Expressing and Exploiting Multi-Dimensional Locality in DASH

Tobias Fuchs and Karl Fürlinger

DRAFT

December 7, 2015

Abstract DASH is a realization of the PGAS (partitioned global address space) programming model in the form of a C++ template library. It provides a multi-dimensional array abstraction which is typically used as an underlying container for stencil- and dense matrix operations. Efficiency of operations on a distributed multi-dimensional array highly depends on the distribution of its elements to processes and the communication strategy used to propagate values between them. Locality can only be improved by employing an optimal distribution that is specific to the implementation of the algorithm, run-time parameters such as node topology, and numerous additional aspects. Application developers have no way of knowing these implications which also might change in future releases of DASH. In the following, we identify fundamental properties of distribution patterns that are prevalent in existing HPC applications. We describe a classification scheme of multi-dimensional distributions based on these properties and demonstrate how optimal distribution patterns can be determined automatically and, to a great extent, at compile time.

Tobias Fuchs
Ludwig-Maximilians-Universität (LMU) Munich, Computer Science Department, MNM Team
Oettingenstr. 67, 80538 Munich, Germany
e-mail: `tobias.fuchs@nm.ifi.lmu.de`

Karl Fürlinger
Ludwig-Maximilians-Universität (LMU) Munich, Computer Science Department, MNM Team
Oettingenstr. 67, 80538 Munich, Germany
e-mail: `karl.fuerlinger@nm.ifi.lmu.de`

1 Introduction

In this paper, we present a framework that enables HPC application developers to express constraints on data distribution that are suitable to exploit locality in multi-dimensional arrays.

The DASH library [7] provides numerous variants of data distribution schemes. Their implementations are encapsulated in well-defined concept definitions and are therefore semantically interchangeable. However, no single distribution scheme is suited for every usage scenario. In operations on shared multi-dimensional containers, locality can only be maintained by choosing an optimal distribution. This choice depends on:

- the algorithm executed on the shared container, in particular its communication pattern and memory access scheme,
- run-time parameters such as the extents of the shared container, the number of processes and their network topology,
- numerous additional aspects such as CPU architecture and memory topology.

The responsibility to specify a data distribution that is optimal with respect to locality and communication avoidance lies with the application developers. These, however, do not know implementation-specific implications: a specific distribution might be balanced, but blocks might not fit into a cache line, inadvertently impairing hardware locality.

As a solution, we present a mechanism to find a concrete distribution variant among all available candidate implementations that satisfies a set of properties. In effect, programmers do not need to specify a distribution type and its configuration explicitly. They can rely on the decision of the DASH library and focus only on aspects of data distribution that are relevant in the scenario at hand, unconcerned about implementation details.

For this, we first identify and categorize fundamental properties of distribution schemes that are prevalent in algorithms in related work and existing HPC applications. With DASH as a reference implementation, we demonstrate how optimal data distributions can then be determined automatically and, to a great extent, at compile time.

From a software engineering perspective, we explain how our methodology follows best practices known from established C++ libraries and thus ensures that user applications are not only robust against, but even benefit from future changes in DASH.

The remainder of this paper is structured as follows: The following section introduces fundamental concepts of PGAS and locality in the context of DASH. A classification of data distribution properties is presented in Section 3. In Section 4, we show how this system of properties allows to exploit locality in DASH in different scenarios. Publications and tools related to this work are discussed in Section 5. Finally, Section 6 gives a conclusion and an outlook on future work where the DASH library's pattern traits framework is extended to sparse, irregular, and hierarchical distributions.

2 Background

This section gives a brief introduction to the Partitioned Global Address Space approach considering locality and data distribution. We then present concepts in the DASH library used to express process topology, data distribution and iteration spaces. The following sections build upon these concepts and present new mechanisms to exploit locality automatically using generic programming techniques.

2.1 PGAS and Multi-Dimensional Locality

Conceptually, the Partitioned Global Address Space (PGAS) paradigm unifies memory of individual, networked nodes into a virtual global memory space. In effect, PGAS languages create a shared namespace for local and remote variables. This, however, does not affect physical ownership. A single variable is only located in a specific node's memory and local access is more efficient than remote access from other nodes. This is expected to matter more and more even within single (NUMA) nodes in the near future [2]. As locality directly affects performance and scalability, programmers need full control over data placement. Then, however, they are facing overwhelmingly complex implications of data distribution on locality.

This complexity increases exponentially with the number of data dimensions. Calculating a rectangular intersection might be manageable for up to three dimensions, but locality is hard to maintain in higher dimensions, especially for irregular distributions.

2.2 DASH Concepts

Expressing data locality in a Partitioned Global Address Space language builds upon fundamental concepts of process topology and data distribution. In the following, we describe these concepts as they are used in the context of DASH.

2.2.1 Topology: Teams and Units

In DASH terminology, a *unit* refers to any logical component in a distributed memory topology that supports processing and storage. Conventional PGAS approaches offer only the differentiation between local and global data and distinguish between private, shared-local, and shared-remote memory. DASH extends this model by a more fine-grained differentiation that corresponds to hierarchical machine models as units are organized in hierarchical *teams*. For example, a team at the top level could group processing nodes into individual teams, each again consisting of units referencing single CPU cores.

2.2.2 Data Distribution: Patterns

Data distributions in general implement a two-level mapping:

1. from index to process (*node-* or *process mapping*)
2. from process to local memory offset (*local order* or *layout*)

Index sets separate the logical index space as seen by the user from physical layout in memory space. This distinction and the mapping between index domains is usually transparent to the programmer.

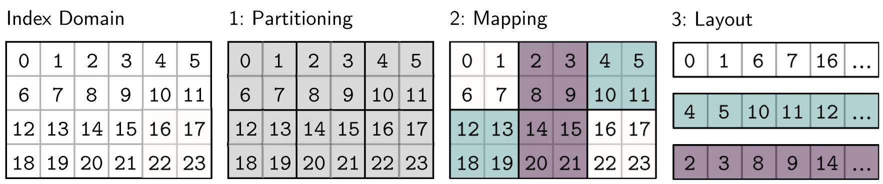

Fig. 1 Example of partitioning, mapping, and layout in the distribution of a dense, two-dimensional array.

Process mapping can also be understood as *distribution*, arrangement in local memory is also referred to as *layout* e.g. in Chapel [3].

In DASH, data decomposition is based on index mappings provided by different implementations of the DASH *Pattern* concept. Listing 1 shows the instantiation of a rectangular pattern, specifying the Cartesian index domain and partitioning scheme. Patterns partition a global index set into *blocks* that are then mapped to units. Consequently, indices are mapped to processes indirectly in two stages: from index to block (*partitioning*) and from block to unit (*mapping*). Figure 1 illustrates a pattern's index mapping as sequential steps in the distribution of a two-dimensional array. While the name and the illustrated example might suggest otherwise, blocks are not necessarily rectangular.

In summary, the DASH pattern concept defines semantics in the following categories:

Distribution Well-defined distribution of indices to units, depending on properties in the subordinate categories:

- **Partitioning** Grouping indices into blocks
- **Mapping** Distributing blocks to units in a team

Layout Arrangement of blocks and block elements in local memory

Indexing Operations related to index sets for iterating data elements in global- and local scope

Layout semantics specify the arrangement of values in local memory and, in effect, their iteration order. Indexing semantics also include index set operations like slicing and intersection but do not affect physical data distribution.

We define distribution semantics of a pattern type depending on the following set of operations:

$\mathrm{local}(i_G) \mapsto (u, i_L)$	Index i_G to unit u and local offset i_L
$\mathrm{global}(u, i_L) \mapsto i_G$	Unit u and local offset i_L to global index i_G
$\mathrm{unit}(i_G) \mapsto u$	Index i_G to unit u
$\mathrm{local\_block}(i_G) \mapsto (u, i_{LB})$	Index i_G to unit u and local block index i_{LB}
$\mathrm{global\_block}(i_G) \mapsto i_{GB}$	Index i_G to global block index i_{GB}

with n-dimensional indices i_G, i_L as coordinates in the global / local Cartesian element space and i_{GB}, i_{LB} as coordinates in the global / local Cartesian block space. Instead of a Cartesian point, an index may also be declared as a point's offset in linearized memory order.

```
// Brief notation:
TilePattern<2> pattern(global_extent_x, global_extent_y,
                       TILED(tilesize_x), TILED(tilesize_y));
// Equivalent full notation:
TilePattern<2, dash::default_index_t, ROW_MAJOR>
  pattern(DistributionSpec<2>(
              TILED(tilesize_x), TILED(tilesize_y),
          SizeSpec<2, dash::default_size_t>(
              global_extent_x, global_extent_y),
          TeamSpec<1>(
              Team::All()));
```

Listing 1 Explicit instantiation of DASH patterns.

DASH containers use patterns to provide uniform notations based on view proxy types to express index domain mappings. User-defined data distribution schemes can be easily incorporated in DASH applications as containers and algorithms accept any type that implements the Pattern concept.

Listing 2 illustrates the intuitive usage of user-defined pattern types and the `local` and `block` view accessors that are part of the DASH Container concept. View proxy objects use a referenced container's pattern to map between its index domains but do not contain any elements themselves. They can be arbitrarily chained to refine an index space in consecutive steps, as in the last line of Listing 2: the expression `array.local.block(1)` addresses the second block in the array's local memory space.

In effect, patterns specify local iteration order similar to the partitioning of iteration spaces e.g. in RAJA [8]. Proxy types implement all methods of their delegate container type and thus also provide `begin` and `end` iterators that specify the iteration space within the view's mapped domain. DASH iterators provide an intuitive notation of ranges in virtual global memory that are well-defined with respect to distance and iteration order, even in multi-dimensional and irregular index domains.

```
CustomPattern pattern;
dash::Array<double> a(size, pattern);
double g_first = a[0]           // First value in global memory,
                                // corresponds to a.begin()
double l_first = a.local[0];    // First value in local memory,
                                // corresponds to a.local.begin()
dash::copy(a.block(0).begin(),          // Copy first block in
           a.block(0).end(),            // global memory to second
           a.local.block(1).begin()); // block in local memory
```

Listing 2 Views on DASH containers

3 Classification of Pattern Properties

While terms like *blocked*, *cyclic* and *block-cyclic* are commonly understood, the terminology of distribution types is inconsistent in related work, or differ in semantics. Typically, distributions are restricted to specific constraints that are not applicable in the general case for convenience.

Instead of a strict taxonomy enumerating the full spectrum of all imaginable distribution semantics, a systematic description of pattern properties is more practicable to abstract semantics from concrete implementations. The classification presented in this section allows to specify distribution patterns by categorized, unordered sets of properties. It is, of course, incomplete, but can be easily extended. We identify properties that can be fulfilled by data distributions and then group these properties into orthogonal *categories* which correspond to the separate functional aspects of the pattern concept described in Subsection 2.2.2: partitioning, unit mapping, and memory layout. This categorization also complies with the terminology and conceptual findings in related work [13].

DASH pattern semantics are specified by a configuration of properties in these dimensions:

$$\text{Global} \times \underbrace{\text{Partitioning} \times \text{Mapping}}_{\text{Distribution}} \times \text{Layout}$$

Details on a selection of single properties in all categories are discussed in the remainder of this section.

3.1 Partitioning Properties

Partitioning refers to the methodology used to subdivide a global index set into disjoint blocks in an arbitrary number of logical dimensions. If not specified otherwise by other constraints, indices are mapped into *rectangular* blocks. A partitioning is *regular* if it only creates blocks with identical extents and *balanced* if all block has identical size.

rectangular	Block extents are constant in every single dimension, e.g. every row has identical size.
minimal	Minimal number of blocks in every dimension, i.e. at most. one block for every unit.
regular	All blocks have identical extents.
balanced	All blocks have identical size (number of elements).
cache-aligned	Block sizes are a multiple of cache line size.

Note that these properties are independent: rectangular partitionings may produce blocks with varying extents, balanced partitionings are not necessarily rectangular, and so on. For example, partitioning a matrix into triangular blocks could satisfy the *regular* and *balanced* partitioning properties. The fine-grained nature of property definitions allows many possible combinations that form an expressive and concise vocabulary to express pattern semantics.

3.2 Mapping Properties

Well-defined mapping properties exist that have been formulated to define *multipartitionings*, a family of distribution schemes supporting parallelization of line sweep computations over multi-dimensional arrays.

The first and most restrictive multipartitioning has been defined based on the *diagonal* property [12]. In a multipartitioning, each process owns exactly one tile in each hyperplane of a partitioning so that all processors are active in every step of a line-sweep computation along any array dimension as illustrated in Figure 2.

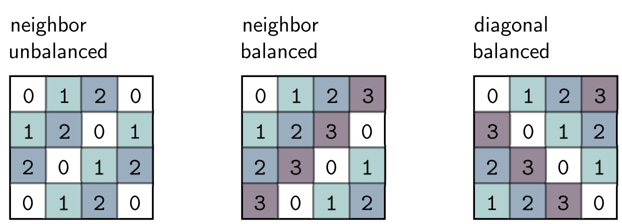

Fig. 2 Combinations of mapping properties. Numbers in blocks indicate the unit rank owning the block.

General multipartitionings are a more flexible variant that allows to assign more than one block to a process in a partitioned hyperplane. The generalized definition subdivides the original diagonal property into the *balanced* and *neighbor* mapping properties [4] described below. This definition is more relaxed but still preserves the benefits for line-sweep parallelization.

balanced	The number of assigned blocks is identical for every unit.
neighbor	A block's adjacent blocks in any one direction along a dimension are all owned by some other processor.
shifted	Units are mapped to blocks in diagonal chains in at least one hyperplane
diagonal	Units are mapped to blocks in diagonal chains in all hyperplanes.
cyclic	Blocks are assigned to processes like dealt from a deck of cards in every hyperplane, starting from first unit.

The constraints defined for multipartitionings are overly strict for some algorithms and can be further relaxed to a subset of its properties. For example, a pipelined optimization of the SUMMA algorithm requires a *diagonal shift* mapping [11, 15] that satisfies the diagonal property but is not required to be balanced. Therefore, the diagonal property in our classification does not imply a balanced mapping, deviating from its original definition.

3.3 Layout Properties

Layout properties describe how values are arranged in a unit's physical memory and, consequently, their natural iteration order. Perhaps the most crucial property is storage order which is either row- or column major. If not specified, DASH assumes row-major order as known from C. The list of properties can also be extended to give hints to allocation routines on the physical memory topology of units such as NUMA or CUDA.

row-major	Row major storage order, used by default.
col-major	Column major storage order.
blocked	Elements are contiguous in local memory within a single block.
canonical	All local indices are mapped to a single logical index domain.
linear	Local element order corresponds to a logical linearization within single blocks (tiled) or within entire local memory (canonical).
cache-aligned	Block elements are cache-aligned.

While patterns assign indices to units in logical blocks, they do not necessarily preserve the block structure in local index sets. After process mapping, a pattern's layout scheme may arrange indices mapped to a unit in an arbitrary way in physical memory. In *canonical* layouts, the local storage order corresponds to the logical global iteration order. *Blocked* layouts preserve the block structure locally such that values within a block are contiguous in memory, but in arbitrary order. The additional *linear* property also preserves the logical linearized order of elements within single blocks. For example, Morton order memory layout as shown in Figure 3 satisfies the *blocked* property, as elements within a block are contiguous in memory, but does not grant the *linear* property.

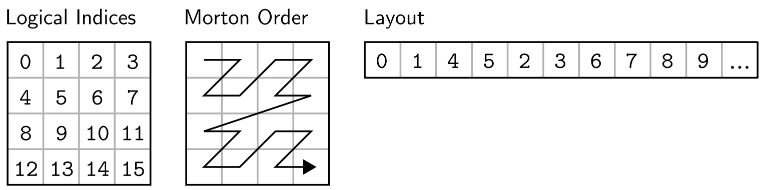

Fig. 3 Morton order memory layout of block elements.

3.4 Global Properties

The *Global* category is usually only needed to give hints on characteristics of the distributed value domain such as the *sparse* property to indicate the distribution of sparse data.

dense	Distributed data domain is dense.
sparse	Distributed data domain is sparse.
balanced	The same number of values is mapped to every unit after partitioning and mapping.

It also contains properties that emerge from a *combination* of the independent partitioning and layout properties and cannot be derived from either category separately. The global *balanced* distribution property, for example, guarantees the same number of local elements at every unit. This is trivially fulfilled for balanced partitioning and balanced mapping where the same number of blocks b of identical size s is mapped to every unit. However, it could also be achieved in a combination of unbalanced partitioning and unbalanced mapping, e.g. when assigning b blocks of size s and $b/2$ blocks of size $2s$.

4 Exploiting Locality with Pattern Traits

The classification system presented in the last section allows to describe distribution pattern semantics using properties instead of a taxonomy of types that are associated with concrete implementations. In the following, we introduce *pattern traits*, a collection of mechanisms in DASH that utilize distribution properties to exploit data locality automatically.

As a technical prerequisite for these mechanisms, every pattern type is annotated with *tag* type definitions that declare which properties are satisfied by its implementation. This enables meta-programming based on the patterns' distribution properties as type definitions are evaluated at compile time. Using tags to annotate type invariants is a common method in generic C++ programming and prevalent in the STL and the Boost library [1].

```
template <dim_t NDim, ...>
class ThePattern {
public:
  typedef mapping_properties<
            mapping_tag::diagonal,
            mapping_tag::cyclic >
    mapping_tags;
  ...
};
```

Listing 3 Property tags in a pattern type definition.

4.1 Deducing Distribution Patterns from Constraints

In a common use case, programmers intend to allocate data in distributed global memory with the use for a specific algorithm in mind. They would then have to decide for a specific distribution type, carefully evaluating all available options for optimal data locality in the algorithm's memory access pattern.

To alleviate this process, DASH allows to automatically create a concrete pattern instance that satisfies a set of constraints. The function `make_pattern` returns a pattern instance from a given set of properties and run-time parameters. The actual type of the returned pattern instance is resolved at compile time and never explicitly appears in client code by relying on automatic type deduction.

[1] http://www.boost.org/community/generic_programming.html

```
static const dash::dim_t NumDataDim = 2;
static const dash::dim_t NumTeamDim = 2;
// Topology of processes, here: 16x8 process grid
TeamSpec<NumTeamDim> teamspec(16, 8);
// Cartesian extents of container:
SizeSpec<NumDataDim> sizespec(extent_x, extent_y);
// Create instance of pattern type deduced from
// constraints at compile time:
auto pattern =
  dash::make_pattern<
    partitioning_properties<
      partitioning_tag::balanced >,
    mapping_properties<
      mapping_tag::balanced,
      mapping_tag::diagonal >,
    layout_properties<
      layout_tag::blocked >
  >(sizespec, teamspec);
```

Listing 4 Deduction of an Optimal Distribution.

To achieve compile-time deduction of its return type, `make_pattern` employs the *Generic Abstract Factory* design pattern [1]. Different from an *Abstract Factory* that returns a polymorphic *object* specializing a known base type, a Generic Abstract Factory returns an arbitrary type, giving more flexibility and no longer requiring inheritance at the same time.

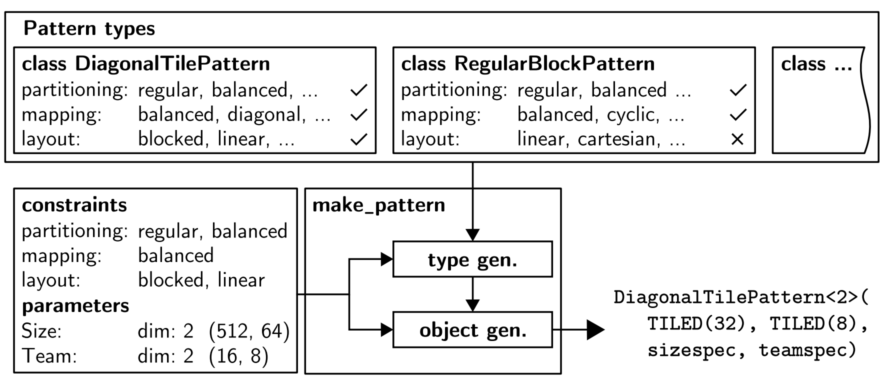

Fig. 4 Type deduction and pattern instantiation in dash::make_pattern.

Pattern constraints are passed as template parameters grouped by property categories as shown in Listing 4. Data extents and unit topology are passed as run-time arguments. Their respective dimensionality (*NumDataDim*, *NumTeamDim*), however, can be deduced from the argument types at compile time. Figure 4 illustrates the logical model of this process involving two stages: a type generator that resolves a pattern type from given constraints and argument types at compile time and an object generator that instantiates the resolved type depending on constraints and run-time parameters.

Every property that is not specified as a constraint is a degree of freedom in type selection. When more than one pattern type satisfies the constraints, the implementation with the least complex index calculation is preferred.

The automatic deduction also is designed to prevent inefficient configurations. For example, pattern types that pre-generate block coordinates to simplify index calculation are inefficient and memory-intensive for a large number of blocks. They are therefore disregarded if the blocking factor in any dimension is small.

4.2 Deducing Distribution Patterns for a Specific Use Case

With the ability to create distribution patterns from constraints, developers still have to know which constraints to choose for a specific algorithm. Therefore, we offer shorthands for constraints of every algorithm provided in DASH that can be passed to `make_pattern` instead of single property constraints.

```
dash::TeamSpec<2> teamspec(16, 8);
dash::SizeSpec<2> sizespec(1024, 1024);
// Create pattern instance optimized for SUMMA:
auto pattern = dash::make_pattern<
                 dash::summa_pattern_traits
               >(sizepec, teamspec);
// Create matrix instances using the pattern:
dash::Matrix<2, int> matrix_a(sizespec, pattern);
dash::Matrix<2, int> matrix_b(sizespec, pattern);
...
auto matrix_c = dash::summa(matrix_a, matrix_b)
```

Listing 5 Deduction of an optimal distribution pattern for a given use-case.

4.3 Checking Distribution Constraints

An implementer of an algorithm on shared containers might want to ensure that their distribution fits the algorithm's communication strategy and memory access scheme.

The traits type `pattern_constraints` allows querying constraint attributes of a concrete pattern type at compile time. If the pattern type satisfies all requested properties, the attribute `satisfied` is expanded to `true` Listing 6 shows its usage in a static assertion that would yield a compilation error if the object `pattern` implements an invalid distribution scheme.

```
// Compile time constraints check:
static_assert(
   dash::pattern_contraints<
       decltype(pattern),
       partitioning_properties< ... >,
       mapping_properties< ... >,
       layout_properties< ... >
   >::satisfied::value
);
// Run time constraints check:
if (dash::check_pattern_contraints<
        partitioning_properties< ... >,
        mapping_properties< ... >,
        indexing_properties< ... >
    >(pattern)) {
  // Object 'pattern' satisfies constraints
}
```

Listing 6 Checking distribution constraints at compile time and run time.

Some constraints depend on parameters that are unknown at compile time, such as data extents or unit topology in the current team.

The function `check_pattern_constraints` allows checking a given pattern object against a set of constraints at run time. Apart from error handling, it can also be used to implement alternative paths for different distribution schemes.

4.4 Deducing Optimal Algorithm Variants

When combining different applications in a work flow or working with legacy code, container data might be preallocated. As any redistribution is usually expensive, the data distribution scheme is invariant and a matching algorithm variant is to be found.

We previously explained how to resolve a distribution scheme that is optimal for a known specific algorithm implementation. Pattern traits and generic programming techniques available in C++11 also allow to solve the inverse problem: finding an algorithm variant that is suited for a given distribution. For this, DASH provides adapter functions that switch between an algorithm's implementation variants depending on the distribution type of its arguments. In Listing 7, three matrices are declared using an instance of `dash::TilePattern` that corresponds to the known distribution of their preallocated data. In compilation, `dash::multiply` expands to an implementation of matrix-matrix multiplication that best matches the distribution properties of its arguments, like `dash::summa` in this case.

```
typedef dash::TilePattern<2, ROW_MAJOR>    TiledPattern;
typedef dash::Matrix<2, int, TiledPattern> TiledMatrix;
TiledPattern  pattern(global_extent_x,  global_extent_y,
                        TILE(tilesize_x), TILE(tilesize_y));
TiledMatrix   At(pattern);
TiledMatrix   Bt(pattern);
TiledMatrix   Ct(pattern);
...
// Use adapter to resolve algorithm suited for TiledPattern:
dash::multiply(At, Bt, Ct); // --> dash::summa(At, Bt, Ct);
```

Listing 7 Deduction of an algorithm variant for a given distribution.

5 Related Work

Various aspects of data decomposition have been examined in related work that influenced the design of pattern traits in DASH.

The Kokkos framework [6] is specifically designed for portable multi-dimensional locality. It implements compile-time deduction of data layout depending on memory architecture and also specifies distribution traits roughly resembling some of the property categories introduced in this work. However, Kokkos targets intra-node locality focusing on CUDA- and OpenMP backends and does not define concepts for process mapping. It is therefore not applicable to the PGAS language model where explicit distinction between local and remote ownership is required.

UPC++ implements a PGAS language model and, similar to the array concept in DASH, offers local views for distributed arrays for rectangular index domains [9]. However, UPC++ does not provide a general view concept and no abstraction of distribution properties as described in this work.

Chapel's Domain Maps is an elegant framework that allows to specify and incorporate user-defined mappings [3] and also supports irregular domains. Chapel does not provide automatic deduction of optimal distribution schemes, however, and no classification of distribution properties is defined.

Finally, the benefits of hierarchical data decomposition are investigated in recent research such as TiDA, which employs hierarchical tiling as a general abstraction for data locality [14]. The Hitmap library achieves automatic deduction of data decomposition for hierarchical, regular tiles [5] at compile time.

6 Conclusion and Future Work

We constructed a general categorization of distribution schemes based on well-defined properties. In a broad spectrum of different real-world scenarios, we then discussed how mechanisms in DASH utilize this property system to exploit data locality automatically.

In this, we demonstrated the expressiveness of generic programming techniques in modern C++ and their benefits for constrained optimization.

Automatic deduction greatly simplifies the incorporation of new pattern types such that new distribution schemes can be employed in experiments with with minimal effort. In addition, a system of well-defined properties forms a concise and precise vocabulary to express semantics of data distribution, significantly improving testability of data placement.

We will continue our work on flexible data layout mappings and explore concepts to further support hierarchical locality. We are presently in the process of separating the functional aspects of DASH patterns (partitioning, mapping and layout) into separate policy types to simplify pattern type generators. In addition, the pattern traits framework will be extended by soft constraints to express preferable but non-mandatory properties.

The next steps will be to implement various irregular and sparse distributions that can be easily combined with view specifiers in DASH to support the existing unified sparse matrix storage format provided by SELL-C-σ [10]. We also intend to incorporate hierarchical tiling schemes as proposed in TiDA [14]. A release of DASH including these features will be available in early 2016.

Acknowledgements We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

1. Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.
2. J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. Abstract machine models and proxy architectures for exascale computing. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 25–32, Piscataway, NJ, USA, 2014. IEEE Press.
3. Bradford L Chamberlain, Sung-Eun Choi, Steven J Deitz, David Iten, and Vassily Litvinov. Authoring user-defined domain maps in Chapel. In *CUG 2011*, 2011.
4. Alain Darte, Daniel Chavarria-Miranda, Robert Fowler, and John Mellor-Crummey. Generalized multipartitioning for multi-dimensional arrays. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*, pages 10–pp. IEEE, 2001.
5. Carlos de Blas Cartón, Arturo Gonzalez-Escribano, and Diego R Llanos. Effortless and efficient distributed data-partitioning in linear algebra. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 89–97. IEEE, 2010.
6. H Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore performance-Portability: Kokkos Multidimensional Array Library. *Scientific Programming*, 20(2):89–114, 2012.

7. Karl Fürlinger, Colin Glass, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedeb, and Huan Zhou. DASH: Data structures and algorithms with support for hierarchical locality. In *Euro-Par 2014 Workshops (Porto, Portugal)*, 2014.
8. RD Hornung and JA Keasler. The RAJA Portability Layer: Overview and Status. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.
9. Amir Kamil, Yili Zheng, and Katherine 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*, page 26. ACM, 2014.
10. Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.
11. Manojkumar Krishnan and Jarek Nieplocha. SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 70. IEEE, 2004.
12. Naomi H Naik, Vijay K Naik, and Michel Nicoules. Parallelization of a class of implicit finite difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(01):1–50, 1993.
13. Adrian Tate, Amir Kamil, Anshu Dubey, Armin Größlinger, Brad Chamberlain, Brice Goglin, Carter Edwards, Chris J Newburn, David Padua, Didem Unat, et al. Programming abstractions for data locality. Research report, PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center (CSCS), Lugano, Switzerland, November 2014.
14. Didem Unat, Cy Chan, Weiqun Zhang, John Bell, and John Shalf. Tiling as a durable abstraction for parallelism and data locality. In *Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2013.
15. Robert A Van De Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.