Authoring User-Defined Domain Maps in Chapel*

Bradford L. Chamberlain Sung-Eun Choi Steven J. Deitz David Iten Vassily Litvinov
Cray Inc.
chapel_info@cray.com

Abstract

One of the most promising features of the Chapel parallel programming language from Cray Inc. is its support for *user-defined domain maps*, which give advanced users control over how arrays are implemented. In this way, parallel programming experts can focus on details such as how array data and loop iterations are mapped to a target architecture's nodes, while end-users can benefit from their efforts when using Chapel's high-level global array operations. Chapel's domain maps also support control over finer-grained decisions like what memory layout to use when storing an array's elements on each node. In this paper, we provide an introduction to Chapel's user-defined domain maps and summarize the framework for specifying them.

1 Introduction

Chapel [13, 7] is a parallel programming language being developed by Cray Inc. with the goal of improving programmer productivity compared to conventional programming notations for HPC, like MPI, OpenMP, UPC, and CAF. Chapel's goal is to greatly improve upon the degree of programmability and generality provided by current technologies, while supporting performance and portability that is similar or better. Chapel has a portable implementation that is available under the BSD license and is being developed as an open-source project at SourceForge[1].

One of Chapel's most attractive features for improving productivity is its support for *global-view arrays*, which permit programmers to apply natural operations to an array even though its implementation may potentially span multiple distributed memory nodes. Unlike previous languages that have supported global-view arrays (*e.g.*, HPF, ZPL, UPC), Chapel allows advanced users to author their own parallel array implementations. This permits them to specify the array's distribution across nodes, its layout within a node's memory, its parallelization strategy, and other important details.

In Chapel, these user-defined array implementations are known as *domain maps* because they map a *domain*—Chapel's representation of an array's index set—down to the target machine. Domain maps that target a single shared memory segment are known as *layouts* while those that target multiple distinct memory segments are referred to as *distributions*. All of Chapel's domain maps are written within Chapel itself. As a result, they can be implemented using Chapel's control-oriented productivity features, including task parallelism, locality control, iterator functions, type inference, object-orientation, and generic programming.

In previous work, we introduced Chapel's philosophy for user-defined distributions and provided a very high-level view of the software framework used to specify them [10]. In this paper, we describe the framework at the next level of detail in order to provide a better picture of what is involved in authoring a domain map. For even more detail, the reader is referred to the documentation for domain maps within the Chapel release itself.

This paper is organized as follows: The next section provides a brief summary of our goals for supporting user-defined domain maps. Following that, we provide a summary of related work, contrasting our approach with previous efforts to support global-view arrays and user-defined distributions. Section 4 provides a brief introduction of Chapel to support and motivate the domain map framework. The detailed description of our framework for defining user-defined domain maps is given in Section 5. Section 6 describes our implementation status while Section 7 summarizes and lists future work.

2 Goals of This Work

The overarching goal of this work is for the Chapel language to support a very general and powerful set of distributed global-view array types, while also providing advanced users with the ability to implement those arrays in whatever ways they see fit. We adopt a language-based solution both for the improved syntax it supports and to expose optimization opportunities to the compiler.

*This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001.

[1] http://sourceforge.net/projects/chapel/

A well-written domain map should support a plug-and-play quality in which one distribution or layout can be substituted for another in a line or two of declaration code. Meanwhile, any global-view operations on the arrays themselves can remain unchanged, thereby insulating the specification of the parallel algorithm from its implementation details. As a simple motivating example, consider the following STREAM Triad kernel, which computes a scaled vector addition:

```
const D = [1..n];       // the problem space

var A, B, C: [D] real; // the three vectors

A = B + alpha * C;      // the computation
```

As written here, the domain `D` representing the problem space is declared without any domain map specification. As a result, it is implemented using Chapel's default layout. This causes arrays `A`, `B`, and `C` to be allocated using memory local to the current node. Moreover, the parallel computation itself will be implemented using the processor cores of that node. Thus, we have written a Chapel program suitable for desktop multicore programming.

To target a large-scale distributed-memory system, we can simply change the domain's value to include a distribution:

```
const D = [1..n] dmapped Cyclic(startIdx=1);

var A, B, C: [D] real; // the three vectors

A = B + alpha * C;      // the computation
```

Here, we specify that the domain's indices should be mapped to the target architecture (`dmapped`) using Chapel's standard `Cyclic` distribution. This will deal its indices out to the nodes on which the program is executing in a round-robin manner, starting from the specified index, 1. The array declaration and computation lines need not change since they are independent of the implementation details. Finally, we can switch to another distribution simply by changing the domain map. For example, a `Block` distribution could be specified as follows:

```
const D = [1..n] dmapped Block([1..n]);

var A, B, C: [D] real; // the three vectors

A = B + alpha * C;      // the computation
```

Because of this plug-and-play characteristic, Chapel supports a separation of roles: Expert HPC programmers can wrestle with the details of implementing efficient parallel data structures within the domain map framework itself. Meanwhile, parallel-aware end-users benefit from their efforts by writing parallel computations using Chapel's high-level notation, without having to understand (or even look at) the low-level implementation details. While this STREAM Triad example is quite simple, the same principle applies to more complex parallel operations as well, such as loops, stencil computations, reductions, etc.

Another important theme in this work is that all of Chapel's standard distributions and layouts will be implemented using the same framework that a typical Chapel programmer would use. We take this approach as a means of ensuring that users can write user-defined domain maps with good performance. It also safeguards against creating a performance cliff when moving from the set of standard domain maps to a user-defined one.

Finally, whereas previous distributed array capabilities have been fairly dimensional and static in nature, we want to ensure that our framework can support very general, holistic, and dynamic implementations of distributed index sets and arrays. Motivating examples include multidimensional recursive bisections, graph partitioning algorithms, and arrays supporting dynamic load balancing.

For a more detailed description of our motivating themes, as well as samples of distributions that our framework was designed to support, please refer to our previous paper [10].

3 Related Work

Chapel's global-view arrays are most closely related to those provided by ZPL [28, 29] and the High Performance Fortran family of languages [22, 23, 18, 11, 1]. Each of these languages supports the concept of an array type that is declared and operated upon as though it is a single logical array even though its implementation may distribute the elements among the disparate memories of multiple distributed-memory nodes. Chapel extends the support for dense and sparse rectangular arrays in HPF/ZPL to include *associative arrays* that map from arbitrary value types to array elements. Chapel also supports *unstructured arrays* that are designed for compactly computing over pointer-based data structures. Chapel supports a language concept for representing first-class index sets like ZPL's *region* but calls it a *domain*. As in ZPL, domains serve as the foundation for defining and operating on global-view arrays.

HPF and ZPL were both fairly restricted in terms of the distributions they supported. HPF supported a small set of regular, per-dimension distributions—Block, Cyclic, and Block-Cyclic—while ZPL traditionally supported only multidimensional Block distributions. In both cases, the distributions were defined by the language and implemented directly in its compiler and runtime. This permitted the language implementors to optimize for their distributions, yet did not provide any means of specifying more general distributions or data structures. In contrast, Chapel provides a general framework for defining distributed arrays and implements all of its standard distributions using the same framework that is available to end-users.

Subsequent work in both HPF and ZPL sought to improve upon their limitations. HPF-2 added support for indirect distributions that could support arbitrary mappings of data to processors [35, 20], though arguably at great cost in space and efficiency. Other extensions to HPF proposed support for distributed compressed sparse row (CSR) arrays by having the programmer write code in terms of distributed versions of the underlying 1D data vectors [32]. Late in the ZPL project, a lattice of distribution types was designed to extend ZPL's generality [15]; however, only a few of these distributions were ever implemented. None of these efforts provide as much generality and flexibility as Chapel's user-defined domain map framework, which is designed to support extremely general array implementations via a functional interface rather than a predefined mapping interface within the language itself.

Unified Parallel C (UPC), a parallel dialect of C, also supports language-based global-view arrays [17], yet it is limited to 1D arrays distributed in a block-cyclic manner. Unlike HPF and ZPL, UPC also supports global pointers that give users the ability to build their own general distributed data structures. This is similar to Chapel's support for programming at lower levels of the language [6], yet without the ability to build up global-view abstractions supported by the language's syntax and compiler. UPC also has the disadvantage of only supporting SPMD-style parallelism compared to Chapel's general multithreaded parallelism.

Unlike UPC, the two other traditional partitioned global address space (PGAS) languages, Co-Array Fortran (CAF) and Titanium, provide multidimensional arrays [25, 34]. However, by our definition their arrays are not global-view since users work with distributed arrays as an array of local per-process arrays rather than a single logical whole. Recent work by Numrich focuses on creating better abstractions for global arrays in CAF by building abstract data types that wrap co-arrays, providing a more global-view abstraction [26]. While this approach bears some similarity to ours, our distributions differ in that they are defined by the Chapel language and therefore known to the compiler and runtime, supporting analysis and optimization. We consider this knowledge key for providing very general operations on global-view arrays without severely compromising performance.

Within the HPC community, a number of interesting data distributions and partitioning schemes have been implemented throughout the years, the most successful of which have typically been libraries or standalone systems [3, 4, 21, 19, 14, 33]. Our goal is not to supplant such technologies, but rather to provide a framework in which they can be used to implement global-view data structures supporting language-based operations. If our approach is successful, not only should the domain map framework be rich enough to support all of these data structures, but Chapel itself should serve as an ideal language for implementing next-generation versions of these technologies.

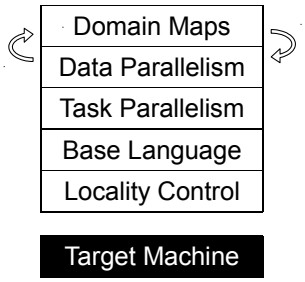

Figure 1: A notional diagram of Chapel's multiresolution design.

4 Chapel Overview

This section provides a brief overview of Chapel as context for this work. For further information on Chapel, the reader is referred to other language overviews [13, 31, 7].

4.1 Language Overview

Chapel was designed with a *multiresolution* approach in which higher-level features such as parallel arrays and domain maps are built in terms of lower-level features that support task parallelism, locality control, iteration, and traditional language concepts [6]. The goal of this approach is to permit programmers to use high-level abstractions for productivity while also permitting them to drop down closer to the machine when required for reasons of control or performance. This is in stark contrast to previous global-view languages like HPF and ZPL in which users had little recourse if the supported set of arrays and distributions failed them. In particular, these languages did not provide any lower-level capabilities for dynamically creating new parallel tasks, nor for having the existing parallel tasks take divergent code paths, create arbitrary per-node data structures, and so forth.

In contrast, Chapel has been designed and implemented in a layered manner in which the higher-level features are implemented in terms of the lower-level ones. Figure 1 provides a notional view of Chapel's language layers, each of which is summarized in the following paragraphs.

Locality Features At the lowest level of Chapel's concept stack are locality-oriented features. Their goal is to permit users to reason about the placement and affinity of data and tasks on a large-scale distributed-memory machine. The central concept in this area is the *locale* type.

A locale abstractly represents a unit of the target architecture that supports processing and storage. On conventional machines, a locale is typically defined to be a single node, its memory, and its multicore/SMP processors. Chapel programmers can reference a built-in array of locales that represents the compute resources on which their program is running. They can make queries about the compute resources via a number of methods on the locale type. In addition, they can control the placement of variables and tasks on the machine's nodes using *on-clauses*, either in an explicit or data-driven manner.

Base Language On top of the locality layer sits the sequential base language with support for generic programming, static type inference, *ranges*, tuple types, CLU-style iterators [24], and a rich compile-time language for metaprogramming. The base language supports object-oriented programming (OOP), function overloading, and rich argument passing capabilities including name-based argument matching and default values. All of these features are embedded in an imperative, block-structured language with support for fairly standard data types, expressions, and statements. Although our user-defined domain map framework could be implemented using existing OOP languages such as Java, C#, or C++, we believe that Chapel's base language features support the complexity of implementing domain maps far more productively.

Task Parallelism The next level contains features for task parallelism to support the creation and synchronization of parallel tasks. Chapel tasks can be created in an unstructured manner using the `begin` keyword. Common cases for supporting groups of tasks are supported by structured `cobegin` and `coforall` statements. Chapel tasks synchronize in a data-oriented manner, either using *synchronization variables* that support full/empty semantics [2] or using transactional sections that can be implemented using hardware or software techniques [30]. Unlike most conventional parallel programming models that rely on a cooperating executables model of parallelism, Chapel tasks are very flexible, supporting dynamic and nested parallelism.

Data Parallelism and Domain Maps Toward the top of Chapel's feature stack are data parallel concepts, which are based on Chapel's arrays (described in detail in the following subsection). Chapel's data parallel operations include parallel loops over arrays and iteration spaces, reduction and scan operations, and whole-array computations through the promotion of scalar functions and operators. The arrows in Figure 1 indicate an interdependent relationship between Chapel's domain maps and arrays in that all Chapel arrays are implemented in terms of domain maps, most of which are themselves implemented using simpler domains and arrays. To break this cycle, Chapel supports a default array layout that is implemented in terms of a C-style primitive data buffer.

4.2 Arrays, Domains, and Domain Maps

A Chapel array is a one-to-one mapping from an index set to a set of variables of arbitrary but homogeneous type. Chapel supports dense or strided *rectangular arrays* that provide capabilities like those of Fortran 90. It also supports *associative arrays*, whose indices are arbitrary values, in order to support dictionary or key-value collections. A third class of array supports unstructured data aggregates such as sets and unstructured graphs. All of these array types are designed to support sparse varieties in which a large subset of the indices map to an implicitly replicated value (often zero in practice).

Chapel arrays are defined using a *domain*—a first-class language concept representing an index set. Chapel's domains are a generalization of the *region* concept pioneered by the ZPL language [5]. In Chapel, domains can be named, assigned, and passed between functions. Domains support iteration, intersection, set-oriented queries, and operations for creating other domains. They are also used to declare, slice, and reallocate arrays. We refer to dense rectangular domains as *regular* due to the fact that their index sets can be represented using $O(1)$ storage via bounds, striding, and alignment values per dimension. In contrast, other domain types are considered *irregular* since they typically require storage proportional to the number of indices.

The main benefit of introducing a domain-like concept into a parallel language is that it simplifies reasoning about the implementation and relative alignment of groups of arrays, both for the compiler and for users. Domains also support the amortization of overheads associated with storing and operating on arrays since multiple arrays can share a single domain.

Chapel domain maps specify the implementation of domains and their associated arrays in the Chapel language. If a domain map targets a single locale's memory, it is called a *layout*. If the domain map targets a number of locales we refer to it as a *distribution*.

Layouts tend to focus on details like how a domain's indices or array's elements are stored in memory; or how a parallel iteration over the domain or array should be implemented using local processor resources. Distributions specify those details as well, but also map the indices and elements to distinct locales. In particular, a distribution maps a complete index space—such as the set of all 2D 64-bit integer indices—to a user-specified set of target locales. When multiple domains share a single distribution, they are considered to be *aligned* since a given index will map to the same locale for each domain. Just as domains permit the amortization of overheads associated with index

sets across multiple arrays, distributions support the amortization of overheads associated with distributing aligned index sets.

An array's elements are mapped to locales according to its defining domain's domain map. In this way, a single domain map can be used to declare several domains, while each domain can in turn define multiple arrays. Chapel also supports *subdomain* declarations, which support semantic reasoning about index subsets.

As an example, the following Chapel code declares a Cyclic distribution named *M* followed by a pair of aligned domains *D1* and *D2* followed by a pair of arrays for each domain:

```
const M = new dmap(new Cyclic(...));

const D1 = [0..n+1] dmapped M,
      D2 = [1..n] dmapped M;

var A1, B1: [D1] int,
    A2, B2: [D2] real;
```

Chapel's domain maps do more than simply map indices and array elements to locales, however. They also specify how indices and array elements should be stored within each locale, and how to implement Chapel's array and domain operations on the data structures. In this sense, Chapel domain maps are recipes for implementing parallel, distributed, global-view arrays. The next section provides more detail on how our framework supports user-defined domain maps.

5 Domain Map Framework

Creating a user-defined domain map in Chapel involves writing a set of three descriptors that collectively implement Chapel's *Domain map Standard Interface* (or *DSI* for short). These DSI routines are invoked from the code generated by the Chapel compiler to implement an end-user's operations on global-view domains and arrays. The descriptors and DSI routines can be viewed as the recipe for implementing parallel and distributed arrays in Chapel. As such, they define how to map high-level operations like Chapel's `forall` loops down to the per-processor data structures and methods that are required to implement them on a distributed memory architecture.

In practice, these descriptors are implemented using Chapel classes. Their methods implement the DSI routines using lower-level Chapel features such as iterators, task parallelism, and locality-oriented features. The classes themselves are generic with respect to characteristics like the domain's index type, the rank of the domain, and the array's element type.

The three descriptors are used to represent the Chapel concepts of (1) domain map, (2) domain, and (3) array, respectively. The descriptors can store whatever state they require to represent the corresponding Chapel-level concepts accurately and to implement the semantics of the DSI routines correctly.

In practice, domain maps that represent distributions tend to allocate additional descriptors on a per-locale basis in order to store state describing that locale's portion of the larger data structure. This helps make the domain map scalable by avoiding the need for $O(numIndices)$ storage in any single descriptor. To distinguish these descriptors, we refer to the three primary descriptors as *global descriptors* and any additional per-locale descriptors as *local descriptors*. It is worth noting that the Chapel compiler only knows about the global descriptors; any local descriptors are simply a specific type of state that a developer may choose to help represent the descriptor's state.

The following three sections describe these descriptors, their state, and their primary DSI routines in more detail. Following that, Section 5.4 describes other descriptor interfaces beyond the required set. For a more detailed technical description of the descriptors and current DSI routines, the interested reader is referred to `technotes/README.dsi` in Chapel release's documentation.

5.1 Domain Map Descriptors

The domain map descriptor stores any state required to characterize the domain map as a whole. Examples might include whether the domain map uses a row- or column-major-order storage layout; the indices to be blocked between locales for a block distribution; the start index for a cyclic or block-cyclic distribution; the block size to be used in a tiled layout or block-cyclic distribution; or the tree of cutting planes used for a multidimensional recursive bisection. For distributions, the global domain map descriptor will also typically store the set of locales that is being targeted.

Since domain maps may be represented using arbitrary characteristics, the constructors for domain map descriptors are invoked explicitly by end-users in their Chapel programs. In our current implementation, this constructor must be wrapped in a new instance of the built-in `dmap` type, representing a domain map value.

In practice, global distribution descriptors use local descriptors to store values that are specific to the corresponding locale. For example, local descriptors may store the subset of the index space owned by that locale; for irregular distributions like graph partitioning algorithms, they may store a subset of the complete distribution's state for the purposes of scalability.

Figure 2 illustrates sample distribution descriptors. It shows a Chapel declaration that creates a new 1D instance of the Cyclic distribution, specifying the starting index as 1 and targeting locales 0, 1, and 2 (denoted in the example as *L0*, *L1*, and *L2*). The left column shows a conceptual view

```
const M = new dmap(new Cyclic(startIdx = 1, targetLocales = Locales[0..2]));
```

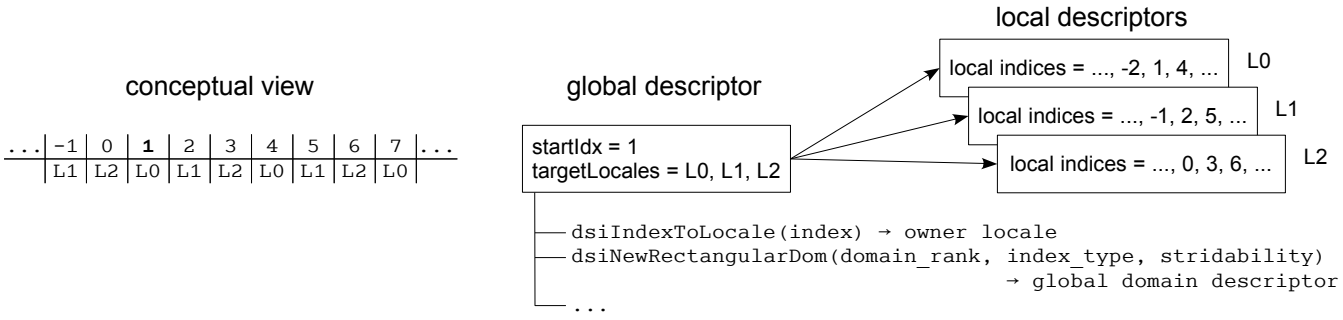

Figure 2: An illustration of the distribution descriptors for an instance of the Cyclic distribution.

of this distribution, illustrating that index 1 is mapped to *L0* while all other 1D indices are cyclically mapped to the locales in both directions. The middle column shows the global descriptor's state, which stores the start index and the set of target locales. The right column shows the local descriptors corresponding to the three target locales, each of which stores the index subset owned by that locale.

The key DSI routines required from a domain map descriptor are as follows:

Index Ownership The domain map descriptor must support a method, `dsiIndexToLocale()` which takes an index as an argument and returns the locale that owns the index. This is used to implement the `idxToLocale` query that Chapel users can make to determine where a specific index is stored. It is also used to implement other operators on domains and arrays.

Create Domain Descriptors A domain map descriptor also serves as a factory class that creates new domain descriptors for each domain value created using its domain map. For example, for each rectangular domain variable in a Chapel program, the compiler will generate an invocation of `dsiNewRectangularDom()` on the corresponding domain map descriptor, passing it the rank, index type, and stridability parameters for the domain value. Similar calls are used to create new associative, unstructured, or sparse domains. Each routine returns a domain descriptor that serves as the runtime representation of that domain value. They also allocate any local domain descriptors, if applicable. Typically, each domain map descriptor will only support the creation of a single type of domain, such as rectangular or associative.

5.2 Domain Descriptors

A domain descriptor is used to represent each domain value in a Chapel program. As such, its main responsibility is to store a representation of the domain's index set. For layouts and regular domains, the complete index set representation is typically stored directly within the descriptor. For distributions of irregular domains that require $O(numIndices)$ storage to represent the index set, a distribution will typically store only summarizing information in its global descriptor. The representation of the complete index set is spread between its associated local descriptors in order to achieve scalability.

For both regular and irregular distributions, local domain descriptors are often used to store the locale's local index subset. These local indices are often represented using a non-distributed domain field of a matching type.

Figure 3 shows a domain declaration that is mapped using the Cyclic distribution created in Figure 2. The left column illustrates that the domain value conceptually represents the indices 0 through 6. The center column shows that the global descriptor stores the complete index set as $0 \ldots 6$. Since this is a regular domain, the global descriptor can afford to store the complete index set since it only requires $O(1)$ space. Finally, the local descriptors store the subset of indices owned by each locale, as defined by the distribution `M`. In practice, these descriptors use a domain field to represent the strided 1D index set locally using the default layout.

The key DSI routines required of the domain descriptors are as follows:

Query/Modify Index Set A domain descriptor must support certain methods that permit its index set to be queried and modified. To implement assignment of rectangular domains, the Chapel compiler generates a call to `dsiGetIndices()` on the source domain descriptor, passing the result to `dsiSetIndices()` on the target domain. These routines return and accept a tuple of ranges to represent the index set in an implementation-independent representation. This supports assignments between domains with distinct distributions or layouts.

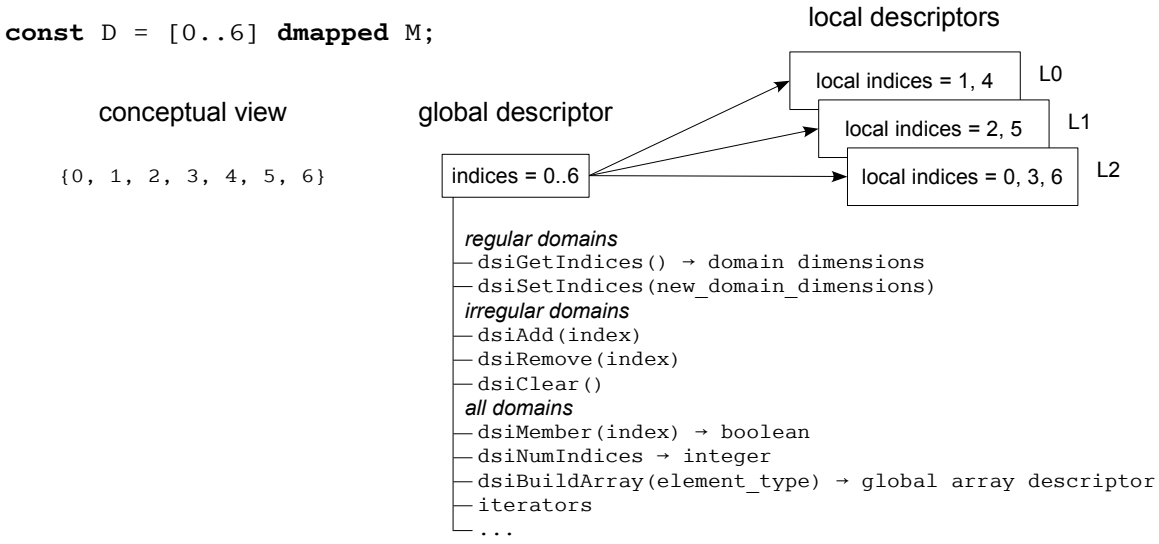

Figure 3: An illustration of the domain descriptors for a domain `D` defining an index set that is mapped using the Cyclic distribution `M` from Figure 2.

Irregular domains support more general `dsiAdd()` and `dsiRemove()` methods that can be used to add or remove indices from the sets they represent. They also support a `dsiClear()` method that empties the index set.

When a distribution uses local domain descriptors, these routines must also partition the new indices between the target locales and update the local representations appropriately.

Query Index Set Properties Domain descriptors also support a number of methods that implement queries on the domain's index set. For example, `dsiMember()` queries whether or not its argument index is a member of the domain's index set. It is used for operations like array bounds checking and user membership queries. Another routine, `dsiNumIndices()`, is used to query the size of a domain's index set. Rectangular domains support additional queries to determine the bounds and strides of their dimensions.

Iterators Domain descriptors must provide serial and parallel iterators that generate all of the indices described by their index set. The compiler generates invocations of these iterators to implement serial and parallel loops over domain values. Parallel iterators for distributions will typically be written such that each locale generates the local indices that it owns. Parallel iteration is a fairly advanced topic in Chapel due to its use of a novel *leader/follower* iterator strategy to support zippered parallel iteration, which is beyond the scope of this paper.

Create Array Descriptors Domain descriptors serve as factories for array descriptors via the `dsiBuildArray()` method. This call takes the array's element type as its argument and is generated by the compiler whenever a new array variable is created. The `dsiBuildArray()` routine allocates storage for the array elements and returns the array descriptor that will serve as the runtime representation of the array. If applicable, `dsiBuildArray()` also allocates the local array descriptors which, in turn, allocate local array storage.

5.3 Array Descriptors

Each array value in a Chapel program is represented by an array descriptor at runtime. As such, its state must represent the collection of variables representing the array's elements. Since arrays require $O(numElements)$ storage by definition, distributions will typically farm the storage for these variables out to the local descriptors, while layouts will typically store the array elements directly within the descriptor. The actual array elements are typically stored within a descriptor using a non-distributed array declared over a domain field from the corresponding domain descriptor.

Figure 4 shows a Chapel array of integer values defined over the domain from Figure 3. The left column illustrates that an integer variable is allocated for each index in the domain. The middle column shows the global descriptor which represents the array's element type, but no data values. Instead, the array elements are stored in the local array descriptors. Each one allocates storage corresponding to the indices it owns according to the domain descriptors.

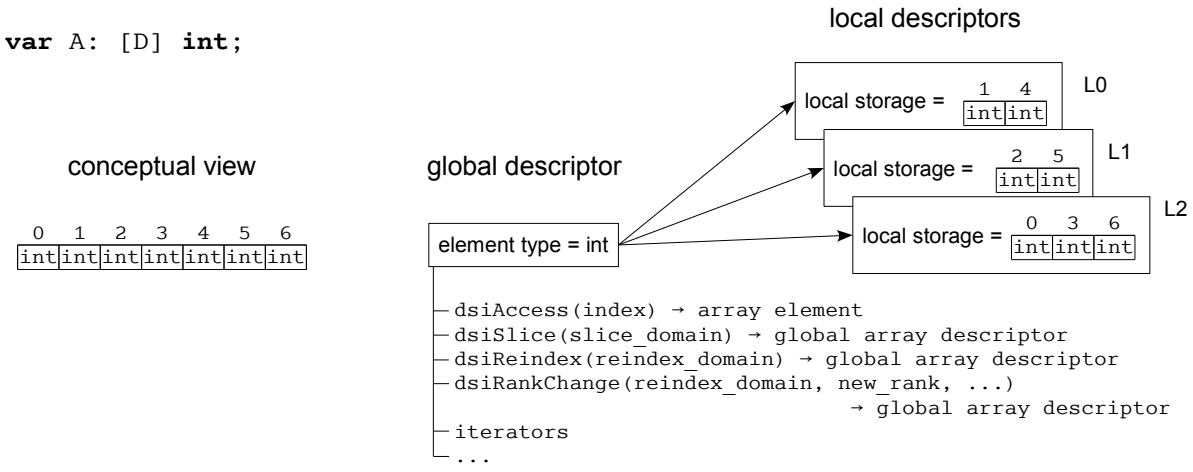

Figure 4: An illustration of the array descriptors for an array `A` of integers defined in terms of the domain `D` from Figure 3.

Though not shown in these figures, a single distribution can support multiple domain descriptors, and each domain descriptor can in turn support multiple arrays. In this way, the overhead of representing an index set or domain map can be amortized across multiple data structures. This organization also gives the user and compiler a clear picture of the relationship and relative alignment between logically related data structures in a Chapel program.

The following DSI routines must be implemented for array descriptors:

Array indexing The `dsiAccess()` method implements random access into the array, taking an index as its argument. It determines which array element variable the index corresponds to and returns a reference to it. In the most general case, this operation may require consulting the domain and/or domain map descriptors to locate the array element's locale and memory location.

Iterators The array descriptor must provide serial and parallel iterators to generate references to its array elements. Invocations of these iterators are generated by the compiler to implement serial and parallel loops over the corresponding array. As with domains, the parallel iterator will typically yield each array element from the locale on which it is stored.

Slicing, Reindexing, and Rank Change Chapel supports array slicing, reindexing, and rank change operators that can be used to refer to a subarray of values, potentially using a new index set. These are supported on array descriptors using the `dsiSlice()`, `dsiReindex()` and `dsiRankChange()` methods, respectively. Each of these methods returns a new array descriptor whose variables alias the elements stored by the original array descriptor. In the case of reindexing and rank change, new domain and/or domain map descriptors may also need to be created to describe the new index sets and mappings.

5.4 Non-Required Descriptor Interfaces

In addition to the required DSI routines outlined in the preceding sections, Chapel's domain map descriptors can support two additional classes of routines, *optional* and *user-oriented*. Optional interface routines implement descriptor capabilities that are not required from a domain map implementation, but which, if supplied, can be used by the Chapel compiler to generate optimized code.

User-oriented interface routines are ones that an end-user can manually invoke on any domains or arrays that they create using the domain map. These permit the domain map author to expose additional operations on a domain or array implementation that are not inherently supported by the Chapel language. The downside of relying on such routines, of course, is that they make client programs brittle with respect to changes in their domain maps since the methods are not part of the standard DSI interface. This is counter to the goal of having domain maps support a plug-and-play characteristic; however it also provides an important way for advanced users to extend Chapel's standard operations on domains and arrays. By definition, the Chapel compiler does not know about these interfaces and therefore will not generate implicit calls to them.

We give some examples of optional interfaces in the paragraphs that follow. We anticipate that this list of optional interfaces will grow over time.

Privatization Interface Most user-level domain and array operations are implemented via a method call on the

global descriptor. By default, the global descriptor is allocated on the locale where the task that encounters its declaration is running. If a DSI routine is subsequently invoked from a different locale than the one on which the descriptor is stored, communication will be inserted by the compiler. As an example, indexing into an array whose global descriptor is on a remote locale will typically require communication, even if the index in question is owned by the current locale.

To reduce or eliminate such communications, Chapel's domain map framework supports an optional *privatization interface* on the global descriptors. If the interface is implemented, the Chapel compiler will allocate a *privatized copy* of the global descriptor on each locale and redirect any DSI invocations on that descriptor to the privatized copy on the current locale.

Each of the global descriptor classes can support privatization independently of the others. A descriptor indicates whether it supports privatization by defining the compile-time method `dsiSupportsPrivatization()` to return `true` or `false`. If a descriptor supports privatization, it must provide the method `dsiPrivatize()` to create a copy of the original descriptor and the method `dsiReprivatize()` to update a copy when the original descriptor (or its privatized copy) changes. The Chapel implementation invokes these methods at appropriate times; for example when a domain is assigned, its descriptors will be re-privatized.

Fast Follower Interface A second optional interface in our current implementation is the *fast follower interface*. This interface supports optimized iteration over domains and arrays by eliminating overheads related to unnecessarily conservative runtime communication checks. To implement this interface, a domain map needs to support the ability to perform a quick alignment check before starting a loop to see whether or not it is aligned with the other data structures in question. As with any optional routines, failing to provide the interface does not impact the domain map's correctness or completeness, only the performance that may be achieved.

Other Optional Interfaces As the Chapel compiler matures in terms of its communication optimizations, we anticipate that a number of other optional interfaces will be added. One important class of anticipated interfaces will support common communication patterns such as halo exchanges, partial reductions, and gather/scatter idioms. At present, these patterns are all implemented by the compiler in terms of the required DSI routines. In practice, this tends to result in very fine-grain, demand-driven communication which is suboptimal for most conventional architectures. As we train the Chapel compiler to optimize such communication idioms in a user's code, it will target these optional interfaces when provided by the domain map author. In this manner we anticipate supporting communication optimizations similar to those in our previous work with ZPL [12, 9, 16].

6 Implementation Status

Our open-source Chapel compiler implements the domain map framework described in this paper and uses it extensively. As stated in Section 2, all of the standard layouts and distributions in our implementation have been written using this framework to avoid creating an unfair distinction between "built-in" and user-defined array implementations. This section describes our current uses of the domain map framework within the Chapel code base. Most of these distributions can be browsed within the current Chapel release in the `modules/internal`, `modules/layouts`, and `modules/dists` directories.

6.1 Default and Standard Layouts

Each of Chapel's domain types—rectangular, sparse, associative, and unstructured—has a default domain map that is used whenever the programmer does not specify one. By convention, these domain maps are layouts that target the locale on which the task evaluating the declaration is running. Our default rectangular layout stores array elements using a dense contiguous block of memory, arranging elements in row-major order. Its sparse counterpart stores domains using Coordinate list format (COO), and it stores arrays using a dense vector of elements. The default associative layout stores domains using an open addressing hash table with quadratic probing and rehashing to deal with collisions. As with the sparse layout, arrays are stored using a dense vector of values. Our unstructured domain map builds on the associative case by hashing on internal addresses of dynamically-allocated elements. Operations on these default layouts are parallelized using the processor cores of the current locale.

In addition to our default layouts, we have implemented some standard layouts that are included in the Chapel code base and can be selected by a programmer. One is an alternative to the default sparse layout that uses Compressed Sparse Row (CSR) format to store a sparse domain's indices. Another is an experimental layout for rectangular arrays that targets a GPU's memory [27]. Over time, we plan to expand this set to include additional layouts for rectangular arrays (*e.g.*, column-major order and tiled storage layouts) as well as additional sparse formats and hashing strategies for irregular domains and arrays.

6.2 Standard Distributions

For mapping to multiple locales, Chapel currently supports full-featured multidimensional `Block`, `Cyclic`, and `Replicated` distributions for rectangular domains and arrays. The `Block` and `Cyclic` distributions decompose the domain's index set using traditional blocking and round-robin schemes per dimension, similar to HPF. The Replicated distribution maps the domain's complete index set to every target locale so that an array declared over the domain will be stored redundantly across the locales.

We also have some other standard distributions that are currently under development. We have a multidimensional block-cyclic distribution that implements the basic DSI functionality, but is lacking some of the more advanced reindexing/remapping functions and doesn't yet take advantage of parallelism within a locale, only across locales. We are also working on a *dimensional* distribution that takes other distributions as its constructor arguments and applies them to the dimensions of a rectangular domain. In this way, one dimension might be mapped to the locales using the `Block` distribution while another is mapped using `Replicated`. Both of these distributions are being implemented to support global-view dense linear algebra algorithms in a scalable manner.

For irregular distributions, we have prototyped an associative distribution to support global-view distributed hash tables. Our approach applies a user-supplied hash function to map indices to locales and then stores indices within each locale using the default associative layout. We have also planned for a variation on the `Block` distribution that uses the `CSR` layout on each locale in order to support distributed sparse arrays for use in the NAS CG and MG benchmarks.

Over time, we anticipate expanding this set of initial distributions to store increasingly dynamic and holistic distributions such as multilevel recursive bisection, dynamically load balanced distributions, and graph partitioning algorithms.

7 Summary

This paper describes Chapel's framework for user-defined domain maps to support a very flexible means for users to implement their own parallel data structures that support Chapel's global-view operations. We are currently using this domain map framework extensively, not only to implement standard distributions like `Block` and `Cyclic`, but also for our shared memory parallel implementations of regular and irregular arrays. While this paper does not contain performance results, our domain map framework was used to obtain our 2009 HPC Challenge performance results which demonstrated performance that was competitive with MPI and scaled reasonably for some simple benchmarks [8]. Since that time we have made additional performance and capability improvements that we plan to showcase at SC11.

At present, our team's main focus is to expand Chapel's standard set of domain maps while improving the performance and scalability of our current ones. As alluded to in Section 5.4, some of our performance optimization work will involve expanding our set of optional DSI routines to reduce overheads for common communication patterns via latency hiding and coarser data transfers. In other cases we simply need to improve the Chapel compiler and domain map code to eliminate communication that is overly conservative or semantically unnecessary. We also plan to continue improving our documentation of the domain map framework so that general Chapel users can start writing domain maps independently. Finally, we need to improve the robustness and scalability of our descriptor privatization machinery which is currently too global and synchronous to support loosely-coupled computations as well as it should.

Acknowledgments

The authors would like to thank David Callahan, Hans Zima, and Roxana Diaconescu for their early contributions to Chapel and their role in helping refine our user-defined domain map philosophy. We would also like to thank all current and past Chapel contributors and users for their help in developing the language foundations that have permitted us to reach this stage.

References

[1] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel Programming: experience with applications, languages, and systems*, pages 42–56. ACM Press, 1988.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[3] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M.J. Garzaran, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of parallel programming*, pages 48–57. ACM Press, March 2006.

[4] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco. Zoltan 3.0: Parallel partitioning, load balancing and data-management services; user's guide. Technical Report SAND2007-4748W, Sandia National Laboratories, Albuquerque, NM, 2007.

[5] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[6] Bradford L. Chamberlain. Multiresolution languages for portable yet efficient parallel programming. http://chapel.cray.com/papers/DARPA-RFI-Chapel-web.pdf, October 2007.

[7] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[8] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and David Iten. HPC Challenge benchmarks in Chapel. (available from http://chapel.cray.com), November 2009.

[9] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

[10] Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi. User-defined distributions and layouts in Chapel: Philosophy and framework. In *HotPAR '10: Proceedings of the 2nd USENIX Workshop on Hot Topics*, June 2010.

[11] Barbara M. Chapman, Piyush Mehrotra, and Hans P. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[12] Sung-Eun Choi and Lawrence Snyder. Quantifying the effects of communication optimizations. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1997.

[13] Cray Inc., Seattle, WA. *Chapel language specification*. (Available at http://chapel.cray.com/).

[14] Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarría-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *Journal of Parallel and Distributed Computing*, 63(9):887–911, 2003.

[15] Steven J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, 2005.

[16] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2003.

[17] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, June 2005.

[18] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR 90079, Rice University, Center for Research on Parallel Computation, December 1990.

[19] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[20] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 2.0*, January 1997.

[21] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[22] Charles Koelbel and Piyush Mehrotra. Programming data parallel algorithms on distributed memory using Kali. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 414–423. ACM, 1991.

[23] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, September 1996.

[24] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[25] Robert W. Numerich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[26] Robert W. Numrich. A parallel numerical library for co-array fortran. *Springer Lecture Notes in Computer Science*, LNCS 3911:960–969, 2005.

[27] Albert Sidelnik, Bradford L. Chamberlain, Maria J. Garzaran, and David Padua. Using the high productivity language Chapel to target GPGPU architectures. Technical report, Department of Computer Science, University of Illinois Urbana-Champaign, April 2011. http://hdl.handle.net/2142/18874.

[28] Lawrence Snyder. *Programming Guide to ZPL.* MIT Press, Cambridge, MA, 1999.

[29] Lawrence Snyder. The design and development of ZPL. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 8–1–8–37, New York, NY, USA, 2007. ACM.

[30] Srinivas Sridharan, Jeffrey S. Vetter, Peter M. Kogge, and Steven J. Deitz. A scalable implementation of language-based software transactional memory for distributed memory systems. Technical Report FTGTR-2011-02, Oak Ridge National Laboratory, May 2011. http://ft.ornl.gov/pubs-archive/chplstm1-2011-tr.pdf.

[31] Chapel Team. Parallel programming in Chapel: The Cascade High-Productivity Language. http://chapel.cray.com/tutorials.html, November 2010.

[32] Manuel Ujaldon, Emilio L. Zapata, Barbara M. Chapman, and Hans P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10), October 1997.

[33] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 24–33. ACM, 2001.

[34] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamato, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998.

[35] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran — a language specification version 1.1. Technical Report NASA-CR-189629/ICASE-IR-21, Institute for Computer Applications in Science and Engineering, March 1992.