* 80 (2016), 30–40.

[10] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. 2017. Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 318–329. https://doi.org/10.1109/PACT.2017.58

[11] Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. 2014. OpenCoarrays: Open-source Transport Layers supporting Coarray Fortran compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models.* ACM, 4.

[12] Alessandro Fanfarillo, Sudip Kumar Garain, Dinshaw Balsara, and Daniel Nagle. 2019. Resilient Computational Applications using Coarray Fortran. *Parallel Comput.* 81 (2019), 58–67.

[13] Michael Feldman. 2017. Oak Ridge readies Summit supercomputer for 2018 debut. in: Top500.org, http://bit.ly/2ERRFr9. [On-Line].

[14] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2014. Enabling highly-scalable remote memory access programming with MPI-3 one sided. *Scientific Programming* 22, 2 (2014), 75–91.

[15] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2019. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *arXiv preprint arXiv:1904.07162* (2019).

[16] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using advanced MPI: Modern features of the message-passing interface.* MIT Press.

[17] William Gropp and Ewing Lusk. 2004. Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications* 18, 3 (2004), 363–372.

[18] David Henty. 2011. A Parallel Benchmark Suite for Fortran Coarrays. In *Parallel Computing*. Elsevier, 281–288.

[19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[20] Edward Karrels and Ewing Lusk. 1994. Performance analysis of MPI programs. In *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*. 195–200.

[21] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. 1740–1750.

[22] John D McCalpin. 1995. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter* 19 (1995), 25.

[23] MPI Forum. 2015. *MPI: A Message-Passing Interface Standard.* Vol. 3.1. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Accessed: 2019-04-21.

[24] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2015. Non-volatile storage. *Commun. ACM* 59, 1 (2015), 56–63.

[25] Sai Narasimhamurthy, Nikita Danilov, Sining Wu, Ganesan Umanesan, Stefano Markidis, Sergio Rivas-Gomez, Ivy Bo Peng, Erwin Laure, Dirk Pleiter, and Shaun De Witt. 2018. SAGE: Percipient Storage for Exascale Data-centric Computing. *Parallel Computing* (2018).

[26] Robert W Numrich and John Reid. 1998. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, Vol. 17. ACM, 1–31.

[27] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. 2017. Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International.* IEEE, 683–692.

[28] Daniel A Reed and Jack Dongarra. 2015. Exascale computing and big data. *Commun. ACM* 58, 7 (2015), 56–68.

[29] John Reid. 2018. The new features of Fortran 2018. In *ACM SIGPLAN Fortran Forum*, Vol. 37. ACM, 5–43.

[30] John Reid and Robert W Numrich. 2007. Co-arrays in the next Fortran Standard. *Scientific Programming* 15, 1 (2007), 9–26.

[31] Sergio Rivas-Gomez, Roberto Gioiosa, Ivy Bo Peng, Gokcen Kestor, Sai Narasimhamurthy, Erwin Laure, and Stefano Markidis. 2018. MPI Windows on Storage for HPC Applications. *Parallel Computing* 77 (2018), 38–56.

[32] Gabriel Rodríguez, María J. Martín, Patricia González, Juan Touriño, and Ramón Doallo. 2010. CPPC: A Compiler-assisted Tool for Portable Checkpointing of Message-passing Applications. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 749–766. https://doi.org/10.1002/cpe.v22:6

[33] David Schneider. 2018. US supercomputing strikes back. *IEEE Spectrum* 55, 1 (2018), 52–53.

[34] Monika ten Bruggencate and Duncan Roweth. 2010. DMAPP - An API for One-sided Program Models on Baker Systems. In *Cray User Group Conference.*

[35] Rob F Van der Wijngaart and Timothy G Mattson. 2014. The Parallel Research Kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC).* IEEE, 1–6.

[36] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. 2018. The design, deployment, and evaluation of the CORAL pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis.* IEEE Press, 52.