A Portable Multidimensional Coarray for C++

Felix Mößbauer*, Roger Kowalewski†, Tobias Fuchs†, Karl Fuerlinger†
Computer Science Department, MNM Team
Ludwig-Maximilians-Universität (LMU) München
Email: *felix.moessbauer@campus.lmu.de, †first.last@nm.ifi.lmu.de,

***Abstract*—Fortran Coarrays are a well known data structure in High Performance Computing (HPC) applications. There have been various attempts to port the concept to other programming languages that have a wider user base outside of scientific computing. While a popular implementation of the partitioned global address space (PGAS) model is Unified Parallel C (UPC), there is currently no portable implementation of Coarrays for C++. In this paper a portable version is presented, which is closely based on the Coarray C++ implementation of the Cray Compiling Environment. In this work we focus on a common subset of all proposed features by Cray. Our implementation utilizes the distributed data structures provided by the DASH library, demonstrating their universal applicability. Finally, a performance evaluation shows that our proposed Coarray abstraction adds negligible overhead and even outperforms native Coarray Fortran.**

I. Introduction

The main objective of Coarrays is to allow Fortran programmers to operate on globally shared data without the burden to explicitly invoke communication primitives such as Message Passing Interface (MPI) routines. Fortran Coarrays are based on the Partitioned Global Address Space (PGAS) model which utilizes a one-sided *put / get* interface to emulate shared memory semantics on distributed memory systems. Like traditional message passing, PGAS follows a single program multiple data (SPMD) approach. Each process has a local memory address space and contributes some portion of it to the globally accessible memory. Since communication and synchronization are semantically decoupled, processes can access globally shared data independently from each other. In Coarray Fortran (CAF), processes are called images. While each image owns its own data objects, the array syntax is extended with square brackets denoting the image index. Using this mechanism, data on remote images can be accessed with conventional array semantics.

In this paper, we describe the design and implementation of a distributed data structure called `dash::Coarray` which can be accessed using a CAF-like syntax. The approach is based on a proposal by Cray to implement Coarray semantics in the C++ language [1]. In our implementation, the main functionality of the interface is implemented using existing DASH containers and closely follows the DASH global memory space concepts [2] [3]. In contrast to Fortran, C-like languages utilize square brackets for array accesses. Hence we introduce round brackets for accessing remote images.

A brief overview of the interface is provided in Listing 1. For specifying the extents of the Coarray, we follow closely the C++ array syntax. Coarrays can be scalar (one element per image) or n-dimensional, where all dimensions except one have to be specified at compile time.

```
dash::Coarray<int> i;           // scalar Coarray
dash::Coarray<int[10][20]> x;   // 2D-Coarray
dash::Coarray<int[][20]> y(n); // one open dim,
                                // set at runtime in ctor

// access syntax
i(unit) = value; // global access
i       = value; // local access

x(unit)[idx1][idx2] = value; // global access
x[idx1][idx2]       = value; // local access
```

Listing 1. Interface of the Coarray for scalar and array types showing local and global accesses.

The remainder of this paper is organized as follows. Section II summarizes essential concepts about DASH and Coarray Fortran to set the stage for this paper. Section III and IV elaborate the C++ Coarray in more detail and explain how we provide fundamental Coarray semantics in our PGAS abstraction. Section V conducts an experimental evaluation and reveals that the approach proposed in this work can even outperform native Coarray Fortran. In section VI we compare our implementation with similar research done in this field. Finally, section VII concludes.

II. Background

Before describing the C++ Coarray we briefly discuss our C++ library DASH, which serves as the underlying PGAS abstraction to provide basic mechanisms to communicate distributed data.

A. DASH

DASH aims at providing distributed data structures and parallel algorithms operating on them. The DASH library follows the design principles of the C++ STL (standard template library) to offer users a familiar interface and to ensure compatibility with existing algorithms, thus helping in the parallelization of existing C++ codes.

At the heart of the DASH template library is a set of distributed data structures, including one- and multidimensional arrays as well as lists and unordered maps. The distribution of data among the participating processes (called *units* in DASH) can be controlled by user-defined data distribution patterns [3].

Values in the distributed data structures are accessed through iterators and overloaded C++ operators, hiding the underlying put and get operations in traditional value assignment semantics. To facilitate efficient local data access and thus to avoid overhead, all DASH containers provide access to a unit's local elements through local iterators or raw pointers. DASH generalizes the C++ concepts of pointer, reference, and iterator to support a virtual global address space that spans the memory of multiple compute nodes in the form of a `GlobPtr<>`, `GlobRef<>`, and `GlobIter<>`, respectively.

All communication operations in DASH are executed through the DART runtime library, which aims at providing a thin abstraction of the underlying communication backend. While we have focused on using MPI-3 remote memory access (RMA) [4], implementations based on other PGAS communication substrates are also possible. An overview of the interaction between DASH, DART, and MPI is depicted in Figure 1.

B. CAF

Coarray Fortran (CAF) [5] can be seen as a continuation of the High Performance Fortran (HPF) effort of the 1990s, in which the goal was to make distributed memory systems more easily usable by employing a more productive programming model than explicit message passing. Where MPI requires explicit data partitioning and careful orchestration of send and receive operations, HPF features a single thread of control and data parallelism that is expressed in the form of compiler directives. HPF was ultimately not widely accepted, in part because of immature compiler technology and missing features[6]. In contrast, CAF is an explicitly parallel SPMD programming model, where data distribution and parallelism is exposed to the programmer and remote data access is made explicit using the Coarray notation using round brackets. CAF features have found their way into the Fortran standard since Fortran 2008. Additional features have been proposed for CAF such as teams and events [7]. Support for these and other new features in the next version of the Fortran standard is currently discussed.

III. The DASH Coarray

The `dash::Coarray` is internally implemented on top of the `dash::NArray` class template with a suitable data distribution pattern to store the data. `dash::NArray` provides a distributed N-dimensional fixed-size array, where the assignment of data elements to nodes (partitioning) is specified using a pattern. This approach is beneficial, as it enables to use the memory coherence model of the DASH Runtime (DART) [4]. Hence, we can address some of the CAF 2008 shortcomings, as pointed out by a group at Rice University without major effort [7]. This includes the following aspects:

- Added support for image subsets, called teams.
- Introduction of global pointers.
- Provision for synchronous and asynchronous access.
- Support for a flush operation, to wait until completion of asynchronous operations across a team.

We provide support for arbitrary element types as long as they are trivially copyable. This limitation is necessary as remote elements internally have to be copied into local memory before each access.

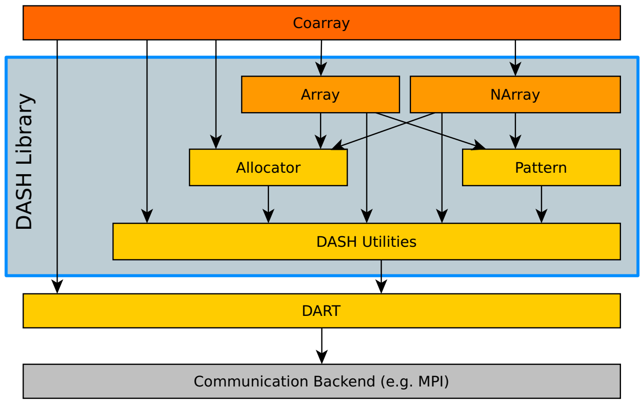


Fig. 1. Layered architecture of the DASH Coarray.

In the following sections we define an N-dimensional Coarray as a global array with N local array dimensions plus one codimension. A scalar Coarray is hence 0-dimensional and can be used in expressions like a variable as shown in Listing 1.

A. Allocation

In contrast to native C arrays, the distributed aspect adds more degrees of freedom to the allocation: In its simplest form, a Coarray is declared after DASH has been initialized. Then, the allocation follows the Resource Acquisition Is Initialisation (RAII) principle, and the distributed memory is allocated in the constructor. To ensure that all units have finished allocation before the first access happens, the exit of the constructor is synchronized using a barrier. The extents of the Coarray are specified similar to native C arrays, as shown in Listing 1.

As DASH follows the concept of teams for building groups of units, a distributed container is always constructed with a team. Accesses to the container are only allowed for the members of this team. Synchronization and collective operations are restricted in the same way. If no team is explicitly passed to the constructor, `dash::Team::All()` is used which contains all available units.

With the support for teams, delayed allocation is necessary as the teams are constructed at runtime. This means that the construction (RAII) and the allocation can be split. This also enables the user to declare the Coarray at a time where DASH has not been initialized. An example of this is using a Coarray as a member in a class, as shown in listing 2.

If the Coarray is declared before DASH has been initialized, a call to `x.allocate()` is required to actually allocate the data.

```
class Foo {
  dash::Coarray<int[10][20]> bar;
  Foo(const dash::Team & my_team = dash::Team::All()){
    // only images in my_team allocate coarray
    bar.allocate(my_team);
  }
};
dash::init(); // init DASH runtime
Foo f;
```

Listing 2. Example demonstrating how to use delayed allocation when placing the Coarray as a member in a struct.

Destruction of a Coarray is also a collective operation. This is necessary to ensure that there will be no remote accesses to an already destroyed object. Otherwise one instance of the object might already have left the scope while another unit accesses its data. This approach adds proper RAII semantics for distributed containers. For convenience it is also possible to explicitly deallocate a Coarray using `deallocate()`.

B. Data Access

The interface implements the container concept of DASH and is similar to the interface of `dash::NArray`. The public interface is similar to the interface proposed in Cray Coarray C++ [1]. The main idea is to separate the co-index from the array index by using different bracket syntax:

- `(unit)` for selecting the unit
- `[index]` for selecting an index within a units range.

If no unit is selected using the round bracket operator, the accesses to the Coarray are performed on the local part of the array.

As the requested element might be non-local, accesses are performed using global references (`GlobRef<T>`). These act as proxy references, providing an assignment operator and a conversion operator for the element type. Hence, elements at arbitrary locations can be accessed transparently. To use `dash::Coarray` elements in expressions, we provide `GlobRef<T>` specializations for native types. Thus, use cases as shown in Listing 3 are possible.

```
dash::Coarray<int> x;
x += 1;        // (1) Coarray::operator+=(int)
x = x(1) + 1; // (2) assign, Coarray::operator+(int)
x = 1 + x(1); // (3) assign, operator+(int, Coarray)
```

Listing 3. Coarray elements and scalar Coarrays can be used in expressions.

To support use cases as show in Listing 3 item (3), we provide specializations of the global arithmetic operators. This is required as the associativity of arithmetic operators in C++ is left to right.[1] While get operations are blocking, all put operations are issued asynchronously. To wait for all outstanding operations, `x.flush()` can be used.

1) Linear Access: The data stored in the Coarray can be accessed using iterators on both local and global ranges. While for global ranges the `begin()` and `end()` methods return a `GlobIter<T...>`, for local ranges a native pointer is returned. Fully transparent access is possible on both local and global ranges using global iterators / global references. Nevertheless a differentiation between local and global addresses is useful to avoid the overhead in the PGAS stack when accessing local data only. While both types of iterators can be passed to STL algorithms, global iterators can also be passed to the corresponding collective DASH algorithm variants. Both local and global iterators meet the `RandomAccessIterator` requirements.

[1]For details on the C++ operator precedence, see
http://en.cppreference.com/w/cpp/language/operator_precedence

```
dash::Coarray<int[10][20]> x;

// global iterators to full range
GlobIter<int> gbegin = x.begin();
GlobIter<int> gend   = x.end();

// global iterator to range on single unit
// (might not be local)
GlobIter<int> gbegin = x(0).begin();
GlobIter<int> gbegin = x(0).end();

// local iterator to local range.
int * lbegin = x.lbegin();
int * lend   = x.lend();
```

Listing 4. The Coarray elements can be iterated on both local and global ranges.

2) Local Types: Multidimensional array access is provided by returning a `dash::NArray` view on the first bracket operator. Hence, each further bracket access on a view object reduces the number of free dimensions by one, until a `GlobRef<T>` is returned for the last dimension. For performance reasons, there is a specialization for local accesses on 1-dimensional Coarrays: Instead of returning a local view, a native pointer to the requested element is returned. To avoid the overhead of chained view proxies (`coarr[x][y]...`), a `coarr.at(x,y,...)` method is provided for direct element access. The interface for local accesses using the bracket operator is provided in Listing 5.

```
Coarray<T>::operator[]:      deleted
Coarray<T[]>::operator[]:
    (const index_type & i) -> & T
Coarray<T[a]...>::operator[]:
    (const index_type & i) -> local_view_type
```

Listing 5. The return type of the bracket operator depends on the rank of the Coarray.

3) Copointers: A global pointer / copointer is expected to behave similar to a native pointer, but can point to an arbitrary location in the global address space. Dereferencing a global pointer should return a reference to the value at its location. This has to be a global reference, as the location of the value might be remote. Likewise, getting the address of an element could be done using the ampersand (`&`) operator. However in a PGAS scenario the return value of this operation is not well-defined: It could be the memory address of the local copy, or a global pointer. While the local address can be passed to STL algorithms (pointers are random access iterators) this only works for local data. DASH distinguishes between global references `GlobRef<T>`, global pointers `GlobPtr<T>` and global iterators `GlobIter<T>`. Here, global pointers behave similar to native pointers, but can point to any location in the global memory space. This also implies that they do not have any knowledge about the container layout. As this differentiation between different types of addresses often confuses programmers [1], we do not overwrite the ampersand operator.

Instead, we provide global iterators which can be used in both DASH and STL algorithms. For convenient access we provide views to access parts of the Coarray. There, each N-D slice (except the last, 0-D) of the Coarray is itself a view, providing `begin()` and `end()` iterators. An example of this interface is given in Listing 6:

```
#include <algorithm>
Coarray<int[100]> x;

auto a = x(2)[0]; // last dim => GlobRef<int>
auto b = x(2);    // NArrayView<...>

auto begin = x(2).begin();  //GlobIter<...>
auto end   = x(2).end();
std::fill(begin, end, 42);
```

Listing 6. Using the DASH Coarray in STL algorithms by getting iterators from the Coarray container.

For technical details, we refer to the `dash::NArray` concept. [2]

4) Atomic Accesses: The `dash::Coarray` is overloaded for atomic types such that atomic operations are used when accessing and modifying elements. Conveniently we get the atomic support for free, as all elements are accessed using global references. These are specialized for the atomic wrapper type `dash::Atomic<T>`. Here, we mimic the interface of `std::atomic` as close as possible. However due to the distributed memory we cannot use it directly as internally remote data is copied to a local buffer before accessing it. The accesses to these elements are performed using the corresponding atomics interface of the communication backend.

An array of atomics is specified by wrapping the element type with `dash::Atomic<T>` as shown in Listing 7.

```
dash::Coarray<dash::Atomic<int>[10]> atomic_arr;
// atomic_arr(i) -> GlobRef<dash::Atomic<int>>
int a = atomic_arr(0)[0].compare_exchange(0, 2);
```

Listing 7. Arrays of atomics can be placed in the Coarray. Then the global references are specialized for atomic operations.

C. Synchronization

1) Team Support: Teams are groups of units which work together on a certain task. Collective operations can then be executed on a team, which is beneficial for large scale systems as not all units have to participate in this operation. In CAF 2008 there is no concept of teams, but research has shown that most algorithms can be implemented efficiently using teams. The alternative CAF 2.0 implementation, proposed in [7] also introduces teams.

As DASH entirely follows the team concept [4], all synchronization of this Coarray implementation is also based on teams. In the trivial case, this is `dash::Team::All()`, which contains all units. However for compatibility with CAF legacy code we provide a `sync_images(list_of_images)` function. Using this mechanism, a synchronization of a group of units selected by their unit id is possible.

D. Mutual Exclusion

For synchronizing global accesses on a Coarray, a distributed mutex, called `dash::Comutex` is introduced. In contrast to the interface proposed by Cray, we do not specialize the Coarray for special types like Mutex. Doing so, would change the semantics of the Coarray, which leads to problems similar to `std::vector<bool>`. Furthermore a multidimensional `Coarray<Mutex[2]>` would be problematic. Our `dash::Comutex` provides a similar interface to the Coarray and is also compatible with `std::lock_guard`, so scoped locks are possible. Support for local accesses is not provided, as there is no valid use case. A common usage is shown in Listing 8.

```
Comutex comx;
comx(2).lock();
comx(2).unlock();
// ...
{
  // mutual exclusion on unit 1
  std::lock_guard<dash::Mutex> lg(comx(1));
}
```

Listing 8. The Comutex provides one `dash::Mutex` per unit which can be used like a `std::mutex`.

E. Coevents

Events have been proposed for introduction in the CAF standard. Our implementation provides a simplistic but powerful API based on the Coarray syntax:

```
Coevent coevt;
// assume running with at least 3 units
if(this_image() < 3){
  coevt(2).post();
}
if(this_image() == 2){
  // wait for 3 incoming dependencies
  coevt.wait(3);
}
```

Listing 9. This example shows a simplistic producer-consumer pattern using events.

Events can be posted from any image to any image, but waiting for events is only supported locally. This limitation enables almost perfect scalability as no scheduler or worker thread is necessary. Furthermore this solution can be implemented using pure one-sided communication. In our case this is achieved by an event counter per unit which is incremented atomically when posting events. The `wait()` call is then busy waiting until the desired number of incoming events has been posted. Then the counter is reset and the call returns. Internally this is done using `compare_and_swap()` on the counter.

Limitations: There is currently no possibility to dispatch on the source of an event. In addition the user is responsible for not over-posting a target: As there is no concept of epochs, it is not allowed to post more events to a target than the target waits for. Before new events are allowed to be posted to a target, it has to be ensured that the wait on the target has been passed. However to avoid cases like that, multiple `dash::Coevent` instances can be used.

It is also possible to test how many events have already arrived, using `test()`. This can be used together with multiple `dash::Coevent` instances to build task graphs. This is possible for both local and remote events. However, spinning on remote events using `test()` is not recommended as this might lead to progress problems.

F. Cofutures

Cofutures are expected to behave similar to `std::future` but internally use non-blocking communication. This feature

[2]Code documentation online at https://codedocs.xyz/dash-project/dash/

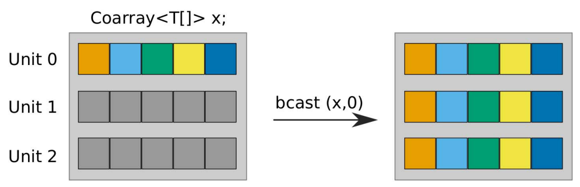


Fig. 2. Example where the local range of unit 0 is broadcasted to all other units.

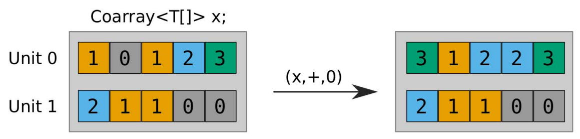


Fig. 3. Visualized version of a coreduce with `dash::Plus<T>` as reduce operation. The results are then collected at unit 0.

is essential to achieve good performance. For single elements this is implemented using asynchronous global references. For ranges, `dash::copy_async` can be used, which returns a `dash::future` for waiting for completion. This follows the STL concepts and internally copies the elements in chunks instead of one at a time. Support for range-references, which encapsulate a number of elements has be considered, but in our opinion this has no benefit over using explicit copy.

```
Coarray<int> x;
// blocking get, non-blocking put
int z = x + 1;

Coarray<int[10]> coarr;
std::vector<int> local_data(10);
// asynchronous copy returns a future object
auto fut = dash::copy_async(coarr(2).begin(), coarr(2).end
     (), local_data.data());
// do some heavy computation
fut.wait();
```

Listing 10. Example showing the two types of future objects: Asynchronous element accesses and copy operations.

IV. Collective Operations

A. Collectives

The collective operations are modeled after their Fortran equivalents and are implemented using the corresponding collective operations of DART. This includes:

cobroadcast: Broadcasts the local part to all other images. While in CAF 2008 only scalar Coarrays can be broadcast, this specification provides support for broadcasting the whole local range. If no master unit is set, the values of unit 0 are broadcast.

coreduce: Performs a broadside reduction of the local parts of all units using a DASH reduce operation. The interface allows the user to specify which unit receives the result of the operation. If no unit is passed, the result is broadcast to all units.

B. Fortran Style Syntactic Sugar

To provide Fortran programmers a familiar interface, we define aliases for common Fortran functions. This also makes the porting of applications simpler.

```
dash::Coarray::this_image() == dash::myid(); // (global)
dash::Coarray::num_images() == dash::size(); // (global)
dash::Coarray.sync_all()    == container.barrier();
```

Listing 11. Coarray functions and their native DASH equivalents.

C. Type parameter syntax

As the interface is based on the Cray proposal, we follow the type parameter syntax as well. Hence, array types are declared similar to their C / C++ equivalents by using `T[n][...]`. The conversion to the `dash::NArray` syntax is done at compile time using meta programming.

D. Local Memory Space

The local memory space is static and continuous which makes integration with other libraries much easier. This could be easily lessened by using dynamic allocators, as the iteration functionality is based on `GlobIter<T>`.

V. Evaluation

The following section presents our techniques for minimizing the overhead of the layered DASH architecture. Moreover, we evaluate the Coarray implementation using one mini-app from the Mantevo benchmark suite as well as a dedicated FORTRAN latency benchmark.

We conducted our measurement on three different systems: the Cray XC 40 system 'Cori' installed at LBNL as well as the IBM iDataPlex system 'SuperMUC' (phase 2) installed at the Leibniz Supercomputing Centre (LRZ). The system specifications are summarized in Table I. All codes were compiled using the Intel compiler (icc / ifort). As communication layer for the `dash::Coarray` benchmarks we use DART with MPI backend.

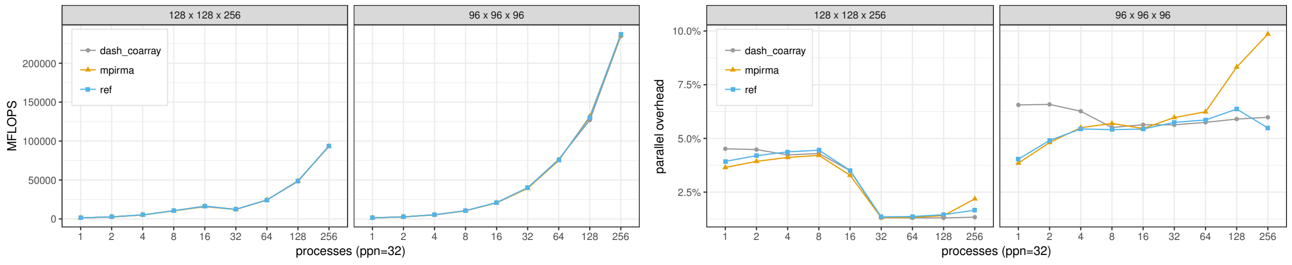
A. Implementation

Getting the highest performance in PGAS implementations is a challenging task. Following the library-based approch has the disadvantage that the compiler cannot optimize the remote accesses. However studies on UPC and UPC++ have shown that the impact of the compiler optimization is often limited to trivial cases like loops [7, p.12] [8, p.6]. These can easily be mapped to C++ functions which reduces this limitation.

A common problem of layered architectures is the performance loss between each layer due to additional function calls and data passing. As this also holds for our implementation, we carefully checked that the overhead in each layer is minimal. This is achieved by using `constexpr` and inlining wherever possible. We did some inspections of the compiler optimization reports of the Intel compiler (icc): Most performance critical parts of the Coarray layer are completely optimized away. Hence, there is only minimal overhead in the DASH / C++ part of the implementation.

TABLE I
TEST SYSTEM OVERVIEW.

System	CPU	Memory / Node	Network	Compiler	MPI
Cori (HSW)	2 x E5-2680v3 12C	128 GB	Cray Aries	ICC 18.0.0	CCE 8.5.3
Cori (KNL)	1 x Xeon Phi 7250	96 GB + 16 GB (HBM)	Cray Aries	ICC 18.0.0	CCE 8.5.3
SuperMUC	2 x E5-2697v3	64 GB	IB FDR14	ICC 18.0.0	IBM POE 1.4 & Intel 5.1



(a) Performance in MFLOPS (b) Communication overhead

Fig. 4. HPCCG Performance using Cray MPI on KNL

TABLE II
HPCCG IMPLEMENTATION COMPARISON

Setup	Matrix Storage	Communication Layer
reference	native memory	two-sided mpi
mpirma	mpi-window	one-sided mpi
DASH Coarray	dash::Coarray	DART

Another aspect which has to be considered is the overall overhead of the DASH + DART + MPI stack. MPI primary was optimized for large data transfers but not for PGAS scenarios. Hence, we provide dedicated copy algorithms and support for non-blocking puts and gets. This makes overlapping of communication and computation possible and reduces the impact of the latency added by MPI. One of these optimizations is bypassing MPI for purely local data transfers. However best performance can only be achieved by optimizing the communication pattern of the application. [9, p.3]. Combining the techniques described above, a very high application performance can be achieved, as shown in a prior evaluation of DASH in the context of block-based linear algebra algorithms [10].

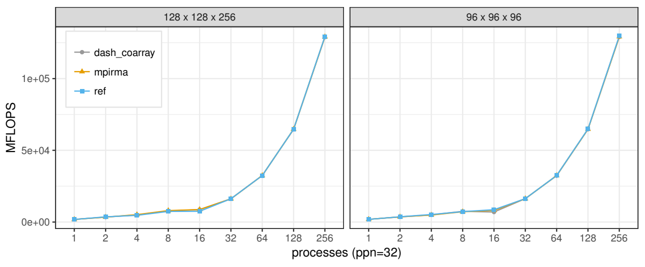
B. HPCCG

The HPCCG mini-app is a simple conjugate gradient benchmark code for a 3D chimney domain [11]. As the application has no communication-computation overlap, communication latency is exposed more prominently. Based on the reference implementation we ported the application to two different setups: a version using only MPI-RMA as well as a version using the `dash::Coarray`. A comparison is given in Table II. For all setups we performed a weak-scaling analysis for two different problem sizes. These are chosen according to the recommendations of Mantevo. The small problem uses a matrix of $96 \times 96 \times 96$ which requires ≈ 608 MB per unit. For the large problem size we decided to use 60% of the main memory of SuperMUC which results in a matrix of extents $128 \times 128 \times 256$. For both Haswell partitions we disabled OpenMP and ran the tests with one thread per MPI rank. The results show that all implementations based on MPI-RMA show no clear advantage over the reference implementation as shown in Figure 6 (SuperMUC) and Figure 5 (Cori HSW). This is almost certainly due to the synchronous nature of the data exchange. While the performance of all implementations is similar, the amount of time spent in communication varies. Here, the `dash::Coarray` implementation performs even better than the pure MPI-RMA version. This is due to optimized data-transfer strategies for block/range transfers implemented in DASH [10]. As HPCCG has no short circuit implemented for running with only one unit, there is still parallel overhead in this case.

The benchmark setup on the Knights Landing (KNL) partition of Cori is slightly different: While each KNL node exposes 68 physical cores with 4 virtual cores each, we decided to mix OpenMP and MPI to use a more light-weight parallelism. In our tests we got the best results by using 32 MPI processes with four OpenMP threads each. To keep the weak-scaling analysis fair, we also use only four OpenMP threads when running with less than 32 MPI processes. To avoid necessary code-changes, the KNL nodes are run in quad-cache mode. This also explains why the smaller problem size performs considerably better, as almost all data fits into the 16 GB HBW memory. Due to the higher memory bandwidth on KNL, the local work-packages are processed faster and hence the share of communication time increases to roughly 6% (Figure 4).

C. CAF-Bench

CAF-Bench is a benchmarking set developed for measuring the performance of various parallel operations involving For-


(a) Performance in MFLOPS

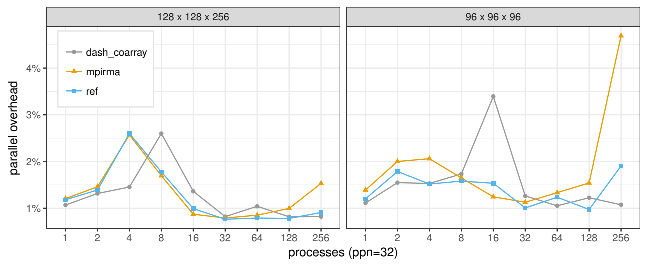
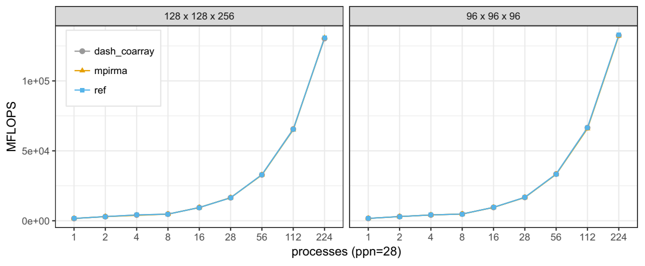

(b) Communication overhead

Fig. 5. HPCCG Performance using Cray MPI on HSW


(a) Performance in MFLOPS

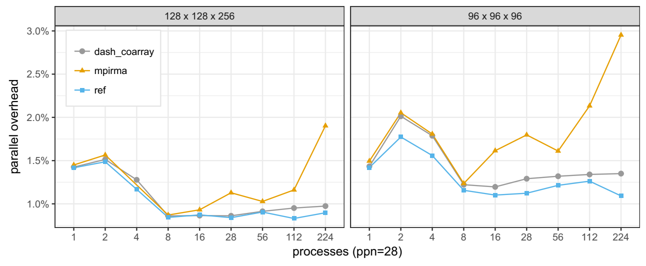
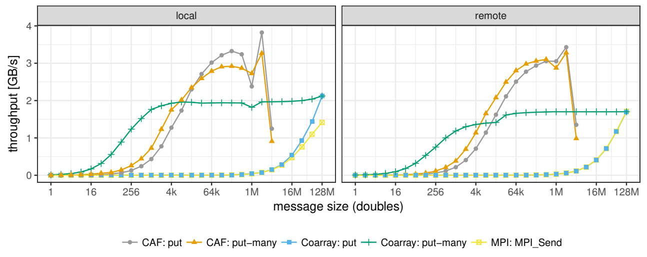

(b) Communication overhead

Fig. 6. HPCCG Performance using IBM MPI on SuperMUC


(a) Throughput when sending

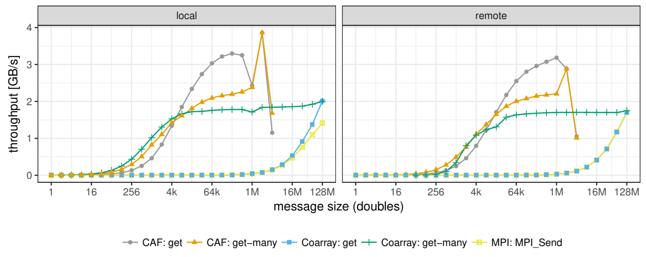

(b) Throughput when receiving

Fig. 7. CAF-Bench throughput for sending and receiving various message sizes using Intel-MPI on SuperMUC.

tran Coarrays. While it contains a bunch of tests for almost all possible Fortran data transfer patterns, we ported only the parts focusing on features which are also provided by the `dash::Coarray` implementation. This includes point-to-point (ping-pong) data transfer and various kinds of synchronisation mechanisms. For the evaluation we consider both local (intra-node) and remote (inter-node) communication: For the intra-node case we use two MPI processes (DASH) respectively two images (CAF) placed at different sockets. The remote case uses two nodes located at the same island of SuperMUC. Hence, the network-distance between them is one. Instead of a synthetic ping-pong test, we focus on a setup where multiple asynchronous accesses happen between each synchronisation phase.

The results confirm that asynchronous puts and gets have to be used to get a reasonable throughput. Both `MPI_Send` and `dash::Coarray::put` are not efficient for small data transfers, as they are blocking (synchronized) after each request. Surprisingly, for small message sizes up to 4000 doubles our Coarray implementation performs significantly better compared to the CAF version (Figure 7). Since this is a common message size for PGAS applications it is crucial. For larger transfers both CAF version perform better, however we had problems sending more than 16 million elements as the transfer did not complete at all.

VI. Related Work

CAF is a realization of the PGAS programming model of which many incarnations for other languages have been developed. Global Arrays (GA) [12] was one of the first PGAS models for C that allows the declaration of one-and multi-dimensional arrays distributed over the memory of multiple nodes. GA uses ARMCI [13] for inter-node communication, while recent efforts on porting the model to GPUs [14] and

Intel Xeon Phi Coprocessors [15] have also been reported. UPC [16] is another well known realization of the PGAS model based on an extension of the C language and custom (pre-) compilers. Many new high-productivity languages also are PGAS approaches, such as Chapel [17] and X10 [18]. More recently, PGAS programming systems have been developed based on C++. For example, UPC++ [19] is similar in spirit to UPC but requires no custom (pre-)compiler and instead uses C++ mechanisms (e.g., operator overloading).

DASH [2] has the same goals but offers a richer set of data structures and data distribution patterns than UPC++, including a true multidimensional distributed array [10] and provides interoperability with the C++ standard template library (STL). Of the efforts to bring Coarray semantics to C/C++, the Coarray implementation provided by Cray is the most complete in terms of covering the features of CAF 1.0 [1] This implementation is only available as part of the Cray Compute Environment (CCE) and relies on the proprietary Cray networking libraries. The `dash::Coarray` is in contrast a fully portable solution only relying on the availability of MPI and C++11. Xcalable MP (XMP) is another approach for realizing PGAS semantics using compiler directives for C/C++ and Fortran programs which also has provisions for Coarray semantics [20].

VII. Conclusion

In this paper we have shown an implementation of the CAF 2008 programming model in pure C and C++ without the need for special compilers. Many shortcomings of Fortran could be fixed by using the possibilities of the C++ language. Furthermore we used a more modern communication model based on teams which enables using this implementation in very large HPC systems. Combined with the other synchronization structures like events and mutexes, highly performant application code can be written. By implementing the Coarray on top of DASH, the user is able to use DASH's distributed algorithms on the Coarray as well. Various optimizations have been shown to achieve high performance in applications using the Coarray. A quantitative performance evaluation of a mini-application and a dedicated CAF benchmark showed that our C++ Coarray implementation does not lose performance due to its layered architecture.

The next steps for our work are implementing a DART version based on GASNET. This is expected to further enhance the communication performance as this is optimized for low-latency accesses.

Acknowledgment

We would like to thank the members of the DASH team and gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

[1] T. A. Johnson, "Coarray C++," in *7th International Conference on PGAS Programming Models*, 2013, p. 54.

[2] K. Fürlinger, C. Glass, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedeb, and H. Zhou, "DASH: Data structures and algorithms with support for hierarchical locality," in *Euro-Par 2014 Workshops (Porto, Portugal)*, 2014.

[3] T. Fuchs and K. Fürlinger, "Expressing and exploiting multidimensional locality in DASH," in *Software for Exascale Computing - SPPEXA 2013-2015*, H.-J. Bungartz, P. Neumann, and E. W. Nagel, Eds. Garching, Germany: Springer, 2016, pp. 341–359.

[4] H. Zhou, Y. Mhedheb, K. Idrees, C. Glass, J. Gracia, K. Fürlinger, and J. Tao, "DART-MPI: An MPI-based implementation of a PGAS runtime system," in *The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct. 2014.

[5] R. W. Numrich and J. Reid, "Co-Array Fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.

[6] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of high performance fortran," *Commun. ACM*, vol. 54, no. 11, pp. 74–82, Nov. 2011.

[7] J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and G. Jin, "A new vision for coarray fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. ACM, 2009, p. 5.

[8] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey, "Co-array fortran performance and potential: An NPB experimental study," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 177–193.

[9] D. Henty, "Performance of fortran coarrays on the cray xe6," *Cray User Group*, 2012.

[10] T. Fuchs and K. Fürlinger, "A multi-dimensional distributed array abstraction for PGAS," in *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016)*, Sydney, Australia, Dec. 2016, pp. 1061–1068.

[11] M. Heroux, "HPCCG microapp," 2007.

[12] 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*, vol. 10, no. 2, pp. 169–189, 1996.

[13] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," *Parallel and Distributed Processing*, pp. 533–546, 1999.

[14] V. Tipparaju and J. S. Vetter, "GA-GPU: Extending a library-based global address spaceprogramming model for scalable heterogeneous-computing systems," in *Proceedings of the 9th conference on Computing Frontiers*. ACM, 2012, pp. 53–64.

[15] P. Cheng, Y. Lu, T. Gao, and C. Wang, "CoGA: Extension of GA on heterogeneous system," in *Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015 IEEE 12th Intl Conf on*. IEEE, 2015, pp. 719–725.

[16] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[18] 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," in *Acm Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 519–538.

[19] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1105–1114.

[20] M. Nakao, J. Lee, T. Boku, and M. Sato, "Productivity and performance of global-view programming with XcalableMP PGAS language," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 402–409. [Online]. Available: https://doi.org/10.1109/CCGrid.2012.118