# DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms

Karl Fuerlinger, Tobias Fuchs, and Roger Kowalewski
Ludwig-Maximilians-Universität (LMU) Munich,
Computer Science Department, MNM Team,
Oettingenstr. 67, 80538 Munich, Germany
Email: first.last@nm.ifi.lmu.de

***Abstract*—We present DASH, a C++ template library that offers distributed data structures and parallel algorithms and implements a compiler-free PGAS (partitioned global address space) approach. DASH offers many productivity and performance features such as global-view data structures, efficient support for the owner-computes model, flexible multidimensional data distribution schemes and inter-operability with STL (standard template library) algorithms. DASH also features a flexible representation of the parallel target machine and allows the exploitation of several hierarchically organized levels of locality through a concept of Teams. We evaluate DASH on a number of benchmark applications and we port a scientific proxy application using the MPI two-sided model to DASH. We find that DASH offers excellent productivity and performance and demonstrate scalability up to 9800 cores.**

## I. Introduction

The PGAS (Partitioned Global Address Space) model is a promising approach for programming large-scale systems [1], [2], [3]. When dealing with unpredictable and irregular communication patterns, such as those arising from graph analytics and data-intensive applications, the PGAS approach is often better suited and more convenient than two-sided message passing [4]. The PGAS model can be seen as an extension of threading-based shared memory programming to distributed memory systems, most often employing one-sided communication primitives based on RDMA (remote direct memory access) mechanisms [5]. Since one-sided communication decouples data movement from process synchronization, PGAS models are also potentially more efficient than classical two-sided message passing approaches [6].

However, PGAS approaches have so far found only limited acceptance and adoption in the HPC community [7]. One reason for this lack of widespread usage is that for PGAS *languages*, such as UPC [8], Titanium [9], and Chapel [10], adopters are usually required to port the whole application to a new language ecosystem and are then at the mercy of the compiler developers for continued development and support. Developing and maintaining production-quality compilers is challenging and expensive and few organizations can afford such a long-term project.

*Library-based* approaches are therefore an increasingly attractive low-risk alternative and in fact some programming abstractions may be better represented through a library mechanism than a language construct (the data distribution patterns described in Sect III-B are an example). Global Arrays [11] and OpenSHMEM [12] are two popular examples for compiled PGAS libraries with a C API, which offer an easy integration into existing code bases. However, precompiled libraries and static APIs severely limit the productivity and expressiveness of programming systems and optimizations are typically restricted to local inlining of routines.

C++, on the other hand, has powerful abstraction mechanisms that allow for generic, expressive, and highly optimized libraries [13]. With a set of long awaited improvements incorporated in C++11 [14], the language has recently been used to implement several new parallel programming systems in projects such as UPC++ [15], Kokkos [16], and RAJA [17].

In this paper we describe DASH, our own C++ template library that implements the PGAS model and provides generic distributed data structures and parallel algorithms. DASH realizes the PGAS model purely as a C++ template library and does not require a custom (pre-)compiler infrastructure, an approach sometimes called compiler-free PGAS. Among the distinctive features of DASH are its inter-operability with existing (MPI) applications, which allows the porting of individual data structures to the PGAS model, and support for hierarchical locality beyond the usual two-level distinction between local and remote data.

The rest of this paper is organized as follows. In Sect. II we provide a high-level overview of DASH, followed by a more detailed discussion of the library's abstractions, data structures and algorithms in Sect. III. In Sect. IV we evaluate DASH on a number of benchmarks and a scientific proxy application written in C++. In Sect. V we discuss related work and we conclude and describe areas for future work in Sect. VI

## II. An Overview of DASH

This section provides a high level overview of DASH and its implementation based on the runtime system DART.

### *A. DASH and DART*

DASH is a C++ template library that is built on top of DART (the DAsh RunTime), a lightweight PGAS runtime system implemented in C. The DART interface specifies basic mechanisms for global memory allocation and addressing using global pointers, as well as a set of one-sided put and get operations. The DART interface is designed to abstract

```
 1 #include <libdash.h>
 2 #include <iostream>
 3 using namespace std;
 4
 5 int main(int argc, char *argv[])
 6 {
 7   dash::init(&argc, &argv);
 8   // private scalar and array
 9   int p; double s[20];
10   // globally shared array of 1000 integers
11   dash::Array<int> a(1000);
12   // initialize array to 0 in parallel
13   dash::fill(a.begin(), a.end(), 0);
14   // global reference to last element
15   dash::GlobRef<int> gref = a[999];
16   if (dash::myid() == 0) {
17     // global pointer to last element
18     dash::GlobPtr<int> gptr = a.end() - 1;
19     (*gptr) = 42;
20   }
21   dash::barrier();
22   cout << dash::myid() << " " << gref << endl;
23   cout << dash::myid() << " " << a[0] << endl;
24   dash::finalize();
25 }
```

Fig. 1. A simple stand-alone DASH program illustrating global data structures, global references, and global pointers.

from a variety of one-sided communication substrates such as GASPI [18], GASNet [19] and ARMCI [20]. DASH ships with an MPI-3 RMA (remote memory access) based implementation of DART called DART-MPI [21] that uses shared memory communicators to optimize intra-node data transfer [22]. A single-node implementation utilizing System-V shared memory has also been developed as proof-of-concept, and experimental support for GPUs was added in DART-CUDA [23].

## B. Execution Model

DASH follows the SPMD (single program, multiple data) model with hierarchical additions. In order to liberate terminology from a concrete implementation choice, we refer to the individual participants in a DASH program as **units** (cf. threads in UPC and images in Co-Array Fortran). Units may be implemented as full operating system processes or realized as lightweight threads. The total number of units is determined when the program is launched and stays unchanged throughout the program's lifetime. Units are organized into **teams** that can be dynamically created and destroyed at runtime. The sole method to establish a new team is to create a subset of an existing team starting from `dash::Team::All()`, the built-in team representing all units in a program.

In the DASH programming model, teams form the basis for all collective synchronization, communication, and memory allocation operations. Constructors for DASH data structures have an optional `Team` parameter that defaults to `dash::Team::All()`. Since a team represents physical resources (the set of memory allocations performed by the team members and their execution capabilities), `dash::Team` is implemented as a move-only type that cannot be copied.

Fig. 1 shows a simple stand-alone DASH program. `init()` initializes the runtime, `finalize()` reclaims resources. `myid()`

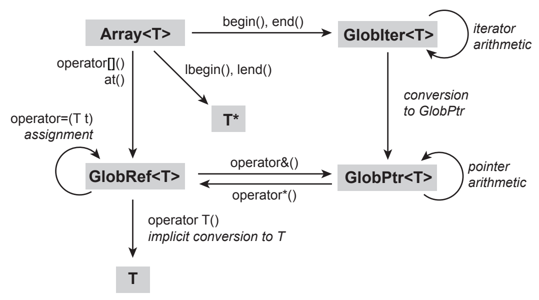

Fig. 2. The interplay of central abstractions in DASH.

is shorthand for `dash::Team::All().myid()` and returns the zero-based ID of the calling unit. Similarly, `size()` returns the number of participating units. `dash::barrier()` is shorthand for `dash::Team::All().barrier()` and synchronizes all units.

## C. Memory Model

DASH implements a *private-by-default* memory model where regular C++ variables and STL (standard template library) containers are private and cannot be accessed by other units (Fig. 1, line 9). To make data *shared* and accessible by other units, the containers provided by DASH (such as `dash::Array` and `dash::Matrix`) are allocated over a team (line 11). The team members provide the memory and can later access the shared data via one-sided put and get operations that are typically triggered automatically in response to higher-level access operations (see Sect. II-D).

The sources and targets for one-sided put and get operations are specified in terms of global pointers. A DART global pointer identifies a location in global shared memory and consists of a unit ID, a memory segment identifier, and an offset within this segment. DART global pointers are 16 bytes in size and can address any 64 bit memory location on up to $2^{32}$ units. The remaining bits are used for flags or reserved for future use.

## D. Referencing and Accessing Data

When using an STL container, such as `std::vector`, accessor functions (`.at()` and the subscript operator`[]`) return a reference to the stored value. C++ references can be thought of as named aliases, implemented using memory addresses (i.e., pointers), with additional safety guarantees enforced by the compiler. Since a DASH container holds elements that may not reside in a unit's local memory, data access cannot happen by C++ reference. Instead, accessor functions return a **global reference** proxy object of type `GlobRef<T>` where `T` is the element's data type.

`GlobRef<>` mimics C++ references and behaves in the following way: A `GlobRef<T>` is implicitly convertible to a variable of type `T`. For example, given the definition of `gref` in Fig. 1, `cout<<gref` in line 22 will output 42. Whenever a conversion of global reference to value type is requested, and the global reference denotes a remote location, a `get` operation is performed and the remote value is fetched. If the location

is local, the value is directly accessed in shared memory. Conversely, `GlobRef<>` implements an assignment operator for objects of type `T` that performs a `put` operation of the supplied value to the global memory location referenced by `GlobRef<>`.

A **global pointer** object `GlobPtr<T>` is a thin wrapper around the global pointer provided by DART. The main function of the global pointer is to specify the global memory locations for one-sided put and get operations. Dereferencing a global pointer (line 19 in Fig. 1) creates a `GlobRef<T>` object, thus `(*gptr)=42` sets the value of the last array element to 42. `GlobPtr<>` also supports pointer arithmetic and subscripting, but this only acts on the address part of the global pointer, while the unit ID remains unchanged. In other words, a global pointer does not have any *phase* information associated with it and cannot be used to iterate over a distributed array directly. **Global iterators** are used for that purpose instead. A global pointer `GlobPtr<T>` can be converted to a regular pointer (`T*`) if the memory is local, otherwise `nullptr` is returned.

A `GlobIter<T>` behaves like a random access iterator that can iterate over a chunk of global memory by keeping an internal integer index that can be dynamically converted on-demand to a `GlobPtr<T>`. To realize this index-to-`GlobPtr<>` conversion, the `GlobIter<T>` constructor takes two arguments: A `GlobMem<T>` object that represents a chunk of global memory and a `Pattern` object. A `Pattern` is the DASH approach to express data distribution and memory layout, more details about multidimensional patterns is provided in Sect. III-B.

A schematic illustration of the interplay of global pointers, iterators, and references in shown in Fig. 2. Note, however, that DASH users don't usually need to know or care about these implementation details. Instead, DASH is used with an interface that is familiar to most C++ developers: containers that offer subscripting and iterator-based access, and algorithms working on ranges delimited by iterators (Fig. 1, lines 13 and 23).

### *E. Teams*

A `Team` in DASH is simply an ordered set of units. A new team is always formed as a subset of an existing team, and thus a hierarchy from leaf team to the root (`dash::Team::All()`) is maintained. The simplest operation to create a new team is `team.split(n)`, which creates $n$ new teams, each with an approximately equal number of units. Teams are used to represent hierarchical configurations that arise in the hardware topology or may come from the algorithmic design of the application [24]. To reflect the machine topology, an equal-sized split will generally be inadequate. DASH thus computes so-called Locality Domain Hierarchies by integrating information from a number of sources such as PAPI, hwloc [25] and the OS. Using this mechanism it is, for example, possible to split `dash::Team::All()` into sub-teams representing the shared memory nodes on the first level and to then perform another split into sub-teams corresponding to NUMA domains on the second level. This scheme also supports hardware accelerators such as GPUs or Xeon Phi cards and DASH allows the formation of a sub-team that consists of all Xeon-Phi co-processors allocated to an application run.

These hardware-aware teams can then be used for (static) load balancing by identifying the hardware capabilities of each sub-team and adjusting the number of elements accordingly.

## III. Data Structures and Algorithms in DASH

In this section we describe the fundamental data container offered by dash, the `dash::Array`. We discuss the flexible data distribution schemes in one and multiple dimensions and the algorithms offered by DASH.

### *A. The DASH Array*

The DASH array (`dash::Array`) is a generic, fixed-size, one-dimensional container class template, similar in functionality to the built-in arrays that most programming languages offer (for process-local data) and the UPC *shared array* (for distributed data). Once constructed, the size of the array is fixed and cannot be changed. A `dash::Array` is always constructed over a team and all units in the team contribute an equal amount of memory to hold the array's data. The team used for the allocation is specified as an optional constructor parameter. If no team is explicitly given, the default team, `dash::Team::All()` is used. For example:

```
// globally shared array of 1000 integers
dash::Array<int> arr1(1000);

dash::Team& t1 = ...; // construct a new team
// arr2 is allocated over team t1
dash::Array<int> arr2(1000, t1);
```

The construction of a DASH array is a collective operation. All units have to call the constructor with the same arguments and it is an error if different arguments are supplied. Besides the template parameter that specifies the type of the container elements, the array constructor takes at least one argument: the total number of elements in the array (its global size). The default data distribution scheme used by DASH is `BLOCKED`, which means that each unit stores at most one contiguous block of elements of size $N_{elements}/N_{units}$ rounded up to the next integer.

Optionally, one of the distribution specifiers `BLOCKED`, `CYCLIC`, `BLOCKCYCLIC()` can be supplied explicitly, where `CYCLIC` is an alias for `BLOCKCYCLIC(1)`. As an example, when run with four units, the following declarations give rise to the distribution patterns shown in Fig. 3.

```
dash::Array<int> arr1(20); // default: BLOCKED

dash::Array<int> arr2(20, dash::BLOCKED)
dash::Array<int> arr3(20, dash::CYCLIC)
dash::Array<int> arr4(20, dash::BLOCKCYCLIC(3))
```

**Accessing the elements of a `dash::Array`.** There are various ways in which elements in a DASH array can be accessed. DASH implements a *global-view* PGAS approach in which global data structures are accessed by global indices and iterators. In other words, the expression `a[77]` refers to the same element in the array `a`, regardless of which unit evaluates the expression. Global-view programming has the appealing property that standard sequential algorithms

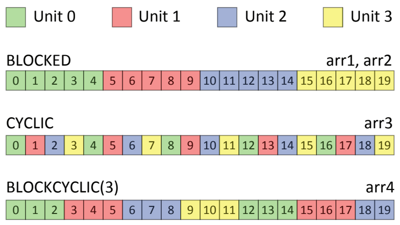

Fig. 3. The one-dimensional data distribution patterns supported by DASH.

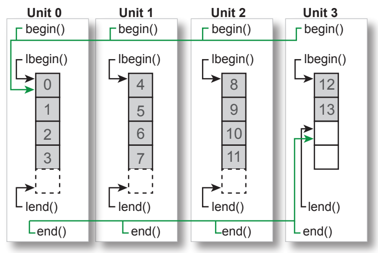

Fig. 4. Distributed memory layout for a DASH Array with 14 elements distributed over four units. The array supports global-view iteration using begin() and end() as well as local view iteration using lbegin() and lend().

can be used directly with the dash array. For example, `std::sort(a.begin(),a.end())` will employ a standard sequential sort algorithm to sort the elements in the dash array.

Global element access is supported by accessor functions (`at()` and `operator[]()`) and through global iterators. Following established STL conventions, `arr.begin()` returns a global iterator to the first element and `arr.end()` is an iterator to one-past-the-last element in the array `arr`. Thus, `dash::Array` works seamlessly with the C++11 range-based for loops, so `for(auto el: arr)cout << el;` prints all elements of `arr`.

For performance reasons it is critically important to take locality into account when working with data and all PGAS approaches support an explicit notion of data locality in some form. DASH adopts the concept of **local view proxy objects** to express data locality on the unit level. In addition to this standard two-level differentiation (local vs. remote) DASH also support a more general hierarchical locality approach. This is discussed in more detail in Sect. II-E.

The local proxy object `arr.local` represents the part of the array `arr` that is stored locally on a unit (i.e., the local view of the array). `arr.local.begin()` and `arr.local.end()`, or alternatively `arr.lbegin()`, and `arr.lend()` provide raw pointers to the underlying storage for maximum performance. These raw pointers can be used to interface with existing software package such as mathematical libraries. Note that the local proxy object does not respect the global element ordering (as specified by an array's pattern). `local[2]` is simply the third element stored locally and depending on the pattern it will correspond to a different global element. If this global information is required, a pointer to the global proxy object can be converted to a global pointer and to a global iterator.

Fig. 4 illustrates this concept for a distributed array with 14

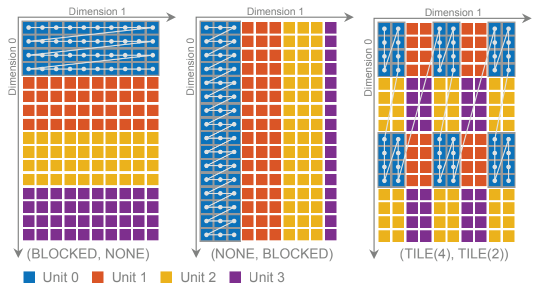

Fig. 5. Visualization of three $16 \times 10$ 2D data distribution patterns available in DASH using four units. The different colors correspond to the different units. For unit 0 the visualization additionally included the memory storage order (white dots and connecting lines).

elements distributed over four units. Each unit gets a block of four elements, the last unit's storage is underfilled with only two elements. `begin()` and `end()` return the same global iterator for each unit, `lbegin()` and `lend()` return unit-local begin and end iterators, respectively.

## B. Working with Multidimensional Data

Besides the one-dimensional array described in Sect. III-A, DASH also supports efficient and productive work with multidimensional data by providing the DASH N-dimensional array (available as `dash::NArray` and the alias `dash::Matrix`). `dash::Matrix` is a distributed N-dimensional array container class template. Its constructor requires at least two template arguments, one for the element type (`int`, `double`, etc.) and one for the dimension (N). The following example creates a two-dimensional integer matrix with 40 rows and 30 columns distributed over all units.

```
dash::Matrix<int, 2> matrix(40, 30); // 1200 elements
```

Just like the distributed 1D array, `dash::Matrix` offers efficient global and local access methods (using coordinates, linear indices, and iterators) and allows for slicing and efficient construction of block regions and lower-dimensional sub-matrix views. Details about the DASH multidimensional array and a case study implementing linear algebra routines with performance results rivaling highly tuned linear algebra packages are presented in a publication under review [26].

As an extension to the one-dimensional case, DASH allows the specification of multidimensional data distribution patterns. The distribution specifiers `CYCLIC`, `BLOCKCYCLIC()`, `BLOCKED`, and `NONE` can be used in one more dimensions and `NONE` means that no distribution is requested in a particular dimension. The following example creates two 2D patterns, each with $16 \times 10$ elements. The resulting pattern (assuming four units) is visualized in Fig. 5 (left and middle).

```
dash::Pattern<2> pat1(16, 10, BLOCKED, NONE);
dash::Pattern<2> pat2(16, 10, NONE, BLOCKED);
```

In addition, DASH supports *tiled* patterns where elements are distributed in contiguous blocks of iteration order. Additionally for any multidimensional pattern the memory layout (or storage order) can be specified using the `ROW_MAJOR` or

`COL_MAJOR` template arguments. If not explicitly specified, the default is row major storage. Fig 5 (right) shows the following tiled pattern with column major storage order as an example:

```
// arrange team in a 2x2 configuration
dash::TeamSpec<2> ts(2,2);

// 4x2 element tiles, column major layout
dash::TilePattern<2, COL_MAJOR> ↙
      pat3(16, 10, TILE(4), TILE(2), ts);
```

Note that `dash::NArray` and `dash::Pattern` support arbitrarily large dimensions (barring compiler limitations when instantiating the templates) but provide specializations and optimizations for the common two- and three-dimensional cases.

## C. *DASH Algorithms*

DASH is first and foremost about data structures, but data structures by themselves are of limited use. Of course, developers are interested in running some computation on the data to achieve results. Often this computation can be composed of smaller algorithmic building blocks. This concept is supported elegantly in the STL with its set of iterator-based standard algorithms (`std::sort()`, `std::fill()`, etc.).

DASH generalizes core underlying STL concepts: Data containers can span the memory of multiple nodes and global iterators can refer to anywhere in this virtual global address space. It is thus natural to support parallel DASH equivalents for STL algorithms. An example for this is `dash::min_element()` which is passed two global iterators to delineate the range for which the smallest element is to be found. While conceptually simple, a manual implementation of `min_element` for distributed data can actually be quite tedious, let alone repetitive.

The DASH algorithm building blocks are collective, i.e., all units that hold data in the supplied range participate in the call. The algorithms work by first operating locally and then combining results as needed. For `min_element` the local minimum is found using `std::min_element()` and then the global minimum is determined using a collective communication operation and finally the result is broadcast to all participating units. The beauty of the implementation lies in the fact that it will work in the same way with any arbitrary range in a DASH container, with any underlying data distribution and not just with simple data types but with composite data types as well.

The algorithms presently available in DASH include `copy()`, `copy_async()`, `all_of()`, `any_of()`, `none_of()`, `accumulate()`, `transform()`, `generate()`, `fill()`, `for_each()`, `find()`, `min_elment()`, and `max_element()`.

## IV. Evaluation

In this section we evaluate DASH on a number of benchmarks and perform a study with a scientific C++ MPI proxy application. The platforms we use in our study are as follows:

*a) IBEX:* A two-socket shared memory system with Intel E5-2630Lv2 (Ivy Bridge-EP) CPUs, 12 physical cores total, 15 MB L3 cache per CPU and 64 GB of main memory.

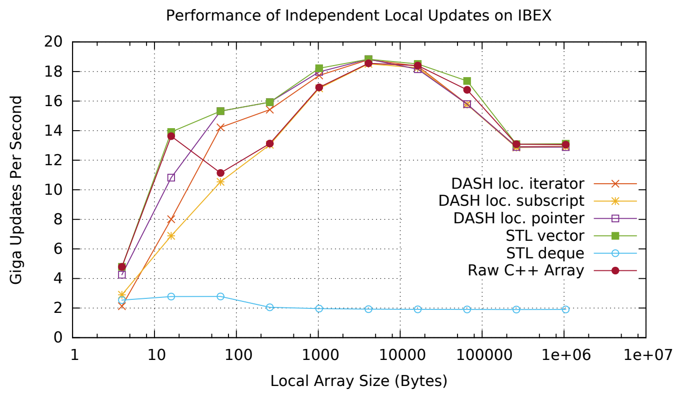

Fig. 6. Efficiency of local update operations in DASH using local subscripts, iterators and pointers, compared to raw array operations and the STL vector and deque.

*b) SuperMUC-HW:* Phase II of SuperMUC at the Leibniz Supercomputing Centre (LRZ) consisting of 3072 nodes, each equipped with two E5-2697v3 CPUs (Haswell-EP) with a total of 28 physical cores per node, 18 MB L3 cache per CPU and 64 GB of memory per node. The nodes are interconnected by an Infiniband FDR14 network.

## A. *Efficient Local Data Access*

This micro-benchmark tests how fast data can be accessed locally. The benchmark allocates a `dash::Array` of integers and initializes each element to 0. Then $N$ rounds of updates are performed, where in each round each element is incremented by 1. Each unit updates the elements it owns ("owner computers") and the total rate of updates per second is reported by the benchmark in GUPS (giga updates per second). Since data is only accessed locally, communication is not an issue for this benchmark and we report data for a single node system (IBEX).

Fig. 6 shows the results achieved for this benchmark. The horizontal axis shows different local array sizes while the vertical axis plots the achieved update rate for a number of DASH access variants and, for comparison, the `std::vector` and `std::deque` container and a raw array (`int[local_size]`). For DASH we test access to local data by local subscript (`.local[i]`), local iterator, and local pointer. As can be seen in Fig. 6, the performance of all these access methods closely matches the performance of the raw array accesses and the very well performing `std::vector` case. `std::deque` shows much lower performance because this container is not a optimized for access through the subscript operator.

## B. *Algorithms and Scalability*

In this benchmark we evaluate the performance and scalability of the DASH algorithm `dash::min_element()`.

The chart in Fig. 7 shows the performance of `dash::min_element()` on SuperMUC. A `dash::Array<int> arr` of varying size is allocated using an increasingly large number of cores, and `dash::min_element(arr.begin(), arr.end())`

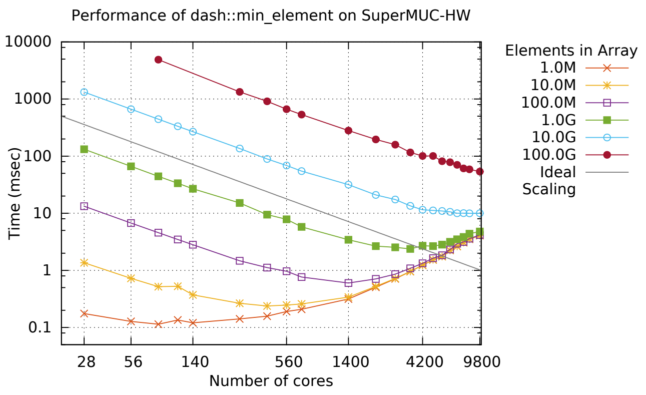

Fig. 7. Performance of the DASH min_element() algorithm on arrays of up to 100 giga elements (400 GB) and up to 9800 cores.

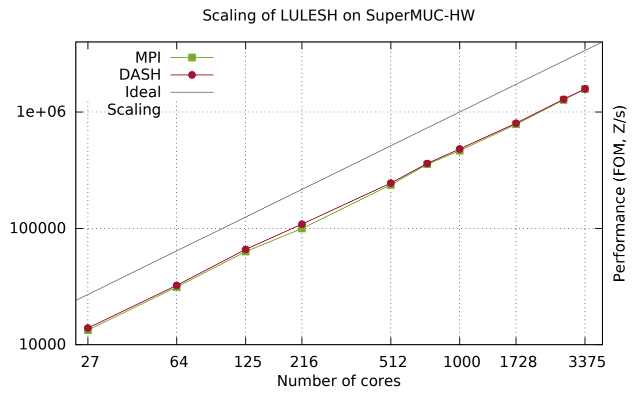

Fig. 8. Performance and scalability (weak scaling) of LULESH, implemented in MPI and DASH.

is called. I.e., the smallest element in the whole range is found. The topmost line represents an array of 100 billion entries (i.e., 400 GB of data) and the algorithm scales nicely up to 9800 cores (350 nodes) of SuperMUC. At the largest scale, finding the smallest entry takes about 50 milliseconds. For smaller arrays, the performance is dominated by communication and larger core counts increase the runtime.

### C. Communication Intensive Applications – NPB DT

The NAS Parallel Benchmarks DT kernel is a communication intensive benchmark where a data flow has to be processed in a task graph. While the initial data sets are randomly generated, the task graphs are quad trees with a binary shuffle. Depending on the problem size, the data sets fit into the L1-cache (class S) and grow with higher problem classes. Since the task graph requires frequent synchronization between the tasks, the crucial factor is a high communication throughput. In the DASH implementation, the data sets are placed in a globally distributed DASH Array which enables to use the `dash::copy_async` algorithm for the large data transfers. Due to the efficient one-sided put operations we achieve up to 24% better performance on the SuperMUC, compared to the native MPI implementation:

| Class | Graph | Size | Mop/s MPI | Mop/s DASH | Speedup |
|---|---|---|---|---|---|
| A | BH | 442368 | 170.80 | 175.16 | 1.03 |
| A | SH | 442368 | 430.50 | 486.33 | 1.13 |
| A | WH | 442368 | 313.02 | 387.47 | 1.24 |
| B | BH | 3538944 | 210.34 | 215.02 | 1.02 |
| B | SH | 3538944 | 776.38 | 905.96 | 1.17 |
| B | WH | 3538944 | 463.20 | 459.91 | 0.99 |

### D. LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a widely used proxy application for calculating the Sedov blast problem [27] that highlights the performance characteristics of unstructured mesh applications.

We ported LULESH v2.0.3 to DASH and perform a comparative performance and scalability study. The Livermore MPI implementation keeps data (coordinates, velocities, accelerations, etc.) in `std::vector` containers and uses two-sided message passing to exchange data along connecting faces, edges, and corners with up to 26 neighbors in a 3D cube domain decomposition.

For our DASH port we place all data in globally distributed 3D `dash::Matrix` containers, arrange units in a 3D cubic topology and use data distribution scheme that is `BLOCKED` in each dimension. For a cubic number of processes this results in the same decomposition used in the original version but requires far less application-side code (index calculations, etc.), since DASH takes care of these details. DASH also has the advantage that the data distribution is not limited to cubic number of processes ($n^3$) but any number $n \times m \times p$ of units can be used. We further replaced all two-sided communication operations in the original version with asynchronous one-sided put operations (using `dash::copy_async`) that directly update the target unit's memory. Fig. 8 shows the performance and scalability comparison (using weak scaling) of the two versions on up to 3375 cores. DASH scales similarly well as the optimized MPI implementation and offers performance advantages of up to 9%.

## V. Related Work

The majority of scalable HPC applications use the message passing programming model in the form of MPI today. However, several factors will make it problematic to scale MPI to future extreme scale machines. First, MPI applications usually require some degree of data replication, which conflicts with the trend of continually shrinking available memory per compute core [28]. Second, MPI applications typically implement a bulk-synchronous execution model and a more locally synchronized dynamic task-based execution approach is not easily realized in this model. Third, while MPI forces the programmer to think about data locality and thus leads to well performing code, it doesn't make it easy to work with data in a productive way – e.g., by providing higher level data structure abstractions and supporting a global view across compute nodes.

PGAS approaches make working with data in the above sense easier. PGAS essentially brings the advantages of threading-based programming (such as global visibility and accessibility of data elements) to distributed memory systems and accounts for the performance characteristics of data accesses by making the locality properties explicitly available to the programmer. Traditional PGAS approaches come in the form of a library (e.g., OpenSHMEM [12], Global Arrays [11]) or language extensions (Unified parallel C, UPC [8], Co-Array Fortran, CAF [29], [30]). Those solutions usually don't address hierarchical locality and offer only a two-level (local/remote) distinction of access costs. In contrast, DASH offers the concept of teams that can be used to express hierarchical organization of machines or algorithms.

Most programming systems also offer only one-dimensional arrays as their basic data-type out of which more complex data structures can be constructed – but that work falls on the individual programmer. More modern PGAS languages such as Chapel [10] and X10 [31] address hierarchical locality (e.g., in the form of locales or hierarchical place trees [32]) but using these approaches requires a complete re-write of the application. Given the enormous amounts of legacy software, complete rewrites of large software packages are unlikely to happen. In contrast, DASH offers an incremental path to adoption, where individual data structures can be ported to DASH while leaving the rest of the application unchanged.

Data structure libraries place the emphasis on providing data containers and operations acting on them. Kokkos [16] is a C++ template library that realizes multidimensional arrays with compile-time polymorphic layout. Kokkos is an efficiency-oriented approach trying to achieve performance portability across various manycore architectures. While Kokkos is limited to shared memory nodes and does not address multi-level machine organization, a somewhat similar approach is followed by Phalanx [33], which also provides the ability to work across a whole cluster using GASNet as the communication backend. Both approaches can target multiple back-ends for the execution of their kernels, such as OpenMP for the execution on shared memory hardware and CUDA for execution on GPU hardware. RAJA [17] and Alpaka [34] similarly target multiple backends for performance portability on single shared-memory systems optionally equipped with accelerators. RAJA, Alpaka and Kokkos are all restricted to a single compute node while DASH focuses on data structures that span the memory of many compute nodes.

STAPL [35] is a C++ template library for distributed data structures supporting a shared view programming model. STAPL doesn't appear to offer a global address space abstraction and can thus not be considered a bona-fide PGAS approach but it provides distributed data structure and a task-based execution model. STAPL offers flexible data distribution mechanisms that do however require up to three communication operations involving a directory to identify the home node of a data item. PGAS approaches in HPC usually forgo the flexible directory-based locality lookup in favor of a statically computable location of data items in the global address space for performance reasons. STAPL appears to be a closed-source project not available for a general audience.

Recently, C++ has been used as a vehicle for realizing a PGAS approach in the UPC++ [15] and Co-array C++ [36] projects. Co-array C++ follows a strict local-view programming approach and is somewhat more restricted than DASH and UPC++ in the sense that it has no concept of teams to express local synchronization and communication. While our previous work on the DASH runtime is based on MPI, UPC++ is based on GASNet. DASH offers support for hierarchical locality using teams, which are not supported by UPC++ and DASH more closely follows established C++ conventions by providing global and local iterators. STL algorithms can thus be applied directly on DASH data containers, which is not possible in UPC++. DASH also comes with a set of optimized parallel algorithms akin to those found in the STL while UPC++ offers no such algorithms. DASH also supports globalview multidimensional distributed arrays with flexible data distribution schemes, UPC++ only supports a local view multidimensional array inspired by Titanium [37].

## VI. Conclusion and Future Work

We have presented DASH, a compiler-free PGAS approach implemented as a C++ template library. Using one-sided communication substrates such as MPI-3 RMA, DASH offers distributed memory data structures that follow established C++ STL conventions and thus offer an easy adoption path for C++ developers. DASH can be integrated into existing applications by porting individual data structures at a time. DASH simplifies working with multidimensional data by providing a multidimensional array abstraction with flexible data distribution schemes. DASH also accounts for increasingly complex machine topologies by providing a Team concept that can be used to express hierarchical locality. Our experimental evaluation has shown that DASH scales well and is able to outperform classic two-sided MPI applications.

DASH is free open-source software, released under a BSD license and is under active development. The current version of the library can be downloaded from the project's webpage at http://www.dash-project.org.

Further developments for DASH are planned in several directions. First, work is under way to support dynamically growing and shrinking containers. While classical HPC applications can typically be well supported by the fixed-size containers currently implemented in DASH, data analytics applications and HPC use cases in less traditional areas such as computational biology often require these more complex data structures. Second, DASH focuses on data-structures and provides efficient support for the owner-computes model but it doesn't currently offer a way to apply computation actions to data elements in a more general way. Work towards this functionality is planned by implementing a general task-based execution model in the runtime and the C++ template library.

## Acknowledgements

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

## References

[1] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill *et al.*, "Exascale software study: Software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[2] A. Gómez-Iglesias, D. Pekurovsky, K. Hamidouche, J. Zhang, and J. Vienne, "Porting scientific libraries to PGAS in XSEDE resources: Practice and experience," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: ACM, 2015, pp. 40:1–40:7.

[3] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, ser. PASCO '07. New York, NY, USA: ACM, 2007, pp. 24–32.

[4] J. Jose, S. Potluri, K. Tomko, and D. K. Panda, "Designing scalable graph500 benchmark with hybrid MPI+OpenSHMEM programming models," in *International Supercomputing Conference*. Springer, 2013, pp. 109–124.

[5] G. Almasi, "PGAS (partitioned global address space) languages," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1539–1545.

[6] R. Belli and T. Hoefler, "Notified access: Extending remote memory access programming models for producer-consumer synchronization," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 871–881.

[7] "PRACE second implementation project. Training and education survey," december 2011. Available at http://www.prace-ri.eu/IMG/zip/d4.1_2ip.zip.

[8] UPC Consortium, "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005. [Online]. Available: http://www.gwu.edu/~upc/publications/LBNL-59208.pdf

[9] A. Aiken, P. Colella, D. Gay, S. Graham, P. Hilfinger, A. Krishnamurthy, B. Liblit, C. Miyamoto, G. Pike, L. Semenzato *et al.*, "Titanium: A high-performance java dialect," *Concurr. Comput*, vol. 10, no. 11-13, pp. 825–836, 1998.

[10] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, August 2007.

[11] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: a portable shared-memory programming model for distributed memory computers," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, 1994, pp. 340–349.

[12] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, p. 2.

[13] L.-Q. Lee and A. Lumsdaine, "Generic programming for high performance scientific applications," in *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. ACM, 2002, pp. 112–121.

[14] International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), ISO/IEC 14882:2011, Standard for programming language C++, 2011.

[15] 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*, 2014, pp. 1105–1114.

[16] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[17] R. Hornung, J. Keasler *et al.*, "The RAJA portability layer: overview and status," *Lawrence Livermore National Laboratory, Livermore, USA*, 2014.

[18] D. Grünewald and C. Simmendinger, "The GASPI API specification and its implementation GPI 2.0," in *7th International Conference on PGAS Programming Models*, Edinburgh, Scotland, 2013.

[19] D. Bonachea, "GASNet specification, v1." *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.

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

[21] 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, pp. 3:1–3:11.

[22] H. Zhou, K. Idrees, and J. Gracia, "Leveraging MPI-3 shared-memory extensions for efficient PGAS runtime systems," in *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, 2015, pp. 373–384.

[23] L. Zhou and K. Fürlinger, "DART-CUDA: A PGAS runtime system for Multi-GPU systems," in *IEEE 14th International Symposium on Parallel and Distributed Computing (ISPDC)*, Limassol, Cyprus, Jun. 2015, pp. 110 – 119.

[24] A. Kamil and K. Yelick, "Hierarchical computation in the spmd programming model," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2013, pp. 3–19.

[25] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[26] T. Fuchs and K. Fürlinger, "A multidimensional distributed array abstraction in PGAS," submitted for review at HPCC'16, copy on file with author.

[27] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 updates and changes," *Livermore, CA, August*, 2013.

[28] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal" for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[29] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, and G. Jin, "A new vision for coarray Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models (PGAS '09)*. New York, NY, USA: ACM, 2009.

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

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

[32] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical Place Trees: A portable abstraction for task parallelism and data movement," in *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Springer-Verlag, 2010, pp. 172–187.

[33] M. Garland, M. Kudlur, and Y. Zheng, "Designing a unified programming model for heterogeneous machines," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, Nov. 2012.

[34] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. KnÃijpfer, W. E. Nagel, and M. Bussmann, "Alpaka – an abstraction library for parallel kernel acceleration," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 631–640.

[35] A. Buss, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, L. Rauchwerger *et al.*, "STAPL: standard template adaptive parallel library," in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM, 2010, p. 14.

[36] T. A. Johnson, "Coarray C++," in *7th International Conference on PGAS Programming Models*, Edinburgh, Scotland, 2013.

[37] A. Kamil, Y. Zheng, and K. Yelick, "A local-view array library for partitioned global address space C++ programs," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM, 2014, p. 26.