A Multi-Dimensional Distributed Array Abstraction for PGAS

Tobias Fuchs (Author)
MNM Team
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München
tobias.fuchs@nm.ifi.lmu.de

Karl Fürlinger
MNM Team
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München
karl.fuerlinger@nm.ifi.lmu.de

***Abstract*—DASH is a realization of the PGAS (partitioned global address space) model in the form of a C++ template library without the need for a custom PGAS (pre-)compiler.**

We present the DASH NArray concept, a multidimensional array abstraction designed as an underlying container for stencil- and dense numerical applications.

After introducing fundamental programming concepts used in DASH, we explain how these have been extended by multidimensional capabilities in the DASH matrix.

Focusing on matrix-matrix multiplication in a case study, we then discuss an implementation of the SUMMA algorithm for dense matrix multiplication to demonstrate how the DASH NArray simplifies the design of efficient algorithms due to its explicit support for locality-based operations.

Finally, we evaluate the performance of the SUMMA algorithm based on the DASH matrix against established implementations of DGEMM and PDGEMM. In combination with mechanisms for automatic optimization of logical process topology and domain decomposition provided by DASH, our implementation yields highly competitive results without manual tuning, significantly outperforming Intel MKL and PLASMA in node-level use cases as well as ScaLAPACK in highly distributed scenarios.

***Index Terms*—Array; Matrix; PGAS; Data Distribution; DGEMM; PDGEMM**

I. Introduction

Many scientific algorithms are naturally expressed in multi-dimensional domains that arise from discretization of time and space. In the widely used SPMD (single program, multiple data) message-passing model, developers are responsible for manually managing the distribution of these multidimensional domains onto processes as well as their layout in memory. In the best case, manually managing distribution, layout, and transfer of data on such a fine-grained level is an inconvenience. In the worst case, it is the source of errors and prevents code modernization because code and data distribution schemes are tied together too closely.

In the PGAS (partitioned global address space) model data structures are global entities and data distribution is specified once, when instantiating the data structure. All PGAS approaches offer flexible distribution schemes for one-dimensional arrays. However, multidimensional arrays with flexible data distribution schemes are more difficult to realize efficiently since expressing and exploiting data locality is a much bigger challenge.

In this paper we present the multidimensional array concept `NArray` implemented in DASH[1], a realization of PGAS as a C++ template library. It provides distributed data structures and algorithms designed for modern HPC systems which include hierarchical network interconnections and increasingly complex and diversified node-level memory spaces.

In distributed containers, storage space is partitioned in a potentially arbitrary process topology and any element access is subject to physical ownership. Locality measures are thus drastically complicated and maintaining efficiency of seemingly trivial operations becomes a major challenge for programmers.

The PGAS model unifies distributed memory in a virtual global address space, in effect providing a programming interface similar to shared memory. Locality of data access can still be stated explicitly as shared variables have well-defined ownership. PGAS abstractions significantly facilitate the implementation of complex distributed algorithms at least on a syntactic level and can even enable elegant programming techniques, especially when supporting a one-sided communication model.

In this work, we demonstrate how DASH aids in robust, efficient algorithmic design using the example of a multidimensional array abstraction. In summary, this work presents the following contributions:

- We describe a multidimensional distributed array container concept for C++ with strong support for exploiting data locality.
- We develop mechanisms for automatic optimization of logical process topology and domain decomposition.
- We implement an efficient variant of the SUMMA algorithm based on the DASH matrix.
- We provide a comprehensive performance evaluation of the SUMMA implementation against MKL and PLASMA (DGEMM) as well as ScaLAPACK (PDGEMM).

The remainder of this paper is structured as follows: The following section outlines considerations on multidimensional data in PGAS and gives an overview of related work. In Sec. III we explain fundamental concepts defined in DASH. Section IV presents the multidimensional distributed array

[1] http://www.dash-project.org

representation in DASH in depth and illustrates its usage in common scenarios. To demonstrate its benefit in productive applications, we show how the DASH NArray allows for a concise and elegant implementation of the SUMMA algorithm in Sec. V. We then conduct a comprehensive performance evaluation in Sec. VI using ScaLAPACK and Intel MKL with a conclusion in Sec. VII.

II. Related Work

C++ libraries utilizing the PGAS model for parallel programming are subject to extensive research, including but not limited to projects like UPC++, Hierarchically Tiled Arrays [1], STAPL [2], Charm++ [3], and HPX [4].

UPC++ implements a PGAS language model and, similar to the array concept in DASH, offers local views for distributed arrays for rectangular index domains [5]. The UPC++ array library provides a multidimensional array [6], however it currently does not allow the distribution of array elements over multiple processes. This restricts its applicability to purely local computations. Also, UPC++ provides few data distribution schemes, in general restricted to cyclic mappings. In addition, the array abstraction of UPC++ is not compatible with concepts defined in the C++ Standard Template Library (STL). Therefore, existing algorithms designed for C++ standard library containers cannot be applied to UPC++ arrays.

The HPX runtime system realizes a language model in C++ that is comparable to PGAS and also shares many design principles with DASH. It does, however, presently not provide support for multidimensional data.

The Kokkos library [7] provides a multidimensional array abstraction but targets intra-node parallelism focusing on CUDA- and OpenMP backends and thus does not provide methods for process mapping. It is therefore not applicable to the PGAS language model where explicit distinction between local and remote ownership is required.

The fundamental concepts of domain decomposition in DASH are comparable to *DMaps* in Chapel [8], [9] with dense and strided regions like previously defined in ZPL [10]. DASH in addition incorporates a classification system that allows to describe domain decomposition by formal properties.

III. DASH Concepts

We refer to *concept* in this paper as the technical term in the context of C++ to denote a predicate that expresses a set of requirements on a type, including syntactic and semantic conditions [11].

This section explains fundamental concepts in the DASH library used to express process topology, data distribution, and iteration spaces.

The DASH run-time uses MPI-3 RMA operations for one-sided data transfers. A general overview of the DASH library and its run-time backend is presented in [12] and [13], respectively.

A. Process Topology: Teams and Units

The execution model of DASH is SPMD (single program, multiple data) and any logical component in a distributed memory topology that supports processing and storage is called a *unit*. 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 be organized into several node teams, each again consisting of units corresponding to single CPU cores. A unit might represent a processes or thread, but also processing components like accelerators that extent the hardware hierarchy. Conventional MPI-communicators only support arrangements on process level.

B. Sequential Containers

DASH follows the C++ STL convention of using iterators as the primary interface to operate on containers. This compliance is a crucial requirement as it ensures that any algorithm defined the C++ `std` namespace can be applied to any DASH container. The DASH iterator concepts also allow to express locality as required in the PGAS model. For example, DASH iterators can be used to specify multidimensional rectangular domains and provide domain arithmetic with locality-based selection criteria.

The most fundamental container in the DASH library is a distributed one-dimensional array that implements the sequential container concept outlined in Table I. A DASH container is instantiated with the size that declares its initial global capacity, a process topology descriptor of type `TeamSpec`, and an instance of the `Pattern` concept that describes distribution of elements among processes.

```
dash::Array<double> array(1024);
// shared array with 1024 elements, same as
dash::Array<double> array(
   dash::SizeSpec<1>(1024),
   dash::TeamSpec<1>(dash::Team::All()),
   dash::BlockPattern<1>(dash::BLOCKED));
// element range specified using iterators:
auto min = dash::min_element(array.begin() + 100,
                             array.begin() + 300);
```

DASH containers also provide a view on elements local to the calling unit. Iterating over local elements using the `local` qualifier has no overhead compared to accessing a raw native pointer:

```
for (double * p = array.local.begin();
     p != array.local.end(); ++p) { *p = 23.42; }
```

C. Data Distribution: Patterns

The DASH sequential container concept extends the concept of the C++ STL by methods to access the container's distribution pattern which is essential for many PGAS-based algorithms.

The *Pattern* concept defines a mapping between local and global index spaces in two stages, from global index to

TABLE I
METHODS IN THE DASH SEQUENTIAL CONTAINER CONCEPT

Method	Returns	Description
begin()	GlobIter	Iterator at first container element.
end()	GlobIter	Iterator past the final container element.
local	LocalRef	Local view on container, implements the referenced container concept.
[pos]	GlobRef or View	Subscript operator, references element at position pos in the container's iteration space or refines the current view.
pattern()	Pattern	Returns instance of Pattern concept that has been used to distribute container elements.

TABLE II
METHODS IN THE DASH PATTERN CONCEPT

Method	Returns	Description
local(gi)	(Unit,Index)	Map given global index gi to unit and local index.
global(u, li)	Index	Map given unit and local index to global index.
block(bi)	View	Creates a view on block at index bi in block coordinate space.
lblock(lbi)	View	Creates a view on block at index bi in local block coordinate space.
coords(i)	Point	Map given global index to global coordinates.
index(c)	Index	Map given global coordinates to global index.

process, and from process to physical memory. The local position of a container element is thus represented by the unit owning the element and its offset in the unit's memory:

```
dash::Array<double> array(NELEM, my_pattern);
auto   lpos = array.pattern().local(gpos);
int    unit = lpos.unit;
size_t offs = lpos.offset;
long   gpos = array.pattern().global(unit, offs);
```

Essential methods of the Pattern concept are listed in Table II.

The mapping of index spaces in a pattern implicitly specifies a domain decomposition. Pattern types are annotated with traits that describe formal properties of their data distribution scheme. As an example, the *balanced partitioning* property describes that data is partitioned into blocks of identical size and the *balanced mapping* property denotes that the same number of blocks is mapped to every unit. Algorithms can be specialized for pattern properties and use them to specify constraints for compile-time optimization. The underlying mechanisms are beyond the scope of this paper and are described in detail in [14].

IV. MULTIDIMENSIONAL ARRAYS IN DASH

In this section, we explain the matrix concept that defines the general usage of multidimensional arrays in DASH. Most important, we then highlight the expressiveness of view modifiers that allow the specification of multidimensional regions with respect to locality.

A. Instantiation and Data Distribution

Domain decomposition, the distribution of container data among processes, involves two independent concepts in DASH: the *Team Specification* to arrange units in a multidimensional process grid, and the *Pattern* describing data partitioning, process mapping of data domains, and finally data layout in physical memory. To partition a two-dimensional $n \times m$ matrix into blocks of size $bn \times bm$ mapped to a $pn \times pm$ process grid, a possible implementation is:

```
dash::TeamSpec<2> teamspec(pn, pm); // pn x pm process grid
dash::NArray<2, double> // partitioned in tiles of bn x bm:
  matrix(n, m, TILED(bn), TILED(bm), teamspec);
```

The `NArray` class accepts any type satisfying the `Pattern` concept described in Sec. III including user-defined patterns:

```
dash::NArray<2, double, CustomPattern<2>>
  matrix(n, m, CustomPattern<2>(bn, bm), ...);
```

Concept specifications ensure that algorithms operating on NArray are not affected by the applied distribution scheme.

B. The NArray Concept

The first general purpose data structure implemented in the DASH library is the one-dimensional distributed array presented in Sec III. We used the C++ standard template library as a reference for its interface design. However the STL does not specify any concepts for multidimensional containers. The boost library[2] defines the generic N-dimensional array concept *MultiArray* that served as a guideline for the design of the NArray concept in DASH.

A distributed container concept must also provide means to specify domain decomposition In addition, specifying views on data ranges depending on locality measures is mandatory for PGAS programming.

1) Sub-Domain Views: Table III lists the essential methods defined in the NArray concept. As matrix types also satisfy the *Container* concept, they provide iterator-based element access using methods `begin` and `end`. Most methods do not execute an actual operation on its values but configure a light-weight proxy object that acts as a *view* on NArray elements. In general, views realize an intersection of the matrix index set with a multidimensional domain and represent a subset of the elements in a referenced container as though it were a separate container instance. For example, a row slice of a three-dimensional matrix can be treated as if it were an independent two-dimensional matrix. Changes made to view elements will be reflected in the original container.

NArray views are of type `NArrayRef<D,Dv,T>` for elements of type *T* where *D* denotes the total number of dimensions of the referenced NArray and *Dv* the number of dimensions of the view. The concrete view types can be deduced at compile time using the `auto` keyword available since C++11. For

[2]https://boost.org

TABLE III
METHODS IN THE DASH NARRAY CONCEPT

Method	Returns	Description
shape()	size_t[D]	Size of the matrix by dimensions.
extent(d)	size_t	Matrix size in dimension d.
sub(d, {b, e})	MatrixRef	View at matrix slice at range b to e in matrix dimension d.
sub(d, o)	MatrixRef	View at matrix slice at offset o in matrix dimension d.
row(o)	MatrixRef	View at matrix row o, same as sub(0, o).
col(o)	MatrixRef	View at matrix column o, same as sub(1, o).
block(b)	MatrixRef	View at block specified by block coordinates or global block offset b.
[](o)	MatrixRef	View at matrix slice at offset o in the current view dimension.

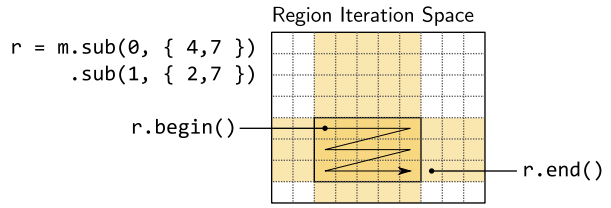

Fig. 1. Chaining view modifiers to select a sub-matrix region

example, the third row of a matrix could be referenced using:

```
auto row = matrix.row(2);
```

`MatrixRef` implements all methods of the matrix concept and can thus again act as a representation of type `Matrix<D,T>`. This allows to compose views via method chaining. As an example, a two-dimensional matrix region can be copied by passing iterators on a view to the function `dash::copy`:

```
auto dom = matrix.sub(0, { 4,7 }).sub(1, { 2,7 });
T * l_copy     = new[region.size()];
T * l_copy_end = dash::copy(dom.begin(), dom.end(),
                            l_copy);
```

As illustrated in Fig. 1, the global iterators returned by `region.begin()` and `region.end()` are relative to the iteration space defined by the region's view.

2) Local Views: Most efficient PGAS-based algorithms facilitate data locality following the *owner-computes* rule such that processes predominantly access values in local memory. All DASH containers provide the member `local` representing a local view that only includes container elements in the calling unit's address space. To restrict operations on local elements, it is sufficient to add the `local` qualifier:

```
dash::NArray<3> m;
for (auto lp = m.local.begin(); l != m.local.end(); ++l)
{  /* iterate local pointers lp */ }
for (auto lv : m.local)
{  /* iterate local values lv   */ }
```

Figure 2 shows a simplified inheritance graph of all matrix types and their relation to the sequential container concept. The illustrated principle also applies to DASH container classes in general. As a local view type fully implements the interface of its referenced container, local views can act as an independent container instances. In the following, we illustrate how view specifiers can be chained to simplify complex operations on multidimensional data ranges.

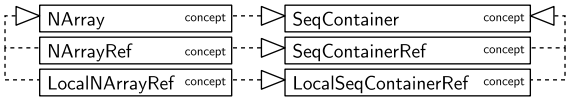

Fig. 2. UML diagram illustrating inheritance of DASH container concepts

3) Block Views: Many common matrix algorithms operate on sub-matrices that correspond to partitions of the matrix domain decomposition. Block regions could be resolved from index calculations on the matrix pattern's blocking factors, however the matrix types provide methods to create views on blocks to simplify these use cases. A matrix block can be referenced by its canonical block offset or a point in the Cartesian block coordinate space. Figure 3 illustrates the semantics of both notations.

4) Combined View Specifiers: The actual expressiveness of views in DASH becomes apparent when combining them in a sequence to specify otherwise complicated index set calculations. The programming interface of views is easy to master as it consists of very few methods with intuitive, consistent semantics.

```
dash::Matrix<2, double> matrix;
// First block in matrix assigned to this unit:
auto l_block    = matrix.local.block(0);
// Map local block back to global index domain, shift
    column:
auto block_west = l_block.global.shift<1>(-1);
// Copy block to second block at this unit:
dash::copy(block_west.begin(), block_west.end(),
           matrix.local.block(1).begin());
// Select matrix block by process grid coordinates:
auto mblock = matrix.block({ 3, 5 });
double * l_submat = new double[mblock.size()];
double * copy_end = dash::copy(mblock.begin(),
                               mblock.end(),
                               l_submat);
```

Listing 1. Combining DASH algorithms and view specifiers

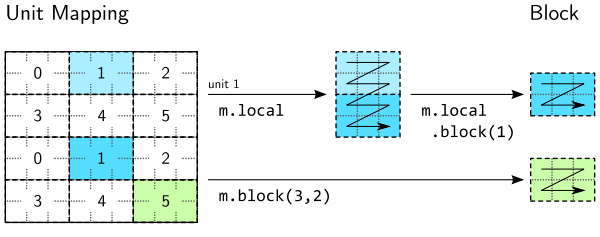

Fig. 3. Example for block views for local and global data domain. On the left, domain decomposition and mapping of blocks to units for 2×3 process grid and block size 2×3. Numbers in blocks represent the unit owning the block.

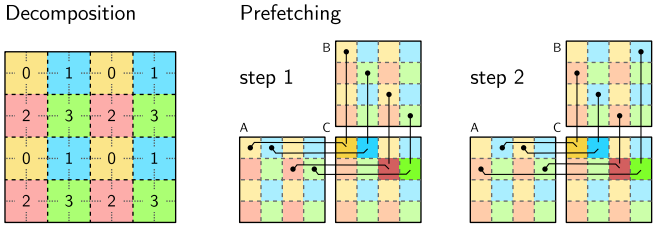

Fig. 4. Illustration of domain decomposition and prefetching in the SUMMA implementation. Numbers in blocks indicate the unit mapped to the block. Sub-matrix blocks of A and B are requested simultaneously in a step as indicated by the lines.

V. Case Study: Matrix-Matrix Product

In the following we describe how the common use case of dense, double-precision matrix-matrix multiplication is implemented in DASH using the matrix concept presented in the previous section. Our implementation is based on a variant of the SUMMA algorithm that facilitates latency hiding using non-blocking, one-sided communication.

A. Implementing SUMMA in DASH

The DASH variant of SUMMA is based on conventional blocked matrix multiplication. For the calculation $C = A \times B$, matrices A, B and C are first distributed using a block-cyclic distribution. Figure 4 illustrates this domain decomposition for a square matrix for simplicity. Our SUMMA algorithm also allows rectangular matrices and unbalanced partitioning.

After this decomposition, a matrix is partitioned into sub-matrices that can be referenced by their respective block coordinates. We use the notation M^g for matrix regions referenced in global memory space and M^l for references in local memory space in the following. Indices in brackets denote block coordinates. For example, the second block mapped to unit 3 in Fig. 4 has coordinates $B^g[1,3]$ and canonical offset $B^g[7]$ in global block coordinate space. Unit 3 can also reference this block in local address space with local block coordinates $B^l[0,1]$ or local canonical block offset $B^l[1]$.

The SUMMA algorithm follows the *owner computes* principle such that every unit computes the multiplication result

$$C^l_{local(i,j)} = A^g_{ik} \times B^g_{kj} = \sum_{k=0}^{K-1} A^g_{ik} B^g_{kj}$$

for all K sub-matrices C^l local to the unit. To achieve overlap of communication and computation, blocks of matrices A and B that are required in the next step are prefetched asynchronously. The sub-matrices A^g_{ik} and B^g_{kj} required for the next computation step are requested using the non-blocking function `dash::copy_async` that returns an instance of `dash::future` and initiates the copy operations in the background. Before starting the next computation phase, calling `wait()` or `get()` on the future instance blocks the unit's execution until the requested data is locally available.

In the implementation of `dash::copy_async`, segments of requested data ranges located on the same processing node are copied in shared memory without calling the communication backend. We also optimize throughput of unavoidable copy operations between processing nodes: when multiple units copy data from the same processing node, interconnect capacity is shared among the units. Some MPI implementations execute concurrent bulk requests to the same MPI process in sequence. We therefore use a communication schedule that balances remote copy operations within a single iteration of the algorithm among all units, conceptually similar to the SRUMMA algorithm [15]. This only required a slight modification to the algorithm: every unit starts with computation of the local sub-matrix $C^l[k_0]$ where k_0 is the unit's row offset in the process grid. Figure 4 illustrates this principle for the first two iterations in the SUMMA algorithm in a simple example: unit 2 and 3, both at row offset 1 in the process grid, each start with computation of their second local block in matrix C. Without this technique, all units would simultaneously request the sub-matrix $A^g[0,0]$ from unit 0 in the first iteration, $A^g[0,1]$ in the second iteration, and so forth.

Algorithm 1 presents the DASH variant of SUMMA in pseudo code. The simplified notation only employs the programming concepts directly supported by DASH as presented in Sec. III. The actual implementation in C++ has identical structure and adds no significant complexity.

Algorithm 1: SUMMA based on the DASH Matrix.

input : Matrix A, Matrix B
output: Matrix $C = A \times B$

```
1  k_0 ← Team.at(myrank).row
2  // prefetch blocks for first computation:
3  C_local ← C.local.block(k_0)
4  A_pref ← A.block(k_0, C_local.col)
5  B_pref ← B.block(C_local.row, k_0)
6  copy(A_pref, bufA_comp)
7  copy(B_pref, bufB_comp)
8  k_max ← C.local.blocks().size()
9  foreach k ← k_0 + 1 to k_0 + k_max do
10     C_local ← C.local.block(k)
11     A_pref ← A.block(k, C_local.col)
12     B_pref ← B.block(C_local.row, k)
13     if k < k_0 + k_max − 1 then
14         // prefetch blocks for next computation:
15         A_fut ← copy_async(A_pref, bufA_get)
16         B_fut ← copy_async(B_pref, bufB_get)
17     end
18     C_local ← C_local + multiply(bufA_comp, bufB_comp)
19     A_fut.wait()
20     B_fut.wait()
21     swap(bufA_get, bufA_comp);
22     swap(bufB_get, bufB_comp);
23 end
24 // wait for completion of all processes:
25 barrier();
```

The local multiplication of sub-matrices, corresponding to line 18 in the pseudo code: $C^l_{local(i,j)} = A^g_{ik} \times B^g_{kj}$

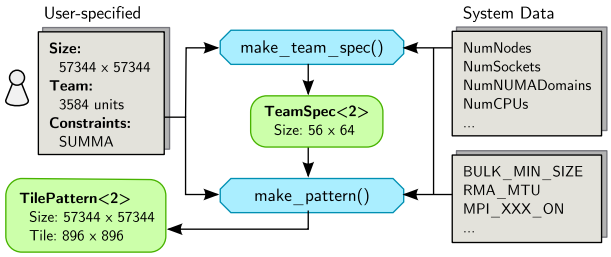

Fig. 5. System diagram of automatic domain decomposition mechanisms in DASH

is performed using serial DGEMM provided by Intel MKL.

B. Automatic Domain Decomposition

If the logical Cartesian arrangement of units is not stated explicitly, DASH derives the logical topology from hardware locality information automatically in a procedure named `make_team_spec` to improve data locality. In this, the process grid is derived from the size of locality scopes such as NUMA domains. As mentioned in Subsec. III-C, algorithms like SUMMA declare constraints on data distribution that can be used to deduce a suitable domain decomposition automatically. The function `make_pattern` deduces a pattern type from given constraints and resolves tiling factors from user-specified parameters and system locality information. Figure 5 outlines the interaction of both mechanisms.

VI. Evaluation

As with other PGAS languages, DASH works on both shared- and distributed memory systems. We consider the scenarios:

Single node: (DGEMM) matrix multiplication on a single processing node for an increasing number of cores
Multi node: (PDGEMM) distributed matrix multiplication for an increasing number of processing nodes

We evaluated the DASH implementation in the single node scenario against multi-threaded Intel MKL and PLASMA [16]. Performance in the multi node scenario is compared against ScaLAPACK [17]. The evaluated performance metric is double-precision floating point operations per second (FLOP/s).

1) DASH: DASH provides automatic optimization of process topology and domain decomposition according to system locality information obtained from e.g. PAPI and hwloc [18]. Different from PLASMA and ScaLAPACK, tile sizes and Cartesian arrangement of units used in all benchmarks of the DASH implementation have been obtained without manual tuning using methods mentioned in Subsec. V-B.

2) PLASMA: PLASMA currently does not detect NUMA specification of the system and does not optimize for node-level memory hierarchy. Performance may therefore be poor if matrices are not distributed among multiple memory nodes. PLASMA allows to specify tile sizes for domain decomposition which significantly affect performance, however no autotuning capabilities are provided at the time of this writing. We resort to pruning methods recommended in the PLASMA Users' Guide[3] to obtain good parameters for tile sizes.

3) Intel MKL: Intel MKL does not optimize for NUMA effects. Tile sizes cannot be specified by the programmer and are therefore not tuned for performance.

4) ScaLAPACK: (Scalable LAPACK) is a library of linear algebra routines for parallel distributed memory machines. Its communication backend for distributed computation is based on two-sided message-passing. We expect that non-blocking RDMA in our implementation significantly reduces synchronization points among units and enables latency hiding by overlapping communication with computation.

A. Benchmark Environment

We evaluate measurements from benchmarks on different systems and for different MPI implementations to substantiate the significance of the presented results. In the following we briefly discuss the benchmark environments which differ in interconnect topology and hardware specifications.

1) SuperMUC: The SuperMUC phase 2 system[4] employs an Infiniband fat tree topology interconnect. Benchmarks have been executed with the Intel MPI and IBM MPI implementations which exhibit characteristic advantages and disadvantages:

- The installation of IBM MPI does not support MPI shared windows, effectively disabling shared memory optimization in the DASH runtime for copying, but offers the most efficient non-blocking RDMA.
- Intel MPI requires additional polling processes for asynchronous RDMA which increases overall communication latency but improved performance of MKL.

2) Cori: Cori phase 1[5] is a Cray system with an Aries dragonfly topology interconnect. An installation of PLASMA is not available.

We repeated all benchmark variations with various runtime configurations and only consider the best results of every evaluated implementation to provide a fair comparison to the best of our abilities.

B. Results

Figures 6 and 7 show measured TFLOP/s in the single-node scenario for Intel MPI and IBM MPI on SuperMUC phase 2. With IBM MPI, DASH cannot exploit MPI shared window optimization. However, even in this pessimistic setup MKL and PLASMA only achieve better performance in singular cases.

As PLASMA does not optimize for NUMA, we tried different configurations of *numactl* as suggested in the official documentation of PLASMA[6] It showed better scalability than Intel MKL, despite the lack of optimization for NUMA in both implementations. However, it is important to note that the ideal tile size for PLASMA had to be obtained in a large series of

[3]http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf
[4]https://www.lrz.de/services/compute/supermuc/systemdescription/
[5]http://www.nersc.gov/users/computational-systems/cori/cori-phase-i/
[6]http://icl.cs.utk.edu/projectsfiles/plasma/html/README.html

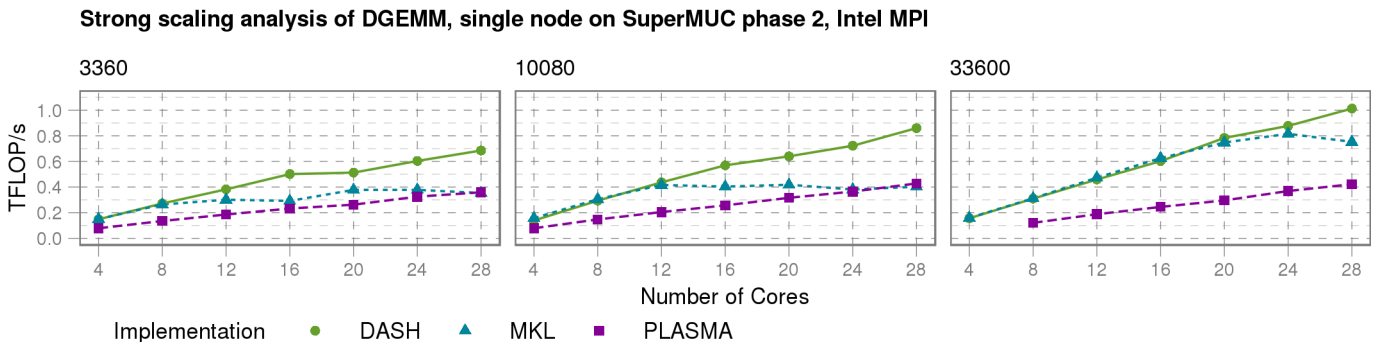

Fig. 6. Strong scaling of matrix multiplication on single node for 4 to 28 cores with increasing matrix size NxN on SuperMUC phase 2, Intel MPI

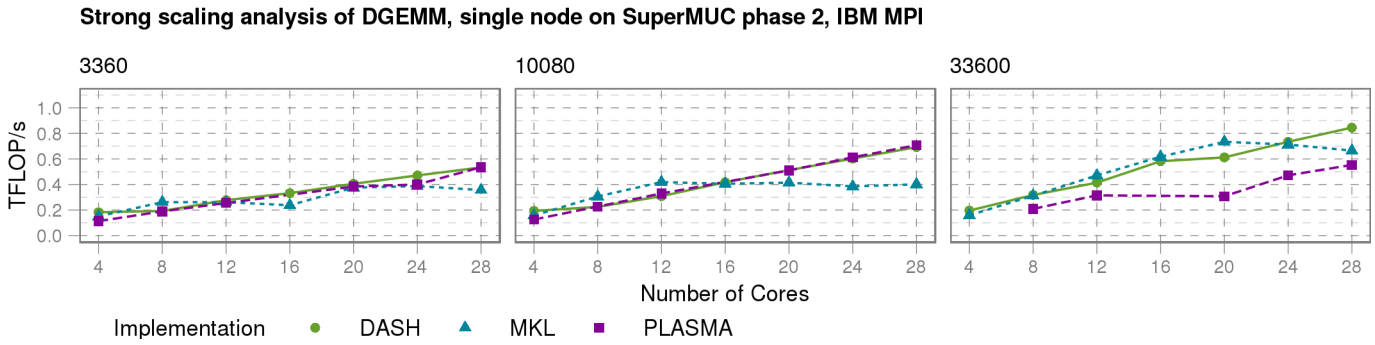

Fig. 7. Strong scaling of matrix multiplication on single node for 4 to 28 cores with increasing matrix size NxN on SuperMUC phase 2, IBM MPI

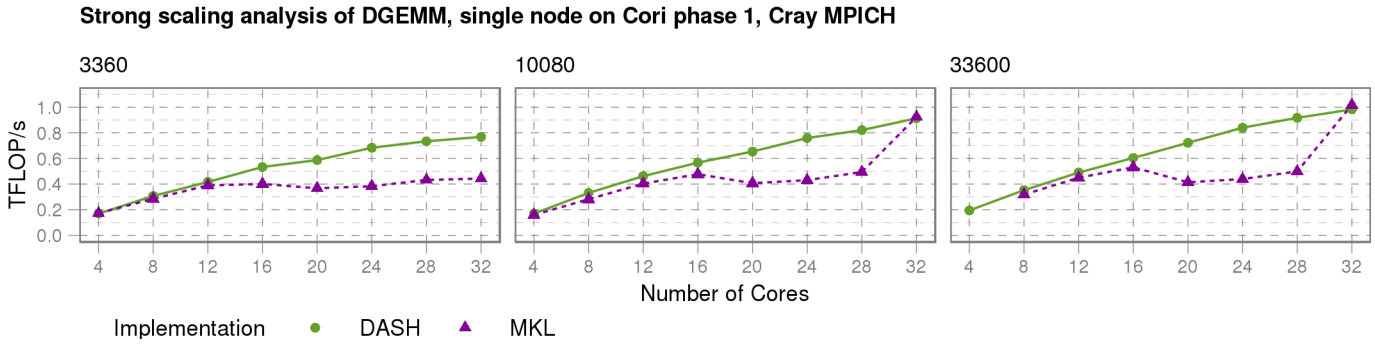

Fig. 8. Strong scaling of matrix multiplication on single node for 4 to 32 cores with increasing matrix size NxN on Cori phase 1, Cray MPICH

tests for every variation of number of cores and matrix size. FLOP/s achieved in the average case were significantly lower than the presented results which only reflect the best cases of all executions.

On Cori phase 1, DASH achieved up to double the performance of Intel MKL when the number of processors was not a power of two.

In the multi-node scenario, our implementation surpassed ScaLAPACK for both Intel MPI and IBM MPI. The less efficient remote memory access in Intel MPI had similar effect on performance of ScaLAPACK and DASH.

In conclusion, the DASH implementation consistently outperformed all tested variants of DGEMM and PDGEMM on distributed- and shared memory scenarios in all configurations of both benchmark environments.

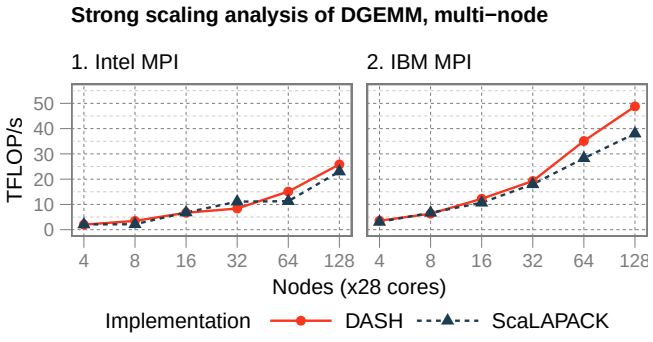

Fig. 9. Strong scaling of dash::summa and PDGEMM (ScaLAPACK) for matrix size 57344 × 57344 on SuperMUC phase 2 for IBM MPI and Intel MPI

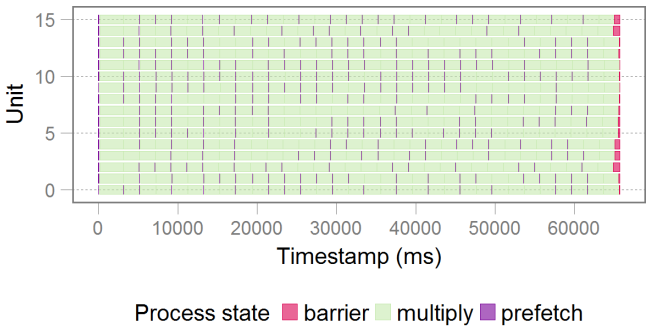

Fig. 10. Process states traced in execution of SUMMA for 16 processes and matrix sizes 2680x2680. Horizontal bars represent time lines of single processes.

To confirm our assumptions on the high degree of latency hiding, we further inspected how computations and communication overlap in the SUMMA implementation and repeated the benchmarks with light-weight event tracing. The analysis of trace data indeed confirmed the expected overlap of communication and computation. An exemplary visualization of program traces obtained in executions of SUMMA is shown in Fig. 10.

VII. Summary and Conclusions

We presented a multidimensional array abstraction and fundamental concepts provided by DASH that allow to specify multidimensional rectangular domains with specific focus on locality. With the SUMMA algorithm as an example, we demonstrated how the DASH Matrix concept simplifies the design of efficient algorithms due to its explicit support for locality-based operations. Our reference implementation proved as highly competitive in performance evaluation compared to the popular solutions Intel MKL, PLASMA, and ScaLAPACK. in node-level as well as highly distributed applications and achieves latency hiding using asynchronous communication.

Re-implementing functions integrated in ScaLAPACK and PLASMA as demonstrated for DGEMM in this work is not a reasonable goal, but we are positive that DASH can help to maintain portable efficiency of existing implementations.

Acknowledgment

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

References

[1] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006, pp. 48–57.

[2] 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.

[3] L. V. Kale and S. Krishnan, *CHARM++: a portable concurrent object oriented system based on C++*. ACM, 1993, vol. 28, no. 10.

[4] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[5] 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.

[6] 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.

[7] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, "Manycore performance-Portability: Kokkos Multidimensional Array Library," *Scientific Programming*, vol. 20, no. 2, pp. 89–114, 2012.

[8] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association, 2010, pp. 12–12.

[9] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov, "Authoring user-defined domain maps in Chapel," in *CUG 2011*, 2011.

[10] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "ZPL: A machine independent programming language for parallel computers," *Software Engineering, IEEE Transactions on*, vol. 26, no. 3, pp. 197–211, 2000.

[11] B. Stroustrup, A. Sutton, L. Voufo, and M. Zalewski, "A concept design for the STL," *ISO/IEC JTC1/SC22/WG21 document N*, vol. 3351, 2012.

[12] 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.

[13] 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.

[14] T. Fuchs and K. Fürlinger, "Expressing and Exploiting Multidimensional Locality in DASH," to appear in Proceedings of the SPPEXA Symposium 2016, Garching, Germany, January 2016.

[15] M. Krishnan and J. Nieplocha, "SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 70.

[16] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[17] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "ScaLAPACK: A portable linear algebra library for distributed memory computers - Design issues and performance," in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*. Springer, 1995, pp. 95–106.

[18] 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 *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 180–186.