An Approach to Data Distributions in Chapel

Roxana E. Diaconescu[a] and *Hans P. Zima*[b,c]

[a]CACR, California Institute of Technology, Pasadena, California 91125
[b]Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California 91109
[c]Institute of Scientific Computing, University of Vienna, Austria

Abstract

A key characteristic of today's high performance computing systems is a physically distributed memory, which makes the efficient management of locality essential for taking advantage of the performance enhancements offered by these architectures. Currently, the standard technique for programming such systems involves the extension of traditional sequential programming languages with explicit message-passing libraries, in a processor-centric model for programming and execution. It is commonly understood that this programming paradigm results in complex, brittle, and error-prone programs, due to the way in which algorithms and communication are inextricably interwoven.

This paper describes a new approach to locality awareness, which focuses on data distributions in high-productivity languages. Data distributions provide an abstract specification of the partitioning of large-scale data collections across memory units, supporting coarse-grain parallel computation and locality of access at a high level of abstraction. Our design, which is based on a new programming language called Chapel, is motivated by the need to provide a high-productivity paradigm for the development of efficient and reusable parallel code. We present an object-oriented framework that allows the explicit specification of the mapping of elements in a collection to memory units, the control of the arrangement of elements within such units, the definition of sequential and parallel iteration over collections, and the formulation of specialized allocation policies as required for advanced applications. The result is a concise high-productivity programming model that separates algorithms from data representation and enables reuse of distributions, allocation policies, and data structures.

1 Introduction

One of the characteristic features of today's high performance computing systems is a physically distributed memory. Efficient management of locality is essential for meeting key performance requirements for these architectures. The standard technique for dealing with this issue has involved the extension of traditional sequential programming languages such as Fortran, C, and C++ with explicit message-passing, in the context of a processor-centric view of parallel computation. This has resulted in complex and error-prone programs in which algorithms and communication are inextricably interwoven.

Existing object-oriented languages, such as Java and C# [18], are limited to supporting shared-memory concurrency [36] or synchronous request-response distributed computing (*e.g.*, RMI [41]). These models do not address locality concerns in non-uniform memory access (NUMA) concurrent computing. The High Performance Fortran (HPF) family of languages provide a high level of support for controlling locality by associating a set of built-in distributions with arrays, while delegating the generation of explicit message passing code to the compiler/runtime system. However, these languages have the disadvantages of being constrained by the semantics of their base language, of providing just a single level of data parallelism, and of supporting only a limited range of distributions. Despite some successes [46, 33], they have not been broadly accepted by the user community.

To fill the gap between these solutions, the HPC community needs a language that provides support for a global view of distributed data aggregates, to improve programmer productivity and relieve scientific programmers from the burden of introducing per-processor communication and detail management into their algorithms. Yet, the parallel

programming expert must be able to implement distributions for these data aggregates in order to optimize their locality and implementation in ways that will benefit a specific algorithm at hand when standard distributions fail them.

This paper describes the design of a powerful facility for defining new data distributions in the context of an object-oriented language. The specific base language selected for that purpose is Chapel, a high-productivity language developed in the Cascade project [6, 8, 13]. User-defined distributions increase the power of the underlying language similar to the way function definitions raise the operational level of a programming language: new data distributions can be generated as first-class objects in a language-provided framework, placed in a library, passed to functions, and reused in array declarations. In the simplest case, the specification of a new distribution can consist of just a few lines of code to define mappings between the global indices of a data structure and memory; in contrast, a sophisticated user (or distribution writer) can control the internal representation and layout of data to an almost arbitrary degree, allowing even the expression of auxiliary structures typically used for distributed sparse matrix data.

Specifically, our distribution framework is designed to support:

- The mapping of arbitrary data collections to units of locality,
- the specification of user-defined mappings exploiting knowledge of data structures and their access patterns,
- the capability to control the layout (allocation) of data *within* units of locality,
- orthogonality between distributions and algorithms,
- the uniform expression of computation for both dense and sparse data structures, and
- reusability and extensibility of the data mapping machinery itself, as well as of the common data mapping patterns occurring in various application domains.

Our approach is the first that addresses these issues completely and consistently at a high level of abstraction; in contrast to the current programming paradigm that explicitly manages data locality and the related aspects of synchronization, communication, and thread management at a level close to what assembly programming was for sequential languages. The challenge is to allow the programmer high-level control of data locality based on the knowledge of the problem without unnecessarily burdening the expression of the algorithm with low-level detail, and achieving target code performance similar to that of manually parallelized programs.

Data locality is expressed via first-class objects called *distributions*. Distributions apply to collections of indices represented by *domains*, which determine how *arrays* associated with a domain are to be mapped and allocated across abstract units of uniform memory access called *locales*. Chapel offers an *open* concept of distributions, defined by a set of classes which establish the interface between the programmer and the compiler. Components of distributions are overridable by the user, at different levels of abstraction, with varying degrees of difficulty. Well-known regular standard distributions can be specified along with arbitrary irregular distributions using the same uniform framework. There are no built-in distributions in our approach. Instead, the vision is that Chapel will be an open source programming language, with an open distribution interface, which allows experts and non-experts to design new distribution classes and support the construction of distribution libraries that can be further reused, extended, and optimized. Data parallel computations are expressed via `forall` loops, which concurrently iterate over domains.

The rest of this paper is organized as follows. After summarizing related research efforts and their relationship to our work in Section 2, we establish the conceptual and notational foundation for the discussion of distributions in Section 3. Section 4 shows how distributions can be defined and used in Chapel, illustrating the power of our framework for specifying new distributions with a set of increasingly complex examples. Section 5 discusses key implementation issues related to the source-to-source translation of such features into locality-aware data-parallel code. Finally, Section 6 states the main conclusion of the paper and summarizes future directions of research.

2 Related Work

Our work builds on research performed by many groups over the past decades. IVTRAN [40], developed for the SIMD architecture ILLIAC IV, was an early language providing high-level control of data distributions. With the advance

of distributed-memory systems in the 1980s, a new class of *data-parallel languages* was explored. In such languages the large data structures in an application are laid out across the memories of a distributed-memory parallel machine. The subcomponents of these distributed data structures can then be operated upon in parallel on all processors. Kali [37] was the first language to introduce distribution declarations in the context of distributed-memory architectures, together with a simple mechanism for user-defined distributions. Fortran D [21], Vienna Fortran [9], and Connection Machine Fortran [1], the major predecessors of the High Performance Fortran [26, 38, 4] effort, all offered facilities for combining multi-dimensional array declarations with the specification of a data distribution or alignment. Several other academic as well as commercial projects also contributed to the understanding necessary for the development of data parallel languages and the required compilation technology [24, 19, 3, 25, 30, 39, 43, 44, 45, 28, 12, 42, 5, 14, 31, 23]. More recently, High Performance Java [35] and a number of parallel Matlab versions [11] have extended their base languages with high-level distribution specifications. The ZPL [7] language supports dimensional distributions organized into types, which give the compiler the information needed to generate loop nests and communication, abstracting the details of the distribution from the compiler's knowledge.

These languages largely rely on a set of built-in mechanisms, such as regular *block* and *block-cyclic* distributions, as well as limited features for irregular distributions, such as *general block* and *indirect*. A major step for defining more general distribution classes was proposed in the Vienna Fortran language specification [48], which introduced a capability for user-defined mappings from Fortran arrays to a set of abstract processors, and for user-defined alignments between arrays. Distribution classes more general than the standard distributions include the Kelp library [20] and the generalized multipartitioning scheme implemented in Rice University's dHPF compiler [15]. The VSIPL++ Library [29] provides C++ classes and functions for a set of standard distributions, with some user-level control.

The class of partitioned global address space (PGAS) languages includes Co-Array Fortran [17], Unified Parallel C (UPC) [27], and Titanium [47]. These languages each provide a set of abstractions for communicating between multiple instances of a single source text running on distinct processors, and represent a reasonable productivity improvement over lower-level communication libraries like MPI. However, they rely on a processor-centric programming model that requires the user to manually decompose their data structures and control flow into per-processor chunks. The one exception to this is UPC, which has support for block-cyclic distributions of one-dimensional arrays over a one-dimensional set of processors (*threads*), and a stylized *upc_forall* loop that supports an affinity expression to map iterations to threads.

Closer to the goals represented by Chapel are two languages developed along with Chapel in DARPA's High Productivity Computing Systems (HPCS) program: X10 [10] and Fortress [2]. Both these languages provide built-in distributions as well as the possibility to create new distributions from existing ones. However, they do not contain features for specifying user-defined distributions and layouts. Furthermore, X10's *locality rule* requires an explicit distinction between local and remote accesses to be made by the programmer at the source language level.

The key differences between existing work and our approach can be summarized as follows. First, we provide a general object-oriented framework for the specification of user-defined distributions, integrated into an advanced high-productivity parallel language. Secondly, our framework allows the flexible formulation of data distributions, locale-internal data arrangements, and associated control mechanisms at a high level of abstraction, tuned to the properties of architectures and applications. This ensures target code performance that is otherwise achievable only via low-level control.

3 Conceptual Framework

In this section we provide a conceptual framework for distributions, which will be used throughout the paper for the specification of their high-level semantics. The lower-level mechanisms provided in the language for controlling the layout of distributed data will later, in Sections 4.5 and 5.1, lead to a refinement of this framework.

3.1 The Chapel Abstract Machine

The Chapel Abstract Machine contains a *memory component* and a *processing component*. Each execution of a Chapel program on the abstract machine is associated with a region of its memory component, called the *execution locale set*, and an unbounded, dynamically managed set of *threads* in the processing component.

The **execution locale set** is a non-empty finite set of identical **locales** determined at the time program execution begins. Locales are units of uniform memory access to which data and threads can be mapped. Accesses of a thread to data are called **local** if thread and data are mapped to the same locale, else **remote**. In terms of performance metrics, a local access is assumed to incur less overhead than a remote access.

3.2 Domains

Domains represent a central element of Chapel, linking index sets, distributions, arrays, and iterators. They generalize features present in other languages, such as HPF's *templates* [26] and ZPL's *regions* [7]. Specifically, a **domain** is a first-class entity, whose principal aspects are:

- An *index set*, which is a finite set of names for identifying components of arrays. Of special importance for scientific computation are the *arithmetic domains*, whose index sets are Cartesian products of sets of integer numbers that form an arithmetic sequence. Their structure is similar to that of Fortran 90 arrays, with a *rank*—known at compile time—that specifies the number of dimensions. However, index sets in Chapel can be more general, such as instances of a class representing nodes in an unstructured grid.
- A *distribution*, which specifies a global mapping from the domain's index set to locales in the execution locale set, as well as the local arrangement of indices and data within locales.
- A set of associated *arrays*, which are mappings from the domain's index set to component variables of a given type. Arrays are allocated according to the domain's distribution. Due to the generality of index sets and types in Chapel the notion of an array is a more powerful concept than its counterparts in traditional languages.

  Conceptually, an array is *always* associated with a domain. However, the language allows the declaration of arrays together with their index set and distribution, without reference to a domain. We speak in this context of *independent* arrays.
- *Iterators*, which are functions defined over the index set of a domain, can be used to control the execution of sequential and parallel loops.

While every domain has a well-defined index set at any time of its existence, its other components are optional. For example, a domain used only for accessing locales in a program may have the index set as its sole component.

3.3 Index Mappings

This section introduces a basic terminology used for modeling distributions.

Definition: Let X, Y denote finite, non-empty sets.

1. An **index mapping from** X **to** Y is a total function $f : X \to \mathcal{P}(Y)$, where $\mathcal{P}(Y)$ denotes the powerset of Y.
2. f is called **proper** iff $f(x) \neq \phi$ for all $x \in X$.
3. For a proper index mapping, f, from X to Y, the set $f(X) := \{y \in Y \mid \exists x \in X : y \in f(x)\}$ is called the **image of** X under f.
4. Given a proper index mapping, f, from X to Y, the **inverse mapping**, f^{-1}, is the inverse of the relation in $X \times Y$ defined by f: for all $y \in Y : f^{-1}(y) := \{x \in X \mid y \in f(x)\}$. If $f(X) = Y$, then f^{-1} is a proper index mapping from Y to X.

5. A proper index mapping is **replication-free** iff $\mid f(x) \mid = 1$ for all $x \in X$. Such a function will be interpreted as a total mapping, $f : X \rightarrow Y$, whenever this is more convenient. If f is surjective, then f^{-1} defines a *partition* of X in the mathematical sense, where all elements of X mapped to the same $y \in Y$ belong to the same equivalence class. □

3.4 Data Distribution for Domains and Arrays

A data distribution is defined for a domain, whose index set is being distributed, and a subset of the execution locale set, which serves as the target of the distribution. The core components of a distribution are two functions referred to as the *global mapping* and the *layout*, both of which are defined over the domain's index set. The global mapping is an index mapping from the domain's index set to the target locales, while the layout maps indices to locations of the locale associated with them via the global mapping, thus allowing the specification of locale-internal data arrangements.

From a programmer's perspective, the global mapping is the primary component of a distribution and must be explicitly specified any time a distribution is defined. In contrast, the specification of the layout may be left to the system, which provides a default layout for any global mapping. Explicit specification of the layout is required if highly sophisticated data representation strategies are to be applied, which are beyond the reach of automatic methods, such as for distributed sparse data structures. In such a case, the layout specification allows virtually complete control over the locale-internal allocation policy. This capability can provide a significant performance gain in situations, where there is a strong correlation between data access patterns and internal data structures reflecting properties of an application that cannot be automatically recognized by the compiler.

The data distribution defined for a domain is applied to all arrays associated with the domain and controls their allocation in memory. [1]

3.4.1 Global Mapping

The global mapping of a domain to a set of target locales associates domain indices with locales. In the following definition we assume a domain *D* with index set *I*, which is distributed to a non-empty subset, LOC, of the execution locale set.

Definition:

1. The **global mapping**, δ, of a data distribution for D is a proper index mapping from I to LOC.
2. D is called the **source domain** of the distribution.
3. LOC is called the **target locale set**; its domain is called the **target domain**.
4. The inverse, δ^{-1}, of the global mapping is called the **ownership function**.
5. Given $loc \in LOC$, $\delta^{-1}(loc)$ is called the **distribution segment** of loc under the global map δ.□

The distribution segment of a locale loc under the global map δ specifies the set of all indices in the source domain, I, which are mapped to loc via δ. The image, $\delta(I)$, of I under δ is called the **actual target locale set** for the distribution. This must be a non-empty set. However, δ is not constrained to be surjective, which means that $\delta(I)$ may be a proper subset of LOC. In that case there exist empty distribution segments.

In general, different distribution segments may contain identical elements. However, if the global mapping is replication-free, then the distribution segments associated with the actual target locale set are pairwise disjoint and constitute a partition of I.

Let A denote an array associated with domain D. Then the mapping $i \mapsto \delta(i)$ for all $i \in I$ determines the set of all locales on which the array component variable $A(i), i \in I$, is to be allocated. Each such locale is called a **home**

[1] The reader familiar with standard distributions such as *block* will notice here that index mappings provide a more powerful mechanism than required since in these cases the global mapping function always associates *exactly one* locale with each index, and not an arbitrary non-empty set of locales. However, our formalism has been chosen such that it can also model replication, in the context of distributions as well as alignments.

for $A(i)$. For every loc in the actual target locale set, $\delta(I)$, the **local array segment** of loc for A is the representation of the portion of A, $A(\delta^{-1}(loc))$, which is associated with the distribution segment of loc. The way in which data are stored in the local array segment is determined by the layout and the array's element type.

As mentioned above, in many cases of interest the global mapping of a distribution is replication-free. However, in some instances the mapping to a powerset is required. Examples include the replication of a "small" data structure (such as a scalar) to *all* locales: $\delta(i) = LOC$ for all $i \in I$, the replication of one or more dimensions of an array across locale dimensions (Section 4.3.2), and the generation of copies of array elements in a halo (Section 4.4). In all these cases, the decision to replicate is motivated by the goal to reduce the communicatio load, while accepting the penalty of increased main memory consumption.

3.4.2 Layout

The **layout** of a data distribution specifies the locale-internal representation of distribution segments and array data in the context of a global mapping. Related functions include methods for accessing data and iterating over index sets.

A detailed discussion of the interface presented to the user for specifying layout will be found in Sections 4.5.3 and 5.1. Here we introduce only one basic function: The *layout mapping* of a data distribution is an index mapping, λ, from the index set of a domain to a set of locations in the locale determined by the global mapping.

For any array A and index $i \in I$, $\delta(i)$ and $\lambda(i)$ are the key components determining the location in which element $A(i)$ is stored.

4 Use and Definition of Distributions in Chapel

In this section, we outline the syntax and semantics of declarations for distributed domains and illustrate the relationships between domains, locales, distributions, and arrays. These features can be understood without requiring knowledge of *how* distributions can be defined by the user.

4.1 Domain Declarations

A typical declaration of a distributed domain has the form:

```
var D: domain_type [dist_spec] [on_clause] [= index_init];
```

where (1) `D` is the identifier of a domain of type `domain_type`; (2) the index set of `D` is obtained by evaluating the expression `index_init`; and (3) the distribution of the domain is specified by `dist_spec`. The source and target locale sets of the distribution are respectively determined as the index set of `D` and the locale set obtained from the `on_clause`.

Details of these components will be discussed in the following subsections.

4.2 Language Representation of Locale Sets and the on-Clause

In a Chapel program, the execution locale set is represented by the predefined rank 1 array of locales declared as:

```
const Locales: [1..num_locales] locale;
```

where `num_locales` is a predefined integer variable initialized with the number of locales in the execution locale set, and `locale` is a predefined type. The initialization of `Locales` is performed implicitly at the time program execution starts. The rank 1 domain with which *Locales* is associated is called the *standard locale domain*, and its index set, [1..num_locales], the *standard locale index set*.

In principle, the access to the execution locale set provided by `Locales` is sufficient to express any program-controlled mapping of data or threads to locales. However, it is often convenient to reflect structural properties of the domain's index set in the target locale set, for example if distributions for a multi-dimensional domain need to be

specified on a dimensional basis (see Section 4.3.2). This goal can be met by allowing the reshaping of `Locales` to a locale array based on a new target locale domain. Reshaping may involve a simple mapping between the standard locale domain and the new domain as well as other array operations such as subscripting, slicing, and subsequence selection. For example, if the value of `num_locales` is 1000, a new target locale array, say `t2D`, can be initialized by reshaping `Locales` as a two-dimensional array with index set [1..100,1..10]:

```
const L2D: domain(2)=[1..100,1..10];          /* new target locale domain of rank 2 */
const t2D: [L2D] locale = reshape(Locales);   /* new target locale array */
```

The mapping between the standard locale domain, [1..1000], and the new domain, `L2D`, is based on lexicographic ordering. Specifically, `t2D`(i, j) serves as an alias of `Locales`$((i-1)*10+j)$ for all $1 \leq i \leq 100, 1 \leq j \leq 10$.

In a Chapel program, linearly ordered sets of locales can be explicitly specified in the context of distributions or statements by the `target_expr` of an **on-clause**, syntactically specified as **on** `target_expr`. For example, let us add to the code above declarations for new domains:

```
const D1: domain(1) distributed(...) on Locales = [1..n1];
const D2: domain(2) distributed(...) on t2D  = [1..n1,1..n2];
const D3: domain(3) distributed(...) on Locales(1..num_locales/2) = [1..n1,1..n2,1..n3];
```

Here, domains `D1` and `D2` are declared as arithmetic domains of respective ranks 1 and 2, which are distributed to locale arrays of corresponding ranks. Domain `D3` of rank 3 is distributed to a one-dimensional subset of `Locales`. The on-clause in declarations is optional, with the default defined as **on** `Locales`.

An example for the use of an on-clause in the context of a statement is given below. The Chapel *forall* statement specifies a parallel iteration over all elements of a domain's index set:

```
forall i in D1 on(D2(i,1) { ... }
```

Let δ_2 denote a replication-free global mapping of the distribution for domain `D2`. The semantics enforced by the on-clause is then that the thread for iteration i must be executed on the locale $\delta_2(i, 1)$ for all i in the index set of domain `D1`. The establishment of such a relationship between threads and a distribution is called **affinity**.

4.3 Basic Distribution Specification for Global Mappings

This section discusses the use of distributions in the context of domain declarations. We deal here only with the global mapping and ignore the layout, assuming an appropriate default for the locale-internal data arrangements.

A distribution is determined as an instance of a user-defined subclass of the predefined class `Distribution` by evaluating the `dist_spec` component of the declaration (Section 4.1) in the context of the domain index set and the target locale set. The syntactic specification of a distribution for an arithmetic domain can be either whole or dimensional. A *whole distribution specification* uses a single distribution class (and mapping function) to determine the global mapping of the source index set, independent of its structure. In contrast, a *dimensional distribution specification* contains a list of component specifications, each of which is associated with one dimension of a multi-dimensional source index set. Each component specification independently determines a distribution class and mapping function for the associated dimension.

An orthogonal classification distinguishes between direct or indirect specification of a distribution. A *direct* specification provides direct information about the properties of the distribution via references to distribution class instances, while an *indirect* or *alignment specification* derives a distribution indirectly by referring to a domain or array with an already defined distribution.

```
const n1 = 1000000;
class MyC: Distribution {/* MyC is introduced as subclass of the predefined class Distribution */
  const z: integer;       /* block size */
  const ntl: integer =... /* number of target locales */

  /* global map for a simplified block-cyclic distribution with block size z≥1;
     the type of argument i is the type of the indices in the source domain: */
  function map(i:index(source)): locale {return Locales(mod(ceil(i/z-1)+1,ntl));}
}

class MyB: Distribution {
  const bl: integer =...;  /* block length */
  /* global map for a simplified regular block distribution with block length bl: */
  function map(i: index(source)): locale {return Locales(ceil(i/bl));}
}
  ...
const D1C: domain(1) distributed(MyC(z=100))=[1..n1];
const D1B: domain(1) distributed(MyB) on Locales(1..num_locales/10)=[1..n1];
var A1: [D1C] float;
var A2: [D1B] float;
...
```

Figure 1: Example: Whole distributions based on user-defined global mappings.

4.3.1 Whole Distribution Specification

A **whole distribution** can be applied to any type of domain. It specifies the distribution of the whole index set of the domain via a single *distribution element*. A direct whole distribution has the form:

distributed (`distribution_element`)

where a *distribution element* is given as C$(a_1, \ldots, a_n)$, with C designating a distribution class, and $a_1, \ldots, a_n$ determining a set of argument expressions. Based on the distribution element, a fully specified distribution is computed as an instance of class C by retrieving the source index set and the target locale set from the context in which the distribution element occurs. Such a distribution element is called *proper*.

The example in Figure 1 illustrates whole distribution specifications based on user-defined distributions introduced in the simplest possible way, by just defining the global mapping function and leaving everything else to the system. We need to emphasize here that this is the *simplest*, not necessarily the best approach. The compiler/runtime system can, in principle, automatically invert the global mapping function, but in many cases (including this example) it will be much more effective if the user overrides the system-provided default by an explicit specification of the inverted function.
Assume `num_locales=1000` in the example. We outline simplified definitions of two distribution classes, `MyC` and `MyB`, both introduced as subclasses of the base class `Distribution`. `MyC` can be used to distribute one-dimensional arithmetic domains in a block-cyclic fashion (with block-length `z`), while `MyB` represents a regular one-dimensional block distribution. `D1C` and `D1B` are declared as one-dimensional domains with the same index set, given as [1..1000000], but different distributions: the distribution of `D1C` is specified by an instance of class `MyC`, with `z` initialized to 100. The global mapping function is evaluated in the context given by the index set of `D1` and the execution locale set represented by `Locales`. The global mapping, δ, is determined as

$$\delta(i) := mod(\lceil \frac{i}{100} \rceil, 1000) \text{ for all } i, 1 \leq i \leq 1000000$$

This is the specification of a block-cyclic distribution with block size `z=100`. For example, index 876543 is mapped to locale `Locales`(766), whose distribution segment is given as

$$\delta^{-1}(766) := \{i \mid i = j * 100000 + 76501 + l, \text{ for all } j, l \text{ such that } 0 \leq j \leq 9, 1 \leq l \leq 99\}$$

The distribution of `D1B` is specified by an instance of `MyB` and evaluated in the context of the target locale set `Locales`(1..100). The global mapping for this distribution is determined as:

$$\delta(i) := \lceil \tfrac{i}{10000} \rceil \text{ for all } i, 1 \leq i \leq 1000000$$

where the block length, `bl`, is 10000. The distribution segments are given as

$$\delta^{-1}(loc) := \{i \mid (loc-1) * bl + 1..loc * bl\} \text{ for all locales } \texttt{Locales}(loc);\ 1 \leq loc \leq 100.$$

Finally, `A1` and `A2` are arrays of floating point numbers whose index sets and distributions are respectively determined by the associated domains `D1C` and `D1B`.

4.3.2 Dimensional Distribution Specification

Dimensional distributions can be applied to arithmetic domains of arbitrary rank. A direct dimensional distribution can be specified in the form:

distributed `(distribution_element-list)`

where the `distribution_element-list` contains n elements for source domains of rank n, which are associated from left to right with the dimensions of the source index set. A distribution element can be either proper, as for whole distributions, or an asterisk, "*", signifying that this dimension does not influence the global mapping function. In this case we also say informally that "this dimension is not distributed". Each proper element links a single dimension of the source index set to a single dimension of the target locale set, providing a distribution for this pair.

The association between source and target dimensions is performed from left to right, skipping non-distributed dimensions. The distribution for the whole index set of the domain is constructed by composing the mappings of the distributed dimensions.

Definition: Consider a dimensional distribution, where the index set of the source domain `D` is given as a regular arithmetic n-dimensional Cartesian product of integer intervals, $I = I_1 \times \ldots \times I_n$, with $n \geq 1$. The list $(d_1, \ldots, d_m)$, where d_j is the j-th distributed dimension of D, counting from left to right, is called the **sequence of distributed dimensions**, where $0 \leq m \leq n$. Similarly, let D', the target locale domain, be an n'-dimensional Cartesian product of integer intervals, $I' = I'_1 \times \ldots \times I'_{n'}$, where $n' \geq 1$. Then:

1. m is called the **rank** of the dimensional distribution and must satisfy: $m \leq n'$.

2. The distributed dimensions of D, $I_{d_1}, \ldots, I_{d_m}$, are one-to-one mapped from left to right to the target locale dimensions $I'_1, \ldots, I'_m$. For each such pair, a global mapping function, δ_{d_k}, is determined by instantiating the distribution class C_{d_k}, and evaluating its *map* function in the context of index set I_{d_k} and target index set I'_k.

3. For each $i = (i_1, \ldots, i_n) \in I$: $\delta(i_1, \ldots, i_n) = \{(i'_1, \ldots, i'_{n'}) \in I' \mid \forall k, 1 \leq k \leq m : i'_k \in \delta_{d_k}(i_{d_k})\}$ □

Figure 2 illustrates examples for dimensional distributions in Chapel. We assume `MyC` and `MyB` as introduced above, and furthermore: `num_locales=1000`, `n1=1000000`, `n2=10`, `n3=16`, `n4=8`, `m1=100`, `m2=10`, and `m3=4`. Two locale arrays are used for specifying distribution targets: (1) `Locales`, representing the standard locale index set, and (2) `t2d`, a locale array of rank 2 with index set [1..100,1..10].

The distribution for `D2` is a dimensional distribution of rank 1, with distribution class `MyB` associated with the first dimension. The global distribution function is determined as follows:

$$\delta(i_1, i_2) = \lceil \tfrac{i_1}{100} \rceil \text{ for all } (i_1, i_2) \in [1..100000, 1..10]$$

```
class MyC: Distribution {...}
class MyB: Distribution {...}
var n1, n2, n3, n4, m1, m2, m3;
...
read(n1, n2, n3, n4, m1, m2, m3);
...
const L2D: domain(2) = [1..m1, 1..m2];
const t2D: [L2D] locale = reshape(Locales);
var D2: domain(2) distributed(MyB(), *) on Locales = [1..n1, 1..n2];
var A3: [1..n3, 1..n4] float distributed(MyB(), MyC(z = 1)) on t2D(1..m3, 1..m3);
...
```

Figure 2: Dimensional distributions example.

`A3` is a two-dimensional independent array with index set [1..16,1..8], which is distributed to the locales subset I'=`t2D(1..4,1..4)` using an instance of `MyB` in the first dimension (with a block length of 4), and of `MyC(z=1)` in the second dimension. The distribution function and the distribution segments can be specified as follows:

$$\delta(4*k+l, i_2) := (k+1, mod(i_2-1, 4)+1) \text{ for all } 0 \le k \le 3, 1 \le l \le 4, 1 \le i_2 \le 8$$

The distribution segments are given as

$$\delta^{-1}(i'_1, i'_2) = (4*(i'_1-1)+1..4*i'_1, i'_2..8 \,\textbf{by}\, 4) \text{ for all } 1 \le i'_1, i'_2 \le 4$$

We conclude this section with a few remarks on properties of dimensional distributions. If $n' > m$, then the dimensional distribution specification provides for an implicit replication of data across $n' - m$ target locale dimensions. The semantics of implicit replication requires all copies of a replicated data item to have identical values. The idea behind this feature is to trade memory space for access speed by converting non-local data accesses to local accesses, at the cost of additional memory space and updates. Our approach follows closely the one implemented in Vienna Fortran [9, 48]. Replication of this type can be also handled differently: for example, ZPL [7] introduces explicitly declared *flood dimensions* for this purpose.

Finally, there are two important special cases for dimensional distributions, which are respectively characterized by $m = n$ and $m = 0$. If $m = n$, then all source dimensions are distributed, and the sequence $(d_1, \ldots, d_m)$ of distributed dimensions is given as $(1, \ldots, n)$. If $m = 0$, then no source dimension is distributed, and $\delta(i_1, \ldots, i_n) = I'$ for all indices of the source domain, i.e., I is totally replicated.

4.3.3 Alignment

Indirect distributions or **alignments** allow the specification of a distribution based upon another already defined distribution. Alignment supports productivity and helps avoiding clerical errors by allowing the reuse of distributions or their components within a well-defined syntactic framework.

Alignment can be expressed in Chapel in a number of ways. The most general mechanism is provided by user-defined distributions. In this section, we slightly modify the previously introduced syntax for direct distribution specifications to cover alignment: (1) the keyword **distributed** is replaced by **aligned** and (2) distribution elements are replaced by *alignment elements*, which do not refer to distribution classes but rather to already distributed domains or arrays. No on-clause is provided in the context of alignment specifications, since the target locale set is derived together with the rest of the distribution information from the alignment elements. In analogy to direct distribution specifications, alignment specifications have a *whole* and a *dimensional* variant.
We illustrate alignment by a simple example. Figure 3 describes a program fragment for dense matrix-vector multiplication. `Mat` is a rank 2 arithmetic domain with index set [1..m,1..n], whose rows are distributed across all locales

```
var m, n;  /* dimension bounds */
...
read(m, n);
...
const Mat: domain(2) distributed(MyB, *) = [1..m, 1..n];
const Row: domain(1) distributed(*) = Mat(2);
const Col: domain(1) aligned(Mat(1)) = Mat(1));

var A: [Mat] float;
var v: [Row] float;
var s: [Col] float;

read(A);
read(v);

s = sum(dim = 2) [(i,j) in Mat] A(i,j) * v(j);
```

Figure 3: Matrix vector multiplication with row distribution.

using `MyB`. The domain `Row`, an arithmetic domain of rank 1, is defined as a projection of the second dimension of `Mat`. Its index set is determined as [1..n], and it is totally replicated across all locales. Finally, domain `Col` of rank 1 is *aligned* with the first dimension of Mat: this means that for all $i \in 1..m$ the global mapping for `Col`(i) is identical to that of `Mat`(i, j) for arbitrary j.

4.4 Halos

Our approach provides a set of language features oriented around the concept of a *halo* for a given data structure. These features support the user-directed optimization of communication for that data structure in a region of the program by allowing:

1. the explicit allocation of copies of non-local elements of the data structure in a locale,
2. the explicit management of coherency between such copies and their home, and
3. the replacement of all non-local references to the data structure by local references.

In a sense, this presents a departure from the stated goal of the Chapel distribution language to avoid explicit management of communication in a program. However, by using halos the experienced programmer may significantly improve the communication behavior of a target program by exploiting knowledge about the application that cannot be automatically derived by the compiler or runtime system. We expect that these features will be used in cases where performance is extremely critical.

Details of the conceptual framework for halos are described in a Technical Report [16].

4.5 User-Defined Specification of Distributions

The Chapel programmer does not necessarily need insight into the details of the specification and the inner workings of a distribution in order to *use* it successfully in an application. A predefined distribution class can be used by just attaching an instance of it to a domain or array declaration, as long as the construct is syntactically valid and it semantic constraints are satisfied. This applies to any distribution, may it be standard or tuned to a class of applications exploiting their special properties. Apart from the knowledge of the interface the programmer only needs guidance for

the effective use of the distribution in view of the array declarations in the program and the related access patterns in algorithms. A key innovation in Chapel is that the language allows the specification of arbitrary distributions. There are no built-in distributions as part of the language and therefore known to the compiler: any distribution required in a program must be explicitly defined and—if to be reused—stored in a library.

In this section we introduce the framework for the specification of user-defined distributions. In a sense, these features can be considered to be at a lower level of abstraction than the rest of Chapel since their implementation interacts with architectural features at a lower level than the other abstractions addressed in the Chapel source language. We can think of specialized *distribution writers* (which of course can be the application developers themselves) being in charge of developing distribution libraries tuned to the requirements of applications.

We begin by explaining the interface of the distribution framework in Section 4.5.1. Distributions can be essentially specified at two levels, one of which deals only with the global mapping while the other uses in addition an explicit layout to control in detail the arrangement of data at the locale-internal level. In the first case (Section 4.5.2), the system provides by default all functionality required for the allocation and management of distributions and distributed arrays based on the user-specified global mapping (and, in most cases, the specification of its inverse). `MyB` and `MyC`, as introduced in Figure 1, illustrate the simplest case for this option. The explicit use of a layout specification will be explained subsequently in Section 4.5.3 by a highly specialized *banded distribution*.

4.5.1 Main Classes

The distribution framework provides the distribution writer with tools for supplying application-specific functionality to the compiler and runtime system via a set of predefined public base classes with an overridable interface.

The base classes involved include `Domain`, `Distribution`, and `LocalSegment`, which are shown in Figure 4 together with public methods. We focus here on the main functionality required in this section, omitting some details that will be used and explained in examples.

A more complete specification of these classes (including their private components) will be provided in Section 5.

The `Domain` class supports the built-in features for domains in the language. Two general iterators are associated with each domain: the sequential `for` and the parallel `forall` iterator. The public method `GetDistribution` provides access to the domain's distribution, while `GetParent` links a subdomain to its parent. A call of the method `extent` yields the number of elements in the domain's index set.

The core component of the framework is the `Distribution` base class. Any user-defined distribution class is a subclass of `Distribution`. The `getSource` method yields the source domain of the distribution, while the `getTargetDomain` and `getTargetLocales` methods respectively yield the target domain and the target locale set. The `map` method represents the global mapping from source indices to target locales. [2] The iterator `DistSegIterator(loc)` produces the elements in the distribution segment associated with locale `loc`, as a sequence of source domain indices. Finally, the function `GetDistributionSegment`, applied to a locale `loc`, yields the domain associated with the distribution segment of `loc`.

The global mapping is the only method that must be always specified by the distribution writer when specifying a new distribution. In addition, the user will in general provide explicit specifications of `DistSegIterator` and `GetDistributionSegment`, although the system, in principle, can determine these functions by automatically inverting the `map` function. In most cases, the specification of these functions by a knowledgable user will significantly enhance performance.

The `LocalSegment` class, a subclass of `Domain`, is a predefined class that provides a default representation of distributions and associated array data in locales. A separate instance of this class is created on every locale in the target locale set of a distribution. This class can be overriden if the user wants explicit control of either mechanism. The function `getLocale` yields the locale for a specific instance of `LocalSegment`. Let `loc` denote such a locale: we discuss the properties of the particular instance of `LocalSegment` for this locale: The value of the public variable `LocalDomain` is the domain for the representation of local array data in locale `loc`. Each array associated with the source domain of the distribution is represented in locale `loc` by a separate array over the domain `LocalDomain`. [3]

[2] The generic type constructor `index`, when applied to a domain, yields its index type.

[3] The constraint that all local array segments associated with a distribution have identical structure can be weakened by including local array segments into the user interface.

```
class Domain {
  iterator for(): IndexType;                          /* sequential iterator */
  iterator forall(): IndexType;                       /* parallel iterator */
  function GetDistribution(): Distribution;           /* distribution of the domain */
  function GetParent(): Domain;                       /* parent domain */
  function extent(): integer;                         /* size of the index set */
}

class Distribution {
  var source: Domain;                    /* the source domain to be distributed */
  var target: Domain;                    /* the target locale domain */

  function getSource(): Domain;
  function getTargetDomain(): Domain;
  function getTargetLocales(): [target] locale;
  function map(i: index(source)): locale;
  iterator DistSegIterator(loc: index(target)): index(source));
  function GetDistributionSegment(loc: index(target)): Domain;
}

class LocalSegment: Domain {
  var LocalDomain: Domain;             /* local data domain */

  function getLocale(): locale;        /* locale associated with an instance of this class */
  function layout(i: index(source)): index(LocalDomain);
  function setLocalDomain(ld: Domain);
  function getLocalDomain();
}
```

Figure 4: Base classes of the distribution framework: public components

For sophisticated problems, such as the representation of distributed sparse matrices, the user may need to generate a set of persistent auxiliary data structures in each locale to support an efficient representation of the distribution and the mapping from global to locale-internal indices. Finally, the `layout` function maps a global source index (that in the given context can be assumed to be in the distribution segment associated with `loc`) to the associated index in `LocalDomain`.

4.5.2 User-Specified Global Mapping

We discuss here an example for user-defined distributions, which deals with the global mapping and leaves the arrangement of locale-internal data structures as well as the definition of the layout mapping to the system. The minimum specification for practical purposes consists of the definition of a subclass of `Distribution`, combined with a definition of the methods `map`, `DistSegIterator`, and `GetDistributionSegment`.

The example considered here, in Figure 5, is a variant of the cyclic distribution introduced in Figure 1. For simplicity we consider here only the case where the block length is 1, and source as well as target domains are arithmetic index sets of rank 1, defining a contiguous interval of integer indices beginning with 1.

The `DistSegIterator` in this specification first determines the number of elements in the source domain, which is assigned to `N`. Secondly, it determines `k` as the number of the locale to which `DistSegIterator` is applied. The iterator successively generates the arithmetic sequence of all values of the form $k + j * ntl$, where $j = 0, 1, \ldots$ and $k + j * ntl \leq N$.

The `GetDistributionSegment` function is similar, except that it produces as its value a domain of rank 1, whose index set is determined by the elements in the arithmetic sequence discussed above.

```
class MyC1: Distribution {
  const ntl: integer=···;     /* number of target locales */

  function map(i:index(source): locale {return Locales(mod(i-1,ntl)+1);}
  iterator DistSegIterator(loc: index(target)): index(source) {
    const N: integer = getSource().extent;
    const k: integer = locale_index(loc);
    for i in k..N by ntl { yield(i); }
  }
  function GetDistributionSegment(loc: index(target)): Domain {
    const N: integer = getSource().extent;
    const k: integer = locale_index(loc);
    return (k..N by ntl)
  }
}

const D1C1: domain(1) distributed(MyC1()) on Locales(1..4) = 1..16;
var A1: [D1C1] float;
```

Figure 5: A cyclic distribution

For the distribution of `D1C1` we obtain: $\delta(i) = mod(i-1,4)+1$ for all $i, 1 \leq i \leq 16$. The distribution segments can be specified as: $\delta^{-1}(k) = k..16$ **by** 4 for $1 \leq k \leq 4$.

4.5.3 User-Specified Layout

In this section, we illustrate the actions required for user-specified layout through a detailed example describing an irregular banded distribution for a square matrix. The specification determines specialized structures for auxiliary data and the actual array storage in each target locale.

Example: Irregular Banded Distribution

In this example [4] we study a *banded distribution* for a square matrix of order n. The problem is specified as follows: Let A denote an array associated with arithmetic domain, `D`, of rank 2 with index set [1..n, 1..n]. For the purpose of this example we define for all $d = 2, 3, \ldots, 2*n$ the *diagonal* $DIAG^d$ as the sequence of all indices (i,j) of D such that $d = i+j$, starting either with an element in the top row or the rightmost column, and proceeding diagonally from top down and from right to left.

In the example of Figure 6, where $n = 9$, the diagonals are identified as $DIAG^2,\ DIAG^3,\ \ldots,\ DIAG^{18}$, where, e.g.,: [5]
$DIAG^2 \ = \ (/(1,1)/)$
$DIAG^5 \ = \ (/(1,4),\ (2,3),\ (3,2),\ (4,1)/)$, and
$DIAG^{16} \ = \ /((7,9),\ (8,8),\ (9,7)/)$.

Now assume $b \geq 1$ to denote a *bandwidth*, and p the number of target locales in the distribution to be constructed. This distribution is characterized as follows:

1. Starting with $d = 2$, diagonals $DIAG^d,\ DIAG^{d+1},\ \ldots,\ DIAG^{d+b-1}$ are mapped to `Locales(1)`. Similarly, the next b diagonals are mapped to `Locales(2)`, and so on, in a cyclic pattern: after the first $b*p$ diagonals have been mapped to locales `Locales(1)` through `Locales(p)`, the process continues for diagonal $b*p+1$ with a mapping to `Locales(1)`, etc.

[4] This problem has been proposed by Brad Chamberlain from Cray Inc.
[5] A sequence of elements $x_1, \ldots, x_m$ is denoted in Chapel in the form $(/x_1, \ldots, x_m/)$.

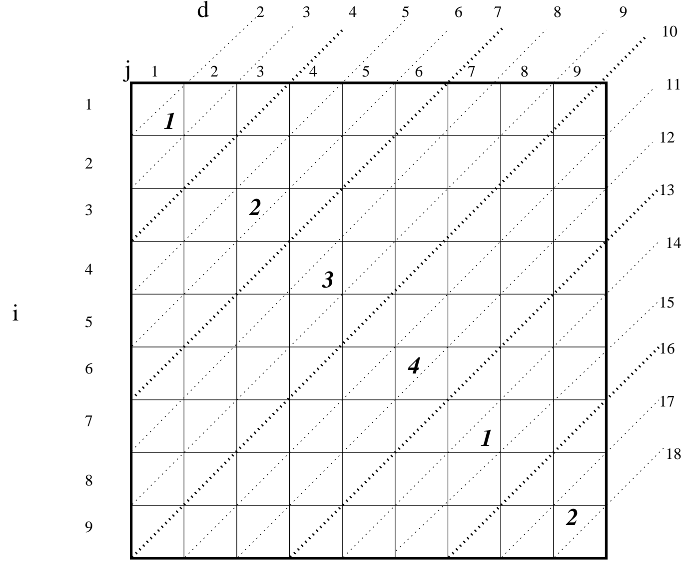

Figure 6: A banded distribution

This results in the following global mapping for index $(i, j) \in [1..n,\ 1..n]$: $\delta(i, j) = ((i + j - 2)/b + 1) \bmod p$. In the matrix of Figure 6, we assume $b = 3$ and $p = 4$. Then $\delta(7, 6) = 4$ (this element belongs to $DIAG^{13}$), and $\delta(8, 9) = 2$. For every $k \in 1..p$, the distribution segments are determined as:

$$\delta^{-1}(k) = \{(i, j) \mid (t * p + k - 1) * b + 2 \leq i + j \leq min(2 * n, (t * p + k) * b + 1), \text{where } t = 0,\ 1,\ \ldots\}$$

In the matrix of Figure 6, the resulting distribution segments are (locales numbered from 1 to 4):

- $\delta^{-1}(1)$ contains the elements in $DIAG^2,\ DIAG^3,\ DIAG^4,\ DIAG^{14},\ DIAG^{15},\ DIAG^{16}$
- $\delta^{-1}(2)$ contains the elements in $DIAG^5,\ DIAG^6,\ DIAG^7,\ DIAG^{17},\ DIAG^{18}$
- $\delta^{-1}(3)$ contains the elements in $DIAG^8,\ DIAG^9,\ DIAG^{10}$
- $\delta^{-1}(4)$ contains the elements in $DIAG^{11},\ DIAG^{12},\ DIAG^{13}$

2. The layout is specified as follows. Let $k, 1 \leq k \leq p$, denote the number of an arbitrary locale used in this distribution, and $DIAGS_k$ a domain, whose index set is the set of all diagonals mapped to k via δ. Then we choose to represent the local domain associated with k as the irregular product domain (see [13], p.115):

$$[DIAGS_k]\ \mathbf{domain}\,(1)$$

For each $d \in DIAGS_k$ the associated subdomain is defined as $1..length(d)$, where $length(d)$ is the number of indices in diagonal d. Thus, an index (i, j) in diagonal $DIAG^d$ can be locally accessed in k in the form (d, l), where l is the position of (i, j) in $DIAG^d$, and positions are counted starting with 1. For example, the element $(7, 6)$ belonging to diagonal $DIAG^{13}$ is mapped to locale 4, and can be locally accessed there via the pair $(13, 4)$.

Figure 7 illustrates a program fragment containing the global declarations in the context of a banded array. Below we show the specifications of the class `Banded`, which defines the global mapping, and of the class `Banded_Layout`, which determines the locale-internal data arrangement.

```
class Banded: Distribution {
  const b: integer;
  const n: integer = getSource()(1).extent();
  const p: integer = getTargetLocales().extent();
  const ndiags: integer = 2*n-1;
  var firstDiagsInDS: [1..p] domain(1);        /* first in every block of b diagonals */
  var diagsInDS: [1..p] seq(integer)= nil;  /* set of all diagonals in distribution segments */
  constructor Banded() {
   forall k in 1..p {
      firstDiagsInDs(k) = (k-1)*b + 2 .. ndiags by b*p;
      forall d in firstDiagsInDs(k) diagsInDs(k) = diagsInDs(k) # d..min(d+b-1,2*n);
   }
  }

  function map(i, j: integer): locale { return Locales(mod((i + j - 2)/b + 1, p)); }
  /* auxiliary functions: */
  /* first component of first element in diagonal d: */
  function first_i(d: index(diagD)): integer{return (if (d<=n+1) then 1 else d-n); }
  /* number of elements in diagonal d: */
  function length(d: index(diagD)): integer){return (if (d<=n+1) then d-1 else 2*n-d+1);}
  /* indices of elements in diagonal d: */
  function diag(d: index(diagD)): seq(integer, integer){
   var AD: seq(integer, integer) = nil;
   for s in 0..length(d) - 1 { AD = AD # (first_i(d) + s, d - first_i(d) - s); }
   return AD;
  }

  iterator DistSegIterator(loc: locale): index(source) {
   const k: integer = locale_index(loc);
   for d in diagsInDS {for i in first_i(d)..first_i(d)+length(d)-1 yield(i, d-i);}
  }
}

class Banded_Layout: LocalSegment {
  const loc: locale=this.getLocale();
  const k: integer = locale_index(loc);
  const DIAGS_k: domain = Banded.diagsInDS(k); /* diagonals in distribution segment loc */

  /* LocalDomain specifies the local index domain of all arrays associated with D. Here we
     chose an irregular product domain that binds each local diagonal to an arithmetic domain
     of rank 1:*/
  const LocalDomain = [DIAGS_k] domain(1);
  constructor Banded_Layout() forall d in DIAGS_k LocalDomain(d)=1..Banded.length(d);
  function layout(i, j: integer): index(LocalDomain) {
   /* Assume k to be the locale index for (i,j), i.e., Banded.map(i,j)=k. The layout function
      performs the mapping (i,j) ↦ (d,l), where d=i+j is the diagonal to which (i,j)
      belongs, and l is the position of (i,j) in the set of all indices belonging to d: */
     return this.index(i + j, i - Banded.first_i(i + j) + 1);
  }
}
```

4.5.4 Distributed Sparse Data Structures

In terms of building the distribution, the generation of a sparse structure differs from that of a dense domain in at least the following points:

- It is necessary to deal with two domains and their interrelationships: the algorithm writer formulates the program based on the original dense domain, indexing arrays in the same way as if they were dense. In contrast, the actual data representation and implementation of the algorithm are based on the sparse subdomain.

```
type eltType;
const n : integer = ...;                                              /*order of matrix */
const bw: integer = ...;                                       /*bandwidth of distribution */
const p : integer = ...;      /*number of locales used: we assume 1 ≤ p ≤ num_locales  */
const D: domain(2) distributed(Banded(b=bw),Banded_Layout()) on Locales[1..p]=[1..n,1..n];
const diagD: domain(1)=2..2*n;   /* the indices in this domain identify the diagonals of D */

iterator across_diagonal(d: index(diagD)): (integer,integer) {
  for i in Banded.first_i(d)..Banded.first_(d) + Banded.length(d) -1 { yield(i,d-i) };
}

var A: [D] eltType;
var dx: (integer,integer);
...
forall d in diagD {
   for dx in across_diagonal(d) { ... A(dx)= ... }
...
}
```

Figure 7: Program using banded array

- In many approaches used in practice, the distribution is determined in two steps: First, the dense domain is distributed, i.e., a mapping is defined for *all* indices of that domain, including the ones associated with zeroes. In general, this will result in an irregular partition, reflecting the sparsity pattern and communication considerations. Secondly, the resulting local segments are represented using a sparse format, such as CRS (compressed row storage).

Our framework is general enough to deal with most aspects of this problem. In particular, the mechanisms provided for extending the `LocalSegment` and `LocalArraySegment` classes allow the construction of persistent auxiliary structures supporting sparse matrices. An open research issue is the automatic generation of efficient code exploiting these specialized data structures based on algorithms essentially formulated in terms of the dense domain. Details are described in [16].

5 Implementation

This section discusses our strategy for the implementation of the distribution framework interface, along with the supporting compiler and runtime structures.

Our framework exposes to the distribution writer a set of well-defined interface functions that can be overridden to express application-specific functionality and optimizations. At the same time, the system conceals low-level distribution aspects such as the distinction between local and remote accesses, communication, and synchronization, and provides efficient default solutions for standard situations.

The compiler uses rewrite rules based on the user-visible interfaces, as well as a set of low-level structures and mechanisms which are not visible to the user, to perform a source-to-source transformation of Chapel locality-aware code into explicitly distributed code augmented with communication calls. In a sense, the distribution class interface plays the role of a *mediator* between the program-specific functionality and compiler transformations and optimizations.

Arrays and domains are implemented as standard Chapel modules using the Chapel language itself. Arrays specialize an abstract `Array` class, while domains specialize an abstract `Domain` class.

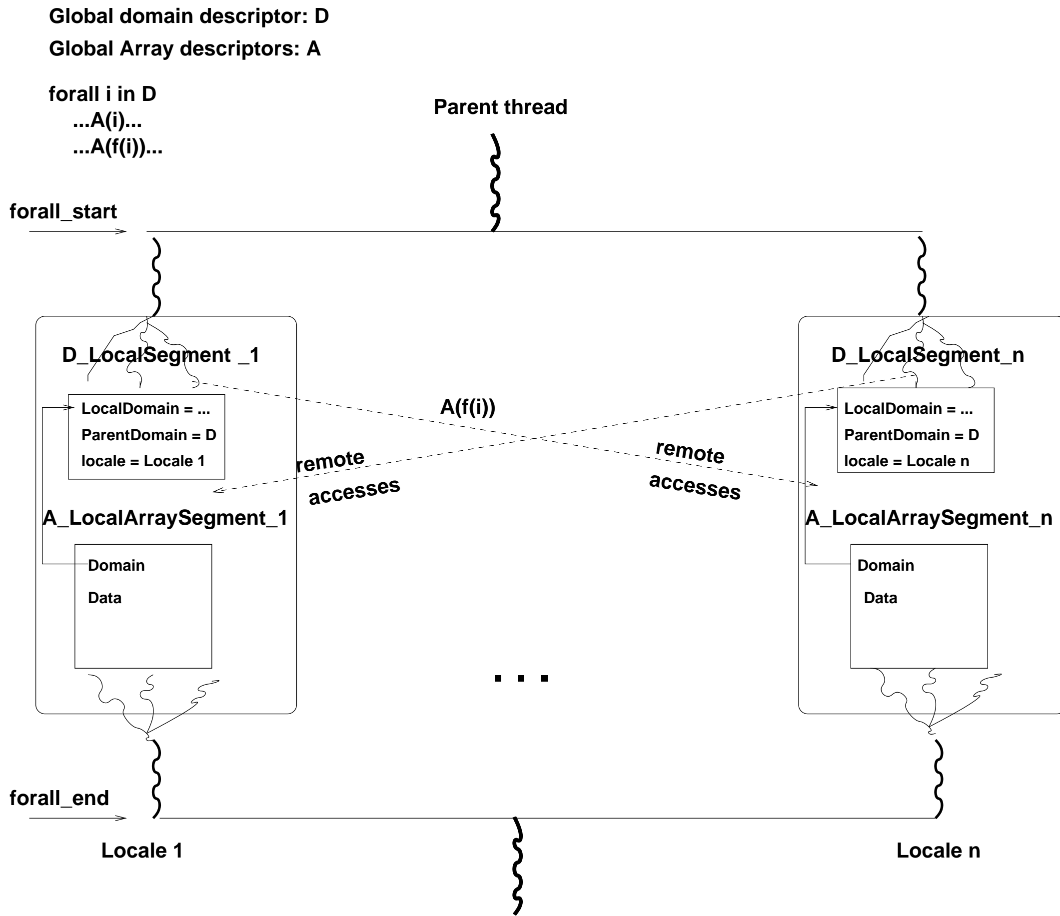

Figure 8: Data-parallel execution model

Chapel supports both task and data parallelism. Here, we focus on the data parallel model: Figure 8 depicts a typical transformation of a `forall` loop, where one thread per locale involved in the computation is spawned upon encountering the `forall` statement. These are heavyweight threads, which account for coarse-grain data parallelism. Within each locale, light-weight threads may be spawned and scheduled depending on the locale-internal configuration.

As an example, we return to the cyclic distribution discussed in Figure 5 of Section 4.5.2. The global map for the distribution of `D1C1` was determined as $\delta(i) = mod(i-1,4)+1$ $(i, 1 \leq i \leq 16)$, and the distribution segments are given as: $\delta^{-1}(1) = \{1,5,9,13\}$, $\delta^{-1}(2) = \{2,6,10,14\}$, $\delta^{-1}(3) = \{3,7,11,15\}$, and $\delta^{-1}(4) = \{4,8,12,16\}$.

Thus, each locale stores exactly 4 elements of array `A1`, whose indices are determined by the distribution segments. The `LocalDomains` of the distribution segments are all defined by default as 1..4, with the associated default layout function performing the nessary index conversion:

$$layout(i) = \lceil \frac{i-1}{4} \rceil \text{ for all } i, 1 \leq i \leq 16$$

Figure 9 outlines the local representations generated by this example.

Data references are classified as *local* if the compiler can assert that the data reside in the same locale as the executing thread; otherwise they are considered *global*. This reflects the fact that Chapel programs can access data

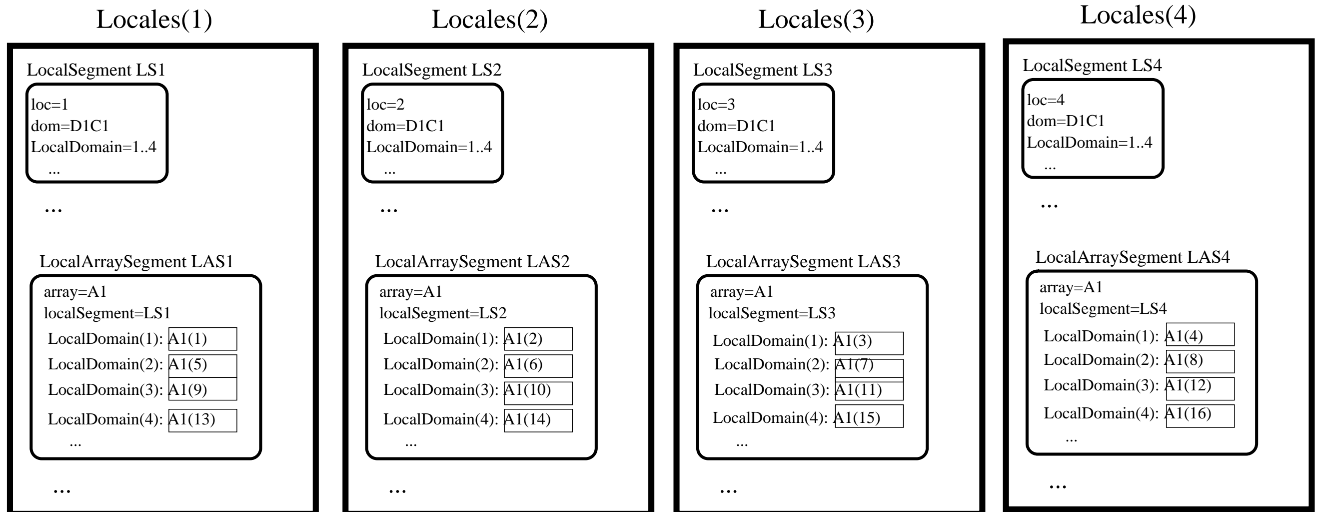

Figure 9: Local representations for cyclic(1) distribution

without taking into account the locality of access. [6] Although for regular codes compiler analyses can often distinguish between local and remote accesses, using this information for the optimization of communication [34, 32], in general this decision can only be made at execution time. In such cases, for example when accessing irregular data structures, optimization of communication must be based on runtime analysis.

Most partitioned global address space (PGAS) languages, such as Co-Array Fortran [17] or X10 [10], deal with this issue by making an explicit distinction between local and non-local accesses at the *source level*. This creates a burden for the programmer but makes it easier for the compiler to generate efficient code.

We have developed a general translation scheme for the data parallel idioms in Chapel. Our scheme primarily addresses (1) the processing of domain declarations, including distribution and local layout assertions, (2) array declarations over distributed domains, (3) array references for arrays associated with user-defined distributed domains, and (4) the transformation of `forall` loops.

5.1 Interfaces

Domains, arrays and distributions, as well as classes representing locale-internal data structures are elements in the Chapel program universe. These classes make up the distribution framework.

5.1.1 The Domain Class

A Chapel domain is implemented as a generic Chapel class, `Domain`. We illustrate its interface—which is a combination of programmer-visible and system-internal functions—in Figure 10. The parts of the interface visible to the programmer are underlined.
The `Domain` class is parameterized by its `IndexType`, the `StorageType`, and the `LocalSegmentType`. The `IndexType` specifies the type associated with the domain's index set. The `StorageType` is a generic type that allows for the differentiation between the types of "storage" used for different kinds of domains. It controls the representation of the index set as well as the representations of the arrays or subdomains associated with the domain, and provides a standard interface for allocation, insertion, and deletion of items. The `LocalSegmentType` is the

[6] In certain contexts the language allows the programmer to assert locality with a special attribute, in order to enhance performance.

```
class Domain {
  /* the domain keeps lists of the arrays and subdomains attached to it, so that
     consistency is guaranteed upon changes in its state: */
  var arrays : List(eltType = Observer(T = IndexType));
  var subdomains: List(eltType = Observer(T = IndexType));
  ...
  type IndexType;                           /* type of the domain's indices */
  type StorageType;
  type LocalSegmentType;

  var storage: StorageType;

  /* associate an array with the domain: */
  function attachArray(array: Observer(T = IndexType));
  /* terminate the association of an array with the domain: */
  function detachArray(array: Observer(T = IndexType));
  function attachSubDomain(subdomain: Observer(T = IndexType));
  function detachSubDomain(subdomain: Observer(T = IndexType));
  function setDistribution(dist : Distribution);  /* define the domain's distribution */

  iterator for () : IndexType;
  iterator forall () : IndexType;

  function getDistribution(): Distribution;  /* determine distribution of the domain */
  function GetParent(): Domain;      /* determine parent domain */
  function extent() : integer;       /* determine the size of the index set */
}
```

Figure 10: The `Domain` class

type of the local segment associated with the domain. The user can override this by providing a `LocalSegment` specialization and defining a `LocalDomain` for it. For the user-provided `LocalDomain`, this type is defined by the user-provided specification. Otherwise, the `LocalDomainType` defaults to the type of the parent domain with indices defined by the corresponding distribution segment.

A distributed `Domain` contains a reference to the `Distribution` class instance with which it is associated in the program. The `getDistribution` method provides access to this instance. For every domain, the system provides default sequential and parallel iterators. The `for` and `forall` methods allow the expression of special behavior by overriding these defaults.

We are using the *observer pattern* ([22], p.293) to describe the relationship between domains and their associated arrays or subdomains. In this pattern, changes in the state of a *subject* are automatically *observed* by a set of *observer* objects that will maintain their state consistent with the state of the subject. Here, the domain is a subject and the associated arrays and subdomains are its observers. Any change in the state of the domain, such as the addition or deletion of indices in a domain, is reflected in the associated arrays and subdomains.

5.1.2 The Array Class

A Chapel array is defined on a domain and does not directly specify data structure or storage constraints—such constraints are embedded in the domain. Arrays are implemented as Chapel classes, with specialized behavior depending on the domain. The following code excerpt illustrates a simplified interface with the distinction between user-visible and non-visible functions.

```
class Array : Observer {
   type eltType;
   type DomainType;
   type StorageType;

   var domain : DomainType;
   var storage : StorageType;

   iterator for () : DomainType.IndexType;
   iterator forall() : DomainType.IndexType;

   function get(i: DomainType.IndexType): eltType {return storage.get(i);}
   function set(i: DomainType.IndexType, v: eltType) {storage.put(i, v);}

   function globalAccess(i: index(DomainType.IndexType): DomainType.IndexType;
   function localAccess(loc: locale,i: DomainType.IndexType):DomainType.IndexType;
}
```

An array is parameterized with (1) a type, `eltType`, common to all its elements, (2) the type of the domain, `DomainType`, it is defined over, and, (3) the storage type, `StorageType`, which depends on the associated domain. The parameterization with the storage type allows the reuse of code across different array types, and provides a standard interface for allocation, addition, and deletion of items.

The functions `globalAccess` and `localAccess` determine the location of an array element in a locale. Whereas `globalAccess` can be used in an arbitrary context, `localAccess` exploits the reduced cost of local accesses in a case where the compiler can assert locality. The programmer does not have direct access to such low-level functionality: however, the specification of the distribution, the domain structure, and the iterators are mechanisms which the programmer can use to control the efficiency of the distributed execution.

The programmer can override the default order of iteration in the `for` and `forall` iterators. This can be useful in the case when an array is not explicitly defined on a domain.

5.1.3 The Distribution Class

Chapel distributions are explicitly defined in a program. A distribution type is implemented as a Chapel class, which specializes the base class `Distribution`.

The interface provided by `Distribution` is depicted in the code excerpt below. Here, `source` and `target` respectively denote the source and target domains of the distribution.

```
class Distribution {
   var source: Domain;                         /* source domain */
   var target: Domain;                         /* target locale domain */

   function getSource(): Domain;
   function getTargetDomain(): Domain;
   function getTargetLocales(): [target] locale;

   function setSource(dom: Domain);
   function setTargetDomain(dom: Domain);
   function setTargetLocales(locales: [target] locale);

   function map(i: index(source)): locale;
   iterator DistSegIterator(loc: index(target)): index(source);
   function GetDistributionSegment(loc: index(target)): Domain;
}
```

The key function a user has to supply for any specified distribution is the global mapping function, map. The layout function in a local segment is a way for the user to specify local addressing explicitly. The combination of these two functions specifies a unique mapping between a global index and an on-locale index.

The DistSegIterator iterator determines for each target locale the indices in the corresponding distribution segment. It can be constructed by default based on the map function and the storage type of the local domain, using an exhaustive search based on the map function. The following code excerpt illustrates this:

```
iterator Distribution.DistSegIterator(loc: locale): source.IndexType {
  forall i in getSource() on loc
    if (map(i) == loc) then yield(i);
}
```

The system-provided version relieves the user from the burden of providing such an iterator. However, regular distributions for which the distribution segment is symbolically computable may not perform well with the system-provided iterator. In this case, the user can override the behavior of the iterator.

An additional function provided by default is GetDistributionSegment. This function, when applied to a locale, loc, yields a domain representing the distribution segment for loc. As before, we note that (1) the system provides a default version for this function, and (2) that the representation of the index set of that domain depends on the type of the domain and can be controlled by the user.

5.1.4 Local Segment and Local Array Segment

The LocalSegment and LocalArraySegment classes are respectively subclasses of the Domain and Array classes, describing the representation of distribution segments and associated local array segments in the target locales of a distribution. These are low-level structures constituting the backbone of implementing distributions and achieving locality. Their role is mostly visible in code transformations for locality-aware execution.

```
class LocalSegment: Domain {
  var LocalDomain: Domain;
  var loc: locale;
  var dom: Domain;

  function getLocale(): locale;  /* locale associated with an instance of this class */
  function layout(i: index(source)) : index(LocalDomain);

  function setLocalDomain(ld: Domain);
  function getLocalDomain();
}

class LocalArraySegment : Array {
  function getArray(): Array;
  function getLocalSegment(): LocalSegment;

  var array : Array;
  var localSegment : LocalSegment;
}
```

A LocalSegment extends the Domain class by adding variables whose values reference (1) loc, the locale which owns the segment, (2) ParentDomain, the parent domain for which it is defined, and (3) LocalDomain, the domain of all local array segments associated with the distribution. This variable can be explicitly defined by the distribution writer; the default assigns a domain with the same type as the parent domain and an index set defined by the distribution segment.

The `LocalArraySegment` extends the `Array` class by adding variables whose values reference the parent array descriptor and the associated instance of `LocalSegment`, whose `LocalDomain` is used to allocate the local array data.

5.2 Program Transformation

This section illustrates a source-to-source transformation from locality-aware features provided by Chapel to a lower-level version of the code, in which the internal components of the interface become visible, and the execution locale set as well as the representation of distributions and distributed arrays are made explicit. The generated program can analyze the locality property of array accesses and insert corresponding optimizations whenever locality of access can be asserted in the compiler.

This lower-level code version is expressed using Chapel, augmented by a pseudo-language construct that helps to deal with the following situation: given a distribution of a domain and its set of target locales, we need to declare and manipulate auxiliary runtime variables which are allocated in each target locale. Such variables are used for the representation of the distribution in the individual locales, as well as for the representation of the associated local array segments. For example, all variables in an instance of the `LocalSegment` class discussed in the previous Section 5.1.4, such as `loc`, `dom`, and `LocalDomain`, belong in this category. The same is true for the components belonging to instances of `LocalArraySegment`. We need not only to declare these local variable instances, but also to access them while iterating over the set of all target locales. While all this can in principle be expressed in Chapel, we introduce the *private* construct to allow a more concise and easily readable formulation. The syntax and semantics of this construct can be outlined as follows:

```
PRIVATE [locale_variable in locale_set] {
 private_variable_declarations
 private_foralls
}
```

Here, the `locale variable` is a variable implicitly declared of type *locale*, the `locale_set` is an expression yielding a sequence of locales, say *LOC*, in the execution locale set (usually, the target locale set of a distribution), and `private_variable_declarations` denotes a sequence of one or more variable declarations that use normal Chapel syntax. The specific convention here is that an instance of each of these variables is allocated in *each locale* in *LOC*. Finally, `private_foralls` is a sequence of one or more private *forall* statements, which are a variant of the Chapel *forall*. Each private *forall* is of the form: **forall** `locale_variable { statement };`, where `statement` is a Chapel statement that will normally access one or more of the variables declared in the construct. This is a short form of the Chapel *forall* statement

```
forall locale_variable in LOC on locale_variable statement_ext;
```

where `statement_ext` is identical to `statement`, except that each occurrence of a variable is implicitly qualified with an instance of the `locale_variable`.

Figure 11 shows a source program fragment containing affine references to two arrays in a forall loop, together with the transformed code. The two domains involved are assumed to have the same index set, `xset`, however they are distributed using instances of two potentially different distribution classes. The target locale sets used by these two distributions are identical, and determined by the value of `myLocales`.

In line 6 of the transformed code, a new domain, `_aDom`, is created by calling the constructor of the domain with parameters derived from the corresponding domain declaration in the source program. In line 7, a distribution is created as an instance of class `XDist` as determined by line 1 of the source program and the target locales; this distribution is associated with the new domain. The array descriptor `_myA` is created in line 8.

Similar actions are performed in lines 9 to 11 for `_bDom` and `_myB`.

The subsequent `private` construct declares instances of the variables `_aDomLocSeg` (line 12) and `_bDomLocSeg` (line 13) for each target locale, which respectively represent the local segments for the distributions associated with

Source Code

```
1.  class XDist: Distribution {...}           /* declaration of distribution class XDist */
2.  class YDist: Distribution {...}           /* declaration of distribution class YDist */
3.  type T;                                     /* array element type */

4.  var myLocaleDom : domain(...)=...;        /* target domain */
5.  var myLocales : [myLocaleDom] locale=...;   /* target locale set */

6.  var aDom : domain(...) distributed(XDist(...)) on myLocales=xset;  /* domain declaration */
7.  var myA : [aDom] T;             /* declaration of array myA associated with domain aDom */

8.  var bDom : domain (...) distributed (YDist(...)) on myLocales = xset;
9.  var myB : [bDom] T;             /* declaration of array myB associated with domain bDom */

10. forall i in aDom on aDom(i) {myA(i) = myB(i)};     /* owner computes */
```

Transformed Code

```
1.  class XDist: Distribution{...}
2.  class YDist: Distribution{...}
3.  type T;

4.  var _myLocaleDom : Domain = Domain(...);
5.  var _myLocales : [_myLocaleDom] locale = ...;

6.  var _aDom : Domain = Domain(...);
7.  _aDom.setDistribution(XDist(source=_aDom,target=_myLocaleDom,targetLocales=_myLocales));
8.  var _myA : Array = Array(_aDom, eltType = T);

9.  var _bDom : Domain = Domain(...);
10. _bDom.setDistribution(YDist(source=_bDom,target=_myLocaleDom,targetLocales=_myLocales));
11. var _myB : Array = Array(_bDom, eltType = T);

    PRIVATE [loc in _myLocales] {
12.    var _aDomLocSeg: _aDom.LocalSegmentType;
13.    var _bDomLocSeg: _bDom.LocalSegmentType;
       forall loc {
14.       _aDomLocSeg= LocalSegment(loc,_aDom,_aDom.getDistribution().getLocalDomain(loc));
15.       _bDomLocSeg= LocalSegment(loc,_bDom,_bDom.getDistribution().getLocalDomain(loc));
       }
     }

    PRIVATE[loc in _myLocales] {
16.    var _locMyA : _myA.LocalArraySegmentType;
17.    var _locMyB : _myB.LocalArraySegmentType;
       forall loc {
18.       _locMyA = LocalArraySegment(_myA, _aDomLocSeg);
19.       _locMyB = LocalArraySegment(_myB, _bDomLocSeg);
       }
    }

20. forall loc in _myLocales {
     if (equal(_aDom,_bDom))               /* identical distributions */
21.    then forall i in _aDomLocSeg
               _locMyA.set(_locMyA.localAccess(loc,i),_locMyB.get(_locMyB.localAccess(i)));
22.    else forall i in _aDomLocSeg
               _locMyA.set(_locMyA.localAccess(loc,i),_locMyB.get(_locMyB.globalAccess(i)));
    }
```

Figure 11: Source program and its transformed code

domains _aDom and _bDom. They are initialized in the following forall loop via calls to the LocalSegment constructor in lines 14 and 15. The local segments default to a system-defined class with a LocalDomain.

The next private construct creates and initializes the local array segments (lines 16 through 19), based on the default LocalSegment and its LocalDomain.

Line 20 starts a parallel loop over the locales and, within each locale, a parallel loop iterates over the local segment corresponding to the main iteration domain. The on clause specifies *owner computes*, i.e., computation takes place on the locale where the elements of _myA(i) are placed.
In line 21, if the domains of arrays _myA and _myB are identical (implying identical distributions), then only local accesses are generated. This check is non-trivial and may incur some time penalty. However, it can be optimized and performed ahead of time, possibly in parallel with other computations. Otherwise, in loop 22, accesses to b are global and thus, less efficient. Pointer analysis or user-provided information can help optimizing the global accesses.

5.3 Performance

A modern programming language for HPC systems will only be accepted by the user community if the performance of its target programs matches the best performance of manually parallelized programs, such as those written using the MPI library. Since the implementation of Chapel has not yet been completed at the time this paper is written, no quantitative data can be provided here. However, key decisions of our design have been made with that goal in mind. We summarize here the most relevant issues:

- **Generality of the Programming Model**
  The HPF family of languages, as well as most other data parallel languages, focus solely on the data-parallel model of computation. In contrast, Chapel provides high-level language support for data as well as task parallelism.

- **Generality of Global Mappings**
  Most parallel languages are restricted to a small set of built-in standard distributions. If such languages are used in the context of advanced applications dealing with irregular data structures, the solution is necessarily suboptimal. The restriction to block and block-cyclic distributions in HPF-1 was decisive in leading to its failure to be accepted by a broad community [26, 33].

- **High-Level Control of Locale-Internal Data Layout**
  Our framework provides detailed control of locale-internal data structures, as well as the formulation of user-defined local or global iterators. These features permit the construction of auxiliary data structures needed for distributed sparse data representations and their efficient manipulation.

- **Interaction Between Compiler and Framework**
  We expect that the above-mentioned interfaces provide the compiler with enough information for generating target code that is decisively better than that produced for HPF. Only practical experimentation will show if the interface is rich enough to compete with manually parallelized programs. Furthermore, in situation where compile-time knowledge is not available (such as for determining the halos and communication schedules for sweeps over irregular meshes), runtime analyses and transformations must be applied independent of the approach used. Our support for halos, for example, reflects very closely the communication management needed in such situations.

6 Conclusion

This paper presented an approach to the design and implementation strategy for language constructs in the context of user-defined data distributions. This work has been based on Chapel, an explicitly parallel high-productivity programming language. It was motivated by the need to provide better language and system support for productively writing parallel programs. An implementation of the language based on a source-to-source transformation is currently in progress.

A number of issues have not been discussed in this paper due to space and time limitations. These include the full syntactic details of the specification of distributions and layouts, the capability of user-defined distributions to specify general alignments, and the definition and implementation of user-defined halos.

There are also some areas in which work is currently in progress, including the efficient management of distributions under a set of domain operations defined in the language, the optimization of runtime support for the dynamic management of distributions and the dynamic optimization of communication for irregular problems along the lines of the inspector/executor approach.

Finally, it is appropriate to mention that, in principle, our framework is generic. Although it has been defined with Chapel as the base language, exploiting its extensive support for domains and object-oriented programming, it could also be redefined with moderate effort in the context of popular object-oriented languages such as Java or C++.

Acknowledgment

This paper is based upon work supported by the Defense Advanced Research Projects Agency under its Contract No. NBCH3039003. The research described in this paper was partially carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

We would like to thank our collaborators and proponents of the Chapel language, David Callahan, Bradford Chamberlain, and Steve Deitz of Cray Inc., for continuously providing ideas and constructive feedback, and exposing interesting issues related to distributions.

References

[1] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Jr. Guy L. Steele. 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] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele Jr., and S. Tobon-Hochstadt. The Fortress language specification version 0.707. Technical report, Sun Microsystems, Inc., July 2005.

[3] F. Andre, J.-L. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, Amsterdam, The Netherlands, June 1990.

[4] Siegfried Benkner and Hans P. Zima. High Performance Fortran for distributed-memory architectures. *Trystram,D.(Ed.): Parallel Computing 25, Special Anniversary Issue*, pages 1785–1825, 1999.

[5] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[6] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60. April 2004.

[7] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Programming Language*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 2001.

[8] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal on HPC Applications*, 2007. In this Special Issue.

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

[10] P. Charles, C.Grothoff, V.Saraswat, C.Donawa, A.Kielstra, K.Ebcioglu, C.von Praun, and V.Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 519–538, 2005.

[11] Ron Choy and Alan Edelman. Parallel MATLAB: Doing it Right. http://www.interactivesupercomputing.com/downloads/pmatlab.pdf.

[12] Pacific Sierra Research Corporation. *MIMDizer User's Guide, Version 7.02*. Los Angeles, CA, 1990.

[13] Cray Inc. *Chapel Specification 4.0*, February 2005. http://chapel.cs.washington.edu/specification.pdf.

[14] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in Split-C. In *Proc.of Supercomputing'93*, pages 262–273, November 1993.

[15] Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarra-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *J. Parallel Distrib. Comput.*, 63(9):887–911, 2003.

[16] Roxana E. Diaconescu and Hans P. Zima. A New Approach to Locality Awareness in High Productivity languages. Technical report, Jet Propulsion Laboratory, California Institute of Technology, June 2006.

[17] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform Co-Array Fortran compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

[18] ECMA. Standard ECMA-334 C# language specification. http://www.ecma-international.org/publications/standards/Ecma-334.htm, June 2005. 3rd edition.

[19] Dennis Gannon *et al.* Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of Supercomputing '93*, November 1993.

[20] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient run-time support for irregular block-structured applications. *J. Parallel Distrib. Comput.*, 50(1-2):61–82, 1998.

[21] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and M.-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Rice University, Center for Research on Parallel Computation, Houston, TX, December 1990.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing. Addison-Wesley, 18th printing edition, September 1999.

[23] Dennis Gannon, Peter Beckman, Elizabeth Johnson, Todd Green, and Mike Levine. HPC++ and the HPC++Lib Toolkit. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 73–108, 2001.

[24] L. Hamel, P. Hatcher, and M. Quinn. An Optimizing C* Compiler for a Hypercube Multicomputer. In Joel Saltz and Piyush Mehrotra, editors, *Languages, Compilers, and Run–Time Environments for Distributed Memory Machines*. North–Holland, Amsterdam, The Netherlands, 1992.

[25] P. Hatcher, A.Lapadula, R.Jones, M.Quinn, and J.Anderson. A production quality c* compiler for hypercube machines. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, April 1991.

[26] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Technical report, Rice University, Center for Research on Parallel Computation, January 1997.

[27] Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the Berkeley UPC compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, New York, NY, USA, 2003. ACM Press.

[28] K. Ikudome, G.Fox, A.Kolawa, and J.Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Fifth Distributed-Memory Computing Conference*, pages 1105–1114, Charleston, South Carolina, April 1990.

[29] High Performance Embedded Computing Software Initiative. VSIPL++ specification. http://www.hpec-si.org, April 2006. Parallel Specification 1.0 Final.

[30] J.Li and M.Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, pages 865–876, New York, NY, November 1990.

[31] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[32] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems*, 21(6):1251–1297, 1999.

[33] Ken Kennedy, Chuck Koelbel, and Hans P. Zima. The rise and fall of High Performance Fortran: An historical object lesson. In *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, San Diego, CA, June 2007. in preparation.

[34] Ken Kennedy and N. Nedeljković. Combining dependence and data-flow analyses to optimize communication. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, 1995.

[35] Indiana University Pervasive Technology Labs. High Performance Java. http://www.hpjava.org, 2004.

[36] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*, chapter 5 (Concurrency Control). Java Series. Addison-Wesley, second edition, January 1997.

[37] Piyush Mehrotra and John Van Rosendale. Programming distributed memory architectures using Kali. In *Advances in Languages and Compilers for Parallel Computing*. MIT Press, 1991.

[38] Piyush Mehrotra, John Van Rosendale, and Hans P. Zima. High Performance Fortran: History, Status and Future. *Zapata,E. and Padua,D.(Eds.): Parallel Computing, Special Issue on Languages and Compilers for Parallel Computers, Vol.24, No.3*, pages 325–354, 1998.

[39] J.H. Merlin. Adapting Fortran 90 array programs for distributed-memory architectures. In Hans P. Zima, editor, *First International ACPC Conference*, pages 184–200. Lecture Notes in Computer Science 591, Springer Verlag, Salzburg, Austria, 1991.

[40] Robert E. Millstein. Control structures in Illiac IV Fortran. *Commun. ACM*, 16(10):621–627, 1973.

[41] Sun Developer Network. Java RMI. available at http://java.sun.com/products/jdk/rmi/.

[42] D. Pase. *MPP Fortran Programming Model*. High Performance Fortran Forum, January 1991.

[43] A.P. Reeves and C.M. Chase. The Paragon programming paradigm and distributed-memory multicomputers. In Joel Saltz and Piyush Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, 1991.

[44] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80, Portland, Oregon, June 1989.

[45] R.Rühl and M.Annaratone. Parallelization of Fortran code on distributed-memory parallel processors. In *International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990. ACM Press.

[46] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. In *Proc.SC2002*, November 2002.

[47] Katherine Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY, 1998. ACM Press.

[48] Hans P. Zima, Peter Brezany, Barbara M. Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran – a language specification. Technical Report 21, ICASE, NASA Langley Research Center, March 1992.