A Halo abstraction for distributed n-dimensional structured grids within the C++ PGAS library DASH (#73385)

First submission

Guidance from your Editor

Please submit by **5 Jul 2022** for the benefit of the authors (and your $200 publishing discount) .

Structure and Criteria
Please read the 'Structure and Criteria' page for general guidance.

Raw data check
Review the raw data.

Image check
Check that figures and images have not been inappropriately manipulated.

Privacy reminder: If uploading an annotated PDF, remove identifiable information to remain anonymous.

Files

Download and review all files from the materials page.

17 Figure file(s)
9 Latex file(s)
3 Raw data file(s)

Structure and Criteria

Structure your review

The review form is divided into 5 sections. Please consider these when composing your review:

1. **BASIC REPORTING**
2. **EXPERIMENTAL DESIGN**
3. **VALIDITY OF THE FINDINGS**
4. General comments
5. Confidential notes to the editor

🗋 You can also annotate this PDF and upload it as part of your review

When ready submit online.

Editorial Criteria

Use these criteria points to structure your review. The full detailed editorial criteria is on your guidance page.

BASIC REPORTING

✗ Clear, unambiguous, professional English language used throughout.

✗ Intro & background to show context. Literature well referenced & relevant.

✓ Structure conforms to PeerJ standards, discipline norm, or improved for clarity.

✓ Figures are relevant, high quality, well labelled & described.

✓ Raw data supplied (see PeerJ policy).

EXPERIMENTAL DESIGN

✓ Original primary research within Scope of the journal.

✗ Research question well defined, relevant & meaningful. It is stated how the research fills an identified knowledge gap.

✗ Rigorous investigation performed to a high technical & ethical standard.

✓ Methods described with sufficient detail & information to replicate.

VALIDITY OF THE FINDINGS

ℹ Impact and novelty not assessed. *Meaningful* replication encouraged where rationale & benefit to literature is clearly stated.

✓ All underlying data have been provided; they are robust, statistically sound, & controlled.

✓ Conclusions are well stated, linked to original research question & limited to supporting results.

Standout reviewing tips

The best reviewers use these techniques

Tip	*Example*
Support criticisms with evidence from the text or from other sources	*Smith et al (J of Methodology, 2005, V3, pp 123) have shown that the analysis you use in Lines 241-250 is not the most appropriate for this situation. Please explain why you used this method.*
Give specific suggestions on how to improve the manuscript	*Your introduction needs more detail. I suggest that you improve the description at lines 57- 86 to provide more justification for your study (specifically, you should expand upon the knowledge gap being filled).*
Comment on language and grammar issues	*The English language should be improved to ensure that an international audience can clearly understand your text. Some examples where the language could be improved include lines 23, 77, 121, 128 – the current phrasing makes comprehension difficult. I suggest you have a colleague who is proficient in English and familiar with the subject matter review your manuscript, or contact a professional editing service.*
Organize by importance of the issues, and number your points	*1. Your most important issue* *2. The next most important item* *3. …* *4. The least important points*
Please provide constructive criticism, and avoid personal opinions	*I thank you for providing the raw data, however your supplemental files need more descriptive metadata identifiers to be useful to future readers. Although your results are compelling, the data analysis should be improved in the following ways: AA, BB, CC*
Comment on strengths (as well as weaknesses) of the manuscript	*I commend the authors for their extensive data set, compiled over many years of detailed fieldwork. In addition, the manuscript is clearly written in professional, unambiguous language. If there is a weakness, it is in the statistical analysis (as I have noted above) which should be improved upon before Acceptance.*

A Halo abstraction for distributed n-dimensional structured grids within the C++ PGAS library DASH

Denis Hünich Corresp., 1 , **Andreas Knüpfer** Corresp. 1

[1] ZIH, Technische Universität Dresden, Dresden, Deutschland

Corresponding Authors: Denis Hünich, Andreas Knüpfer
Email address: denis.huenich@tu-dresden.de, andreas.knuepfer@tu-dresden.de

The Partitioned Global Address Space (PGAS) abstraction library DASH provides a C++ based abstraction for distributed N-dimensional structured grids. This paper presents enhancements on top of the DASH library to support stencil operations and halo areas to efficiently parallelize ~~these~~ structured grids. The improvements include definitions of multiple stencil operators, automatic derivation of halo sizes, efficient halo data exchanges, as well as communication hiding optimizations. The main contributions of this paper ~~is~~ are two-fold. First, the halo abstraction concept and the halo wrapper software components. Second, comparisons of the code complexity and the runtime against the prevalent alternative MPI.

Not complete sentences

A Halo Abstraction for Distributed N-Dimensional Structured Grids within the C++ PGAS Library DASH

Denis Hünich[1] **and Andreas Knüpfer**[1]

[1]**Center for Information Services and High Performance Computing (ZIH),** ***Technische Universität Dresden*****, Dresden, Saxony, Germany**

Corresponding author:
Denis Hünich[1]

Email address: denis.huenich@tu-dresden.de

ABSTRACT

The Partitioned Global Address Space (PGAS) abstraction library DASH provides a C++ based abstraction for distributed N-dimensional structured grids. This paper presents enhancements on top of the DASH library to support stencil operations and halo areas to efficiently parallelize these structured grids. The improvements include definitions of multiple stencil operators, automatic derivation of halo sizes, efficient halo data exchanges, as well as communication hiding optimizations. The main contributions of this paper is two-fold. First, the halo abstraction concept and the halo wrapper software components. Second, comparisons of the code complexity and the runtime against the prevalent alternative MPI.

1 INTRODUCTION

Historically, HPC was not been a good tool (Moore's Law)

needs citation if this is true

New trends in parallel and HPC programming

is an existing

Define MPI

High Performance Computing (HPC) has always been a tool for challenging scientific and engineering simulations. It has been dominated by MPI and MPI-style parallelism for a long time. Today, the notion of MPI+X is the generally accepted best practice to reach the highest scalability while efficiently using distributed-memory clusters comprised of multi-core or many-core cluster nodes.

The Partitioned Global Address Space (PGAS) concept is an alternative to the message passing concept. It ~~basically~~ allows random memory access between many processes in a parallel program running across many distributed-memory nodes. Remote access is still much slower than local access, but ~~used carefully~~ can reduce the complexity of distributed-memory parallel programming with little performance penalties. Note that MPI, ~~too~~, adopted PGAS in the form of one-sided communication operations with the MPI 3 standard.

I suggest defining stencil before stencil operation

Structured grids, stencil operations and halo areas

Typically, in regular structured grid simulations every grid element is updated by the same operation; a so called "stencil operation". A stencil defines all grid elements participating in the operation with stencil points pointing to the surrounding grid elements (neighbors). The current grid element (center) can also be a stencil point. Each stencil point has a weight to set the impact on the stencil operation. The arrangement of the neighbors defines the shape of a stencil. Figure 1 shows two examples of two dimensional 9-point stencils. Although both stencils are using the same number of neighbors their shape is different. Figure 1a uses neighbor elements up to ±2 in every dimension, while Figure 1b uses all direct neighbors.

Stencils are often not memory bound, and when they are, matrix-free is often a solution. Parallelism is typically used to speed up the stencil operation.

what are partition elements?

In case the grid is to big to fit into the memory of one compute node, it needs to be divided into partitions across many compute nodes (distributed-memory). This also means that partition elements located on the partition boundaries (boundary elements) can't directly access neighbor elements stored on other partitions ~~anymore~~ and need to request them remotely.

Requesting neighbors element-wise is most inefficient, so "halo areas" (Kjolstad and Snir, 2010) are used instead. Halo areas contain copies of all required neighbor elements located on other partitions.

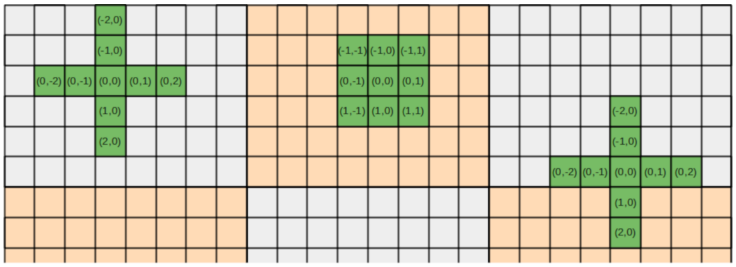


Figure 1. Two shapes of a 9-point stencil: (a) center ±2 stencil in both horizontal and vertical directions, (b) center ±1 stencil point in each direction, and (c) the first stencil crossing the partition boundaries.

every time they are updated? each iteration?

The number and size of halo areas depend on the shape of the applied stencils, the number of neighbor partitions and the size of the partition itself (number of dimensions and distribution pattern). The halo elements are copied from their remote originals from time to time as large memory blocks instead of single elements. Then the halo elements can be accessed locally ~~just like local elements~~.

This procedure significantly increases the performance and eliminates remote requests within stencil operations. Most of the halo update latency can be covered by splitting the partition update (iteration) into two phases. The first phase updates all partition elements with no remote neighbors (inner elements) and in the second phase all remaining partition elements (boundary elements) are updated. The halo updates are started asynchronously before the first phase and all communication transfers are hopefully finished ? within the inner part is updated. Commonly, the number of inner elements is big enough to perfectly cover the halo communication.

2 RELATED WORK

Missing () around citations

Related work regarding C++ parallelization abstractions

We identified three different approaches to combine the PGAS concept with parallel programming, especially for HPC. Not a complete sentence

Could make bulleted list or just bold, e.g. **C++ libraries:** Currently, confusing as incomplete sentences

First, PGAS languages or language extensions. UPC++ Zheng et al. (2014) and Co-Array C++ Eleftheriou et al. (2002) are designed as C++ language extensions, whereas Chapel Chamberlain et al. (2007) and X10 Charles et al. (2005) are separate PGAS programming languages.

Second, libraries with parallel programming APIs. The most dominant one, MPI MPI Forum (2015), also adopted PGAS operations (callign it "one-sided communication"). GASPI Grünewald and Simmendinger (2013) and OpenSHMEM Chapman et al. (2010) are alternative libraries which really follow the PGAS spirit. with similarities to PGAS?

Third, C++ libraries. DASHFürlinger et al. (2014) (section 3), HPX Kaiser et al. (2014), Kokkos Edwards et al. (2014) and STAPL Buss et al. (2010) are libraries that provide communication APIs together with abstractions for distributed data structures. The HPX project addresses distributed memory systems, but doesn't support n-dimensional containers. Kokkos also provides multi-dimensional containers, but focuses on shared memory systems only. STAPL shares concepts like local views on data and representation of distributed containers with DASH, but seems to be a closed source project and doesn't aim classical HPC applications. None of the mentioned PGAS approaches offer stencil and halo abstractions for n-dimensional data containers.

Missing information in related work. The authors list libraries but never discuss them. What is UPC++? Why is it related?

Related work regarding halo exchange mechanisms

The basic concepts of halo areas (also called "ghost cells") and boundary data exchanges are presented in Kjolstad and Snir (2010) and are used in this work and other related approaches. The STELLA project Gysi et al. (2014) provides a domain-specific embedded language using generic programming in C++ and supports stencil codes on structured grids by using OpenMP and CUDA. Compared to the presented approach it is limited to shared memory systems only. DUNE Bastian et al. (2008) and PETSc Balay et al. (1997) are both modular C++ libraries for partial differential equations using grid-based methods and sparse matrix computations. Using MPI, both projects can be used for distributed memory systems. ScaFES Flehmig et al. (2014) also uses MPI to distribute structured grids and to update halo areas.

blue highlights: grammar

Compared to DUNE and PETSc, ScaFES is closest to our approach,but it is limited to MPI two sided communication and isn't maintained anymore.

None of them uses a concept of explicit local and global data accesses such as the DASH data containers.

No existing library

3 THE C++ TEMPLATE LIBRARY DASH

The DASH C++ Template Library (Fürlinger et al., 2014) provides parallel data structures (container) for distributed memory, such as n-dimensional arrays(*NArray*), lists, or unordered maps. Elements in these containers can be accessed by local and global iterators. The iterator concept and other concepts in DASH follow the C++ Standard Library (*SL*) concepts and can be used with algorithms of the *SL* and C++ constructs like range based for-loops. Listing 1 illustrates a local iterator used in the range based for-loop (line 2) and accesses local elements (shared memory) only. The second for-loop (standard for-loop) iterates over all elements with a global iterator (line 7). The global iterator needs to verify the location of every accessed element and triggers a communication request in case of a remote element via the DART library Zhou et al. (2014); a lightweight PGAS runtime. DART provides one-sided put/get communication in a blocking (~~waits for data~~) or asynchronous (~~starts a communication request but doesn't wait for the data~~) mode and local and global synchronization mechanisms. DASH uses the uniform API of DART and is independent of the chosen communication substrate. Currently, DART supports MPI (MPI Forum, 2015) and GASPI (Grünewald and Simmendinger, 2013) based communication substrates.

Difficult to read with so many parentheses

```
1 // local iterator access
2 for(const auto& elem : my_narray.local) {
3         std::cout << elem << "␣";
4 }
5 // global iterator access
6 auto it_end = my_narray.end();
7 for(auto it = my_narray.begin(); it != it_end; ++it) {
8         std::cout << *it << "␣";
9 }
```

Listing 1. Element access via global and local DASH iterator.

To partition the data stored in a container DASH uses pattern (Fuchs and Fürlinger, 2016). A pattern defines how elements are partitioned and arranged based on the number of available processes (named "units" in DASH) and the patterns type.

3.1 DASH NArray and stencil operations

For structured grids with stencil operations the DASH container *NArray* should be used. This container arranges the elements in N dimensions (row or column major) and offers element access via iterators or the subscript operator. A blocked based distribution pattern ensures that each unit allocates one block of contiguous local memory for each process. This memory stores the elements represented by a *n*-dimensional partition of the *NArray*.

To access the elements within a partition the local iterator works fine. But stencil based code need to access, besides the current element (center), neighbors too. This becomes problematic when the neighbor isn't located on the partition. The global iterator can access elements on other partitions, but needs to verify the elements location on every access (locally or remotely) and has to request remotely located elements. This results in ~~a~~ very poor performance. In the end, the programmer still has to organize the update and access of halo elements (1, manage the adjacent partitions and combine the iterator access with the halo access. This is inconvenient, error-prone and requires a lot of code adaption in case the stencil shape changes.

4 HALO ENVIRONMENT FOR THE DASH NARRAY

We developed the *HaloWrapper* abstraction for DASH which hides the complexity of the mentioned halo environment without performance penalties. Goals are (1) easy access of stencil points (center or neighbor elements), (2) hiding the origin of the accessed stencil point (from local partition or from halo),

```
1  using  StencilT        = dash::halo::StencilPoint<2>;
2  using  GlobBoundSpecT  = dash::halo::GlobalBoundarySpec<2>;
3  dash::halo::StencilSpec<StencilT,8>  stencil_spec(
4    StencilT(-1,-1),  StencilT(-1,  0),  StencilT(-1,  1),  StencilT(  0,-1),
5    StencilT(  0,  1),  StencilT(  1,-1),  StencilT(  1,  0),  StencilT(  1,  1));
6  GlobBoundSpecT  bound_spec(BoundaryProp::CYCLIC, BoundaryProp::CYCLIC);
```

Listing 2. Stencil specification for a two dimensional full stencil of width 1. The center element doesn't need to be specified explicitly.

(3) overlap communication for halo exchanges with the stencil computations for the inner region of a partition where no halo elements are required. The existing DASH NArray container and its functionality becomes wrapped by the *HaloWrapper* which provides the extra halo functionality.

In the following we describe essential parts of the *HaloWrapper*, explain its concept, and present performance results.

4.1 *HaloWrapper* specification

The essential parameters of a *HaloWrapper* instance are derived from the shape of the stencils to be applied and from the global grid border preferences.

Stencils can be defined with the *StencilSpec* class, a collection of stencil points (*StencilPoint*) necessary to represent the stencils. Each *StencilPoint* has a weight (coefficient) and coordinates relative to the center. The *StencilSpec* in Listing 2 represents the stencil described in Figure 1(b). The expression StencilPoint<2> defines a two dimensional *StencilPoint*. If no coefficient is passed the constructor expects two coordinates. For example, StencilT(-1,-1) creates a *StencilPoint* pointing to one element before the center for each dimension (north west) and a default weight of 1. specification of

As a convenience, the *HaloWrapper* allows to specify global boundary conditions for the distributed compute grid in the sense of boundary conditions for PDEs, see Listing 2 line 6. They can be passed with the global boundary specification (*GlobalBoundarySpec*) in three ways, separately for each dimension. The default setting `NONE` creates no outside halo areas i.e., no halo area when there is no neighbor partition in that direction. Stencil operations requiring halo access are not executed near such a border. The alternative setting `CYCLIC` logically connects the global boundaries by pointing to the opposite side. The setting `CUSTOM` offers a convenient way to provide arbitrary values for those halo elements, either in a static way (only set initially) or dynamically (updated between the iterations in any way the program sees fit). Such halo regions are not participating in the built-in halo updates.

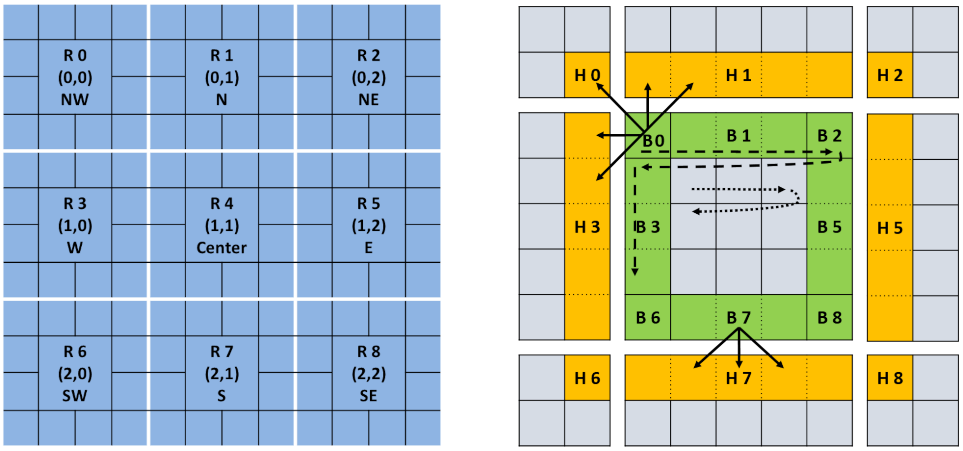
This simple specification is all it needs to instantiate the *HaloWrapper* and to parameterize the following internal components.

4.2 Internal managment of halo partitions

Each distributed partition of an n-dimensional *NArray* is wrapped by a *HaloWrapper* separately. For each partition 3^n *halo and boundary regions* are created, see Figure 2.

Such a region represents an neighbor or the current partition. It has a unique index, unique coordinates, and an extent. Figure 2a shows all possible regions for a 2-dimensional partition with their indices and their coordinates (e.g. (0,1)). The region index follows the row major linearization (last region coordinate grows fastest) and can be easily mapped to the coordinates and vice versa. The width of the region is automatically defined by the maximal distance of all *StencilPoints* from multiple *StencilSpecs* pointing in the same direction.

DASH provides so called *Views* to represent parts of the grid, independent of the elements location. For example, each partition of the grid is represented by a *View*. Special iterators provide access to all elements inside a given *View*. The *HaloWrapper* combines the *View* concept with the aforementioned regions to define all halo areas (*HaloRegions*), boundary areas (*BoundaryRegion*) and the inner elements (*InnerRegion*). Figure 2b shows all *HaloRegions* (yellow) and all *BoundaryRegions* (green) of a partition for a full ±1 stencil on a 2-dimensional grid. The *InnerRegion* includes all elements in the center where no stencil would require elements from any halo. *HaloRegions* are necessary for the halo update process and mark all elements necessary from other partitions. *BoundaryRegions* instead mark local memory parts that need at least one halo element to solve the stencil operation. Using the region concept for the



(a) a

(b) b

Figure 2. Regions concept for a 2-dimensional partition: (a) A center partition and all surrounding neighbor partitions. (b) The regions inside a single partition with halo regions (yellow) and boundary regions (green) and the inner region in the middle.

halo elements is quite obvious, whereas it isn't that obvious for the boundary elements. Figure 2b shows the halo accesses of two stencil operations. While the *BoundaryRegions* on the corner (e.g. B 0) need halo elements of 3 different *HaloRegions*, the *BoundaryRegions* on the edges (e.g. B 7) only need halo elements of one *HaloRegion*. So, the use of *BoundaryRegions* offers the user a method to calculate the boundary elements depending on the already finished update transfers of the necessary *HaloRegions*.

4.3 Internal management of halo memory and halo data exchanges

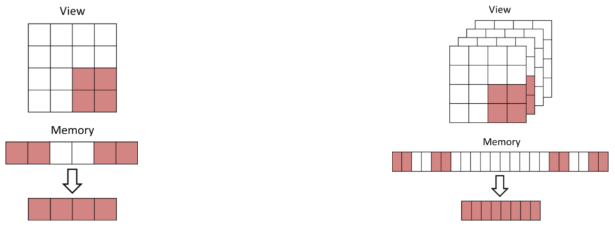
Halo elements marked by the *HaloRegions* are stored in one separate contiguous memory block (*HaloMemory*). A virtual layer separates this *HaloMemory* into *HaloRegions* to support region-wise halo element access. Most of the requested halo elements inside a selected *HaloRegion* can't be transferred with on communication request, because they are not stored contiguous. In Figure 3 we show two corner *HaloRegions* for 2- and 3-dimensional partitions. To transfer the marked elements for the 2-dimensional case (Figure 3a) two communications requests are necessary. In case of three dimensions (Figure 3b) already four communication requests are necessary. The number of requests for corner elements is negligible, compared to for example a one element thin layer of a three dimensional partition, where each halo element needs a communication request. The DART interface support strided communication requests for such cases. In this case the used communication substrate decides how the requested halo elements are transferred. While MPI can handle a big number of communication requests, GASPI doesn't work well for many communication requests. So, to support both communication substrates and to efficiently update the requested halo elements we decided to buffer them in a contiguous memory block, to be transferred with one communication request later on. It requires more memory but results in lower waiting latencies(finalizing all created communication requests).

normally you would pack and then communicate once (or use MPI datatypes)

this is the standard approach

The *HaloWrapper* supports two types of halo data exchanges; blocking and asynchronous. Both need to be called ~~as~~ collectively i.e., jointly by all participating processes. The blocking halo update doesn't return until all ~~started~~ halo updates are finished and doesn't need additional synchronization methods such as barriers. The asynchronous counterpart initiates the halo updates and returns immediately (Listing 3 line 9). A wait function ensures that all or a subset of the started updates are locally finished and all halo elements can be used ~~now~~ (Listing 3 line 17).

To guarantee the correct values of the halo elements all partitions have to be synchronized. Normally, this is achieved by a barrier at the end of an iteration and forces the user to explicitly synchronize its application and wait for the slowest partition. We implemented a signaling environment to synchronize the partitions and their neighbors. To avoid additional function calls, we implemented the signal handling inside the halo update and waiting calls (Listing 3 line 9 and 17). The asynchronous halo update consists of 5 parts: (1) check if the neighbor partitions finished the previous update, (2) buffer all halo elements



(a) a **(b)** b

Figure 3. Halo data exchange: Remote strided data to contiguous memory: (a) Corner halo Region only have one fixed offset between data chunks (b) Corner halo Region already have two different offset between data chunks for three dimensions

requested by neighbor partitions, (3) send signals that all halo elements are ready for transfer, (4) wait until own requested halo elements are ready for transfer, (5) start asynchronous halo transfer. The wait call: (6) blocks until all own requested halo elements are transferred and (7) sends signals to neighbor partitions that the requested halo elements are received. The signals are onesided communication calls to write a defined value into the memory of the selected partitions ((3) and (7)). These partitions poll for the defined value and reset it ((1) and (4)) after the transfer is finished. The signals used in (1) and (7) can be turned on or off with a template parameter. They are necessary for small grids, when a partition already modifies the buffered halo elements, while another partition still uses the buffer.

4.4 Stencil Point Access

The primary concern is to apply the previously defined stencils to every element in the distributed compute grid. All other types of access can still be done through the underlying *DASH::NArray* and its iterators. For convenience we provide an iterator class to apply a given stencil to all grid points of the partition. ~~Yet,~~ this needs to check the location for every stencil point, whether it is inside the local partition or the halo memory, which slows down the stencil operation. For optimal performance we implemented separate iterators for the inner region and the boundary regions of the partition, see 2b. We recommend to apply the stencil in two separate loops, where both loop bodies can be identical, as shown in Listing 3. This conforms to the basic halo concept, see Section 1.

also standard

There are four steps to by executed collectively for all distributed partitions. First, the asynchronous halo updates need to be initiated (Listing 3 line 10). Second, the inner iteration is to be executed (Listing 3 from line 12). Third, the halo updates must be finished (Listing 3 line 23). Most likely, this has happened already during the second step, if the inner region is large enough. Then there is perfect overlap between data transfers and the inner calculation. Finally, the boundary iteration is executed using the updated halo elements. The boundary areas tend to be small compared to the inner area.

MPI typically doesn't have strong progress

cite if true

A *StencilIterator* accesses the center elements equivalent to C++ Standard Library iterators with the address-of operator. Neighbor elements are accessed by member function, which automatically resolves the origin of the accessed element (halo or partition element) and then returns the requested element. The *InnerIterator* is designed to accesses elements without halo neighbors only and doesn't need to validate the locations of the neighbors. The iteration order follows the memory order defined for the using NArray (row or column major). Figure 2b shows the iteration order for row major grids (dotted arrows). Because the iterator needs to evaluate its position inside the partition each time its moved, we ~~especially~~ optimized the mostly used pre-increment operator. Because the neighbors depend on the center, their memory position be easily calculated (fixed offset). The *BoundaryIterator* instead needs to iterate over more than one region and cover all identified *BoundaryRegions*. Inside a *BoundaryRegion* the iteration order is equivalent to the *InnerRegion*. In case the *BoundaryIterator* reaches the end of a *BoundaryRegion* it points to the beginning of the next one, containing at least one element. The order in which the *BoundaryRegions* are traversed is from the smallest to the largest region id. An example of over four *BoundaryRegions* is shown Figure 2b with the dashed arrows, from *BoundaryRegion* 0 until the end of 3.

The *HaloWrapper* is designed to match all *StencilSpecs* passed by the user. But, if the *StencilSpecs* differ in distance or direction, the created *BoundaryRegions* and *InnerRegions* might not fit anymore. For this reason the *HaloWrapper* provides an infrastructure specific for one *StencilSpec* called (*StencilOper-*

```
1  StencilSpecT s_spec( StencilT(-1, 0), StencilT(1, 0),
2  StencilT( 0,-1), StencilT(0, 1));
3  // Periodic/cyclic global border conditions for both dimensions
4  GlobBoundSpecT bound_spec(BoundaryProp::CYCLIC, BoundaryProp::CYCLIC);
5  HaloWrapperT halo_wrapper(src_matrix, bound_spec, s_spec, s_spec_2, ...);
6  // Stencil specific operator for a specific stencil_spec
7  auto stencil_op = halo_wrapper.stencil_operator(s_spec);
8
9  for (auto i = 0; i < iterations; ++i) {
10   halo_wrapper.update_async(); // start asynchronous halo update
11
12   // Calculation of all inner elements (InnerRegion)
13   auto iend = stencil_op.inner.end();
14   for(auto it = stencil_op.inner.begin(); it != iend; ++it) {
15     // it.value_at(0) accesses neighbor represented by StencilT(-1, 0)
16     auto center = *it;
17     double dtheta =
18     (it.value_at(0) + it.value_at(1) - 2 * center) / (dx*dx) +
19     (it.value_at(2) + it.value_at(3) - 2 * center) / (dy*dy);
20     ...
21   }
22
23   halo_wrapper.wait(); // Wait until all halo elements are updated
24
25   // Calculation of all boundary region elements
26   auto bend = stencil_op.boundary.end();
27   for(auto it = stencil_op.boundary.begin(); it != bend; ++it) {
28     //same as the inner part
29   }
30 }
```

Listing 3. 2D iteration loop with halo exchange and stencil operations

ator). Each *StencilOperator* provides the aforementioned *StencilIterators*, methods to simplify stencil operation tasks and management parts of halo environment (see Listing 3 line 7).

To support an alternative access besides iterators we also implemented a coordinate based access (e.g. element_access[-1][0]). This kind of access can also be used for the inner and boundary regions, but the user needs to care about valid coordinates. The iterator based access strategy is recommended, because it can be used with the C++ Standard Library algorithms.

5 EVALUATION

To evaluate the DASH *HaloWrapper* we implemented the heat equation for structured square grids with a Jacobi solver and `CYCLIC` boundary conditions for 2 and 3 dimensions (2D and 3D) in MPI and DASH. The Jacobi solver requires two equivalent compute grids, that act as a read-only source grid and a write-only destination grid, alternating between the iterations. A 5-point-stencil (2D) and a 7-point-stencil (3D) with an offset of ±1 in every dimension were used on a double precision grid elements.

Each iteration is divided into the asynchronous update of the halo elements (*async*), the calculation of all inner elements (*inner*), waiting for the halo transfers to finish (*wait*), and the calculation of all boundary elements (*bound*). The *inner* part should overlap the halo update process started in *async*, and thus minimize *wait*. The number of iterations is fixed to 100 (*calc total*) to compare the implementations with MPI (mpi) and the DASH *HaloWrapper* (dash).

5.1 Source Code Comparison

One aspect of comparison is the code complexity. This is connected with the length and maintainability of the code.

While LOC comparisons are always subjective, because it can be argued, that the certain parts can be

Table 1. Lines of Code (LOC) comparison of MPI and DASH for a 2D and 3D heat eqation simulation with breakdown into the code phases, without code comments and blank code lines. The *2D to 3D* rows show the number of LOC that had to be adapted and in brackets the number of LOC that had to be added. The latter is a subset of the changed LOC.

Implementation	Total	Async. Halo Exchange	Inner Calc.	Boundary Calc.
		(*async* and *wait*)	(*inner*)	(*bound*)
MPI 2D	249	20	8	18
MPI 3D	318	28	11	36
MPI 2D to 3D	132 (69)	18 (8)	6 (3)	30 (14)
DASH 2D	134	2	7	7
DASH 3D	140	2	8	8
DASH 2D to 3D	20 (6)	0 (0)	2 (1)	2 (1)

hidden in extra functions, to be called with one LOC later. The logic of these functions also need to be adapted, in case the environment (e.g. number of dimensions) changes. In our case, we use the LOC to demonstrate the effort a user needs to adapt the application, if the number of dimensions changes.

Table 1 shows the lines of code (LOC) for the four implementations and their code phases. Even though not a strict criterion for complexity, the fewer or much fewer LOC for the DASH variants indicate lower code complexity and opportunities for inconsistencies. Another strong indication for less complexity and better maintainability is the increase in LOC when transforming the code from 2D to 3D. DASH requires only 6 additional LOC and additionally modify 14 LOC, when switching from 2D to 3D. Whereas, MPI needs to add 69 LOC and modify 132 LOC in total. Note, that the two lines for the asynchronous DASH halo exchange needs no adaptation at all.

Sounds like DASH provides opportunities for inconsistencies

Use comparable numbers (e.g. 20 : 132)

Listing 4 shows the 3D iteration loop which is the counterpart to the 2D case in Listing 3. All code examples are provided in the github repository[1].

5.2 Performance Comparison

In HPC the reduction of code complexity is appropriate only, if it doesn't decrease the performance. Therefore, we compared both implementations (plain MPI and DASH) in a weak and a strong scaling scenario. The runtimes were measured on the Bull HPC-Cluster "Taurus" at ZIH, TU Dresden on the *Haswell* and the *Romeo* partition. *Haswell* provides compute nodes with two Haswell E5-2680 v3 CPUs at 2.50GHz (12 physical cores each) and 64 GB memory. Compute nodes on *Romeo* have two AMD EPYC CPU 7702 (64 physical cores each) and 512 GB memory. Hyper-Threading (*Haswell*) and Simultaneous Multithreading (*Romeo*) were disabled. The implementations were compiled with gcc 10.2.0 and used OpenMPI 4.0.5.

Still useful if simpler to program. Most useful if performance remains

The essential runtime contributions are *async*, *inner*, *wait*, *bound* and the total caluclation time (*calc*). Each combination was measured three times and the plots show the mean of these measurements. While *calc total* is measured once per run, the other parts were recorded for each iteration and were summed up in the plots. The weak scaling scenario increases the number of grid elements proportional to the number of compute nodes. Nearly the total memory of each compute node were used. The figures marked with (a) show the runtimes of the phases *calc*, *inner* and *bound*, while figures marked with (b) show the runtimes of the phases *async* and *wait*. The values plotted in the figures are not stacked.

Figure 4 and 5 show the results of the 2D heat equation on *Haswell* and *Romeo*. Both implementations show nearly identical runtimes (Figure 4a) on *Haswell*, while on *Romeo* dash performs slightly better (Figure 5a). Interestingly, the runtime of the components is quite differently. dash is faster calculating the inner elements, but slower in *async* and vice versa for mpi. mpi performs better with many communication requests per neighbor, because the wait stage can process the requests faster, as dash can prepare the halo data for the one communication request per neighbor strategy (see 4.3). In total dash is slightly faster in total.

The results of the 3D pendant are shown in Figure 6 and 7. In total the runtimes of dash and mpi are very similar, again. On *Haswell* dash is slower for one and two nodes, but faster for 4 to 64 nodes. On *Romeo* dash and mpi are closer together with slight advantages here and there. However, the contributions

similar?

[1] `https://github.com/dash-project/dash-apps/heat_eqation`

of the components change entirely. Now, the wait phases in the mpi case is much longer than the dash `async` component. But dash needs more time for the *inner* and *bound* components (Figure 6b and 6a).

Especially on *Haswell* the time used to update the halo elements is a big advantage of dash; at least by a factor of three (Figure 6b). A similar behavior can be seen on *Romeo*, but not as distinct (Figure 7b and 7a). The reason for the changed mpi behavior is the much higher number of individual requests due to strided accesses. dash still uses one request with buffered halo data per neighbor. The buffering approach is superior for higher dimensions and large locals grids.

In both scenarios, 2D and 3D, dash spent no significant time in *wait*, just as mpi in *async*. Also, the amount of time spent for *bound* is very small compared to *inner*. To see these stages behave in a strong scaling scenario we used 55000^2 grid elements for 2D and 1500^3 grid elements for 3D (fits into the main memory of one compute node). The measurements were made on *Haswell* to involve more compute nodes by still using all cores on these nodes.

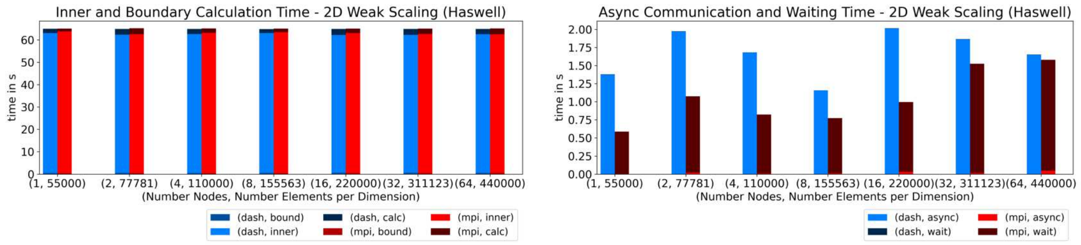
Figures 8 and 9 show the results for the strong scaling scenario (2D and 3D). The behavior is quite similar in both scenarios. Until 16 cores on one node dash outperforms mpi, but on top of that, dash and mpi are close together (Figure 8a and 9a). The results for 16 and 32 compute nodes for 3D are an exception where dash is faster than mpi. Notably, both exhibit the same performance artifact around 12 to 24 cores. The reason is an increased number of last level cache load misses and an indicator that the implementations are memory bound at this number of cores per node.

Interesting, besides the total results, are the individual stages. The *inner* stage is processed faster by dash in both scenarios, while *bound* is slower compared to mpi (Figure 8a and 9a). The *HaloWrapper* has a slight disadvantage, by using separate memory for the halo elements. While mpi uses one memory chunk for grid and halo elements, which can be good optimized. In Figure 8b dash needs significantly

unclear what you are trying to say

```
StencilSpecT s_spec( StencilT(-1,  0,  0), StencilT(1, 0, 0),
                     StencilT( 0, -1,  0), StencilT(0, 1, 0)
                     StencilT( 0,  0, -1), StencilT(0, 0, 1));
// Periodic/cyclic global border conditions for both dimensions
GlobBoundSpecT bound_spec(BoundaryProp::CYCLIC, BoundaryProp::CYCLIC,
                          BoundaryProp::CYCLIC);
HaloWrapperT halo_wrapper(src_matrix, bound_spec, s_spec, s_spec_2, ...);
// Stencil specific operator for a specific stencil_spec
auto stencil_op = halo_wrapper.stencil_operator(s_spec);

for (auto i = 0; i < iterations; ++i) {
  halo_wrapper.update_async(); // start asynchronous halo update

  // Calculation of all inner elements (InnerRegion)
  auto iend = stencil_op.inner.end();
  for(auto it = stencil_op.inner.begin(); it != iend; ++it) {
    // it.value_at(0) accesses neighbor represented by StencilT(-1, 0)
    double dtheta =
      (it.value_at(0) + it.value_at(1) - 2 * center) / (dx*dx) +
      (it.value_at(2) + it.value_at(3) - 2 * center) / (dy*dy) +
      (it.value_at(4) + it.value_at(5) - 2 * center) / (dz*dz);
    ...
  }

  halo_wrapper.wait(); // Wait until all halo elements are updated

  // Calculation of all boundary region elements
  auto bend = stencil_op.boundary.end();
  for(auto it = stencil_op.boundary.begin(); it != bend; ++it) {
    //same as the inner part
  }
}
```

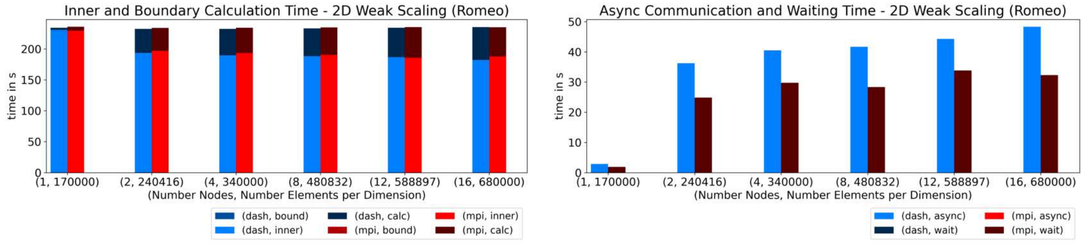
Listing 4. 3D iteration loop with halo exchange and stencil operations



(a)

(b)

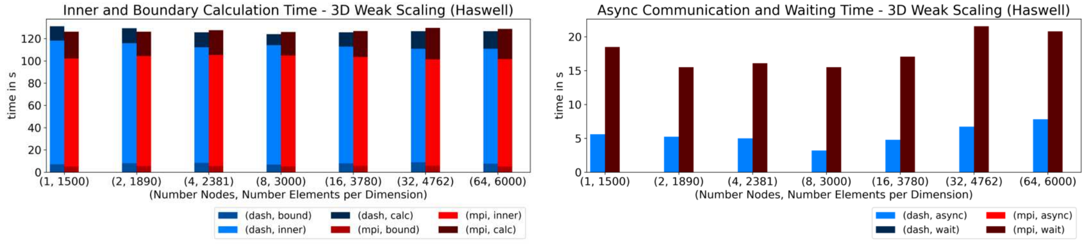
Figure 4. Weak Scaling for 2D Heat Equation - DASH vs. MPI on Haswell (values not stacked)



(a)

(b)

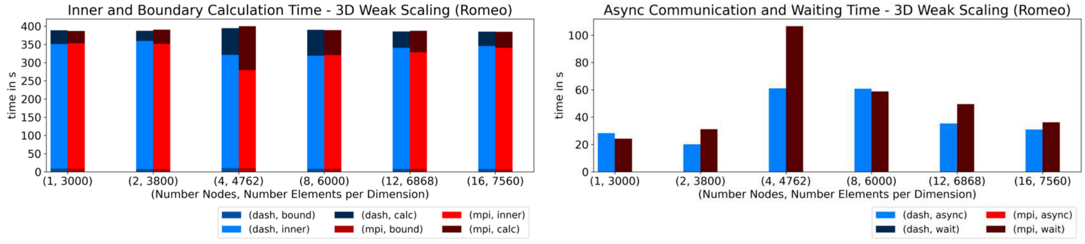
Figure 5. Weak Scaling for 2D Heat Equation - DASH vs. MPI on Romeo (values not stacked)



(a)

(b)

Figure 6. Weak Scaling for 3D Heat Equation - DASH vs. MPI on Haswell (values not stacked)



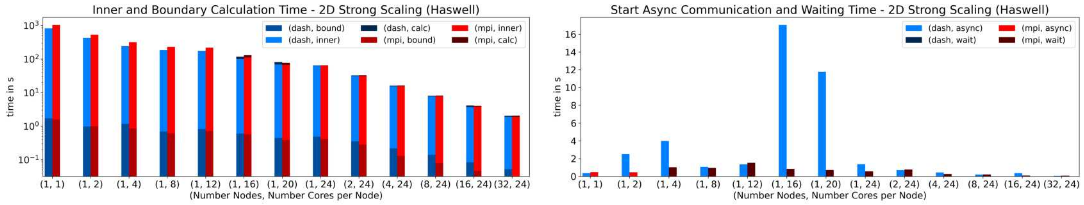
(a)

(b)

Figure 7. Weak Scaling for 3D Heat Equation - DASH vs. MPI on Romeo (values not stacked)

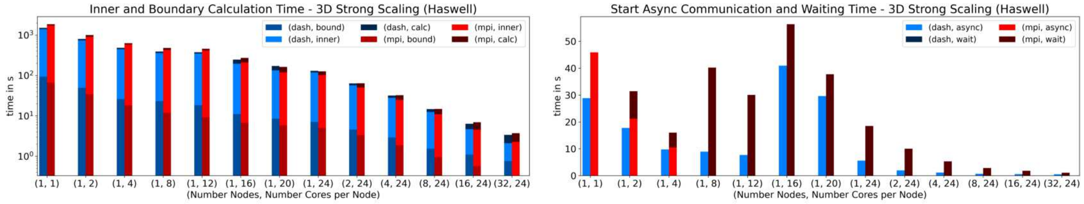
longer in *async* (16 and 20 cores) and is based on the preparation of the halo elements. As mentioned before, the implementations suffer performance by a lot of L3 cache misses, which in this case also influence the *async* stage in dash. While dash still has no significant runtime in *wait*, mpi now spends time in *async* until 4 cores in the 3D scenario.

Overall, dash can compete with the plain mpi example for all presented scenarios;by less LOC. The *HaloWrapper* can also be easily adapted to higher number of dimensions, other stencil shapes or boundary conditions.



(a) (b)

Figure 8. Strong Scaling for 2D Heat Equation - DASH vs. MPI on Haswell (values not stacked)



(a) (b)

Figure 9. Strong Scaling for 3D Heat Equation - DASH vs. MPI on Haswell (values not stacked)

6 CONCLUSION AND FUTURE WORK

In this paper we presented an abstraction for stencil codes and halo areas for distributed n-dimensional structured grids and the data container classes for this purpose provided by the DASH C++ template library. This greatly simplifies programming and code maintenance of parallel computations on distributed compute grids because stencils, halo areas, and boundaries only need to be specified and parameterized instead of implemented from scratch. Changes of stencils or even dimensionality require ~~only~~ few code changes.

The DASH halo concept is compared to the typical MPI counterpart. First, in terms of code complexity which is estimated based on lines of code and code changes needed. This shows a clear advantage for the presented solution And second in terms of runtime of the data transfer and synchronization steps. Here, the DASH and MPI solutions are on a par with slight advantages for the DASH case on average.

Thus, the advantages for programming and code maintenance comes at no performance costs. Furter optimizations like tasking, multiple threads, or tuning of the stencil operation are perfectly possible on top of the presented approach and are ~~our~~ primary goals for future work.

The source codes used here are available on github[2] to allow studying the shown code excerpts in their context and for reproducibility of our results.

REFERENCES

Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. (1997). *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, pages 163–202. Birkhäuser Boston, Boston, MA.

Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Ohlberger, M., and Sander, O. (2008). A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework. *Computing*, 82(2):103–119.

Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N. M., and Rauchwerger, L. (2010). Stapl: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA. ACM.

Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312.

[2] `https://github.com/dash-project/dash-apps/tree/master/heat_equation`

Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., and Smith, L. (2010). Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proc. of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA. ACM.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538.

Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Eleftheriou, M., Chatterjee, S., and Moreira, J. E. (2002). A c++ implementation of the co-array programming model for blue gene/l. In *Parallel and Distributed Processing Symposium, International*, volume 3, page 0105, Los Alamitos, CA, USA. IEEE Computer Society.

Flehmig, M., Feldhoff, K., and Markwardt, U. (2014). Scafes: An open-source framework for explicit solvers combining high-scalability with user-friendliness. In *ARCS 2014; 2014 Workshop Proc. on Architecture of Computing Systems*, pages 1–8.

Fuchs, T. and Fürlinger, K. (2016). Expressing and exploiting multidimensional locality in DASH. In Bungartz, H.-J., Neumann, P., and Nagel, E. W., editors, *Software for Exascale Computing - SPPEXA 2013-2015*, pages 341–359, Garching, Germany. Springer.

Fürlinger, K., Glass, C., Gracia, J., Knüpfer, A., Tao, J., Hünich, D., Idrees, K., Maiterth, M., Mhedheb, Y., and Zhou, H. (2014). DASH: Data Structures and Algorithms with Support for Hierarchical Locality. In Lopes, L. e., editor, *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *LNCS*, pages 542–552. Springer International Publishing.

Grünewald, D. and Simmendinger, C. (2013). The GASPI API specification and its implementation GPI 2.0. In *7th International Conference on PGAS Programming Models*, volume 243.

Gysi, T., Fuhrer, O., Osuna, C., Cumming, B., and Schulthess, T. (2014). STELLA: A domain-specific embedded language for stencil codes on structured grids. In *EGU General Assembly Conference Abstracts*, volume 16 of *EGU General Assembly Conference Abstracts*, page 8464.

Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., and Fey, D. (2014). HPX: A Task Based Programming Model in a Global Address Space. In *Proc. of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA. ACM.

Kjolstad, F. B. and Snir, M. (2010). Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 4:1–4:9, New York, NY, USA. ACM.

Matthes, A., Widera, R., Zenker, E., Worpitz, B., Huebl, A., and Bussmann, M. (2017). Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library.

MPI Forum (2015). MPI: A Message-Passing Interface Standard. Version 3.1. available at: `mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

Zheng, Y., Kamil, A., Driscoll, M. B., Shan, H., and Yelick, K. A. (2014). Upc++: A pgas extension for c++. *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114.

Zhou, H., Mhedheb, Y., Idrees, K., Glass, C. W., Gracia, J., and Fürlinger, K. (2014). DART-MPI: An MPI-based Implementation of a PGAS Runtime System. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 3:1–3:11, New York, NY, USA. ACM.