Reusable and Extensible High Level Data Distributions

Roxana E. Diaconescu[a], Bradford Chamberlain[b], Mark L.James[c], Hans P.Zima[c,d]
[a]CACR, California Institute of Technology, Pasadena, CA 91125
[b]Cray Inc., Seattle, WA 98104
[c]Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109
[d]Institute of Scientific Computing, University of Vienna, Austria

ABSTRACT

This paper presents a reusable design of a data distribution framework for data parallel high performance applications. We are implementing the design in the context of the Chapel high productivity programming language. Distributions in Chapel are a means to express locality in systems composed of large numbers of processor and memory components connected by a network. Since distributions have a great effect on the performance of applications, it is important that the distribution strategy can be chosen by a user. At the same time, high productivity concerns require that the user is shielded from error-prone, tedious details such as communication and synchronization.

We propose an approach to distributions that enables the user to refine a language-provided distribution type and adjust it to optimize the performance of the application. Additionally, we conceal from the user low-level communication and synchronization details to increase productivity. To emphasize the generality of our distribution machinery, we present its abstract design in the form of a *design pattern*, which is independent of a concrete implementation. To illustrate the applicability of our distribution framework design, we outline the implementation of data distributions in terms of the Chapel language.

1. INTRODUCTION

Today's massively parallel High Productivity Computing Systems (HPCS) are characterized by a modular structure, with a large number of processing and memory units connected by a high-speed network. Locality of access as well as load balancing are primary concerns in these systems that are typically used for high performance scientific computation. Data *distributions* address these issues by providing a range of methods for spreading large data sets across the components of a system. Over the past two decades, many languages, systems, tools, and libraries have been developed for the support of distributions. Since the performance of data parallel applications is directly influenced by the distribution strategy, users often resort to low-level programming models which allow fine-tuning of the distribution aspects affecting performance, but, at the same time, are tedious and error-prone.

In this paper we propose a novel design for the high-level specification of distributions in data parallel applications. Our design is shaped by previous experiences in language and run-time support for distributions as well as by design patterns in manually distributed code. The elements of our distribution machinery are introduced in the Chapel [4] programming language. They include *domains*, *index sets*, generalized arrays and *user-defined distributions*. To stress the generality of our design, we first present the abstract distribution design pattern and its elements, independent of the Chapel language. Then, we discuss the implementation of the design in the context of the language.

This paper is organized as follows. Section 2 introduces the distribution design pattern using the presentation format introduced by Gamma et.al. [10]. Section 3 discusses the implementation of the distribution design in the context of the Chapel language. Section 4 reviews related work. Section 5 concludes the paper.

2. THE DISTRIBUTION DESIGN PATTERN

2.1 Intent

The intent of our distribution design pattern is to capture the common elements of data distributions in scientific computing. Data distributions define the interface for distributing the elements of a collection across multiple *units of locality* without constraining the type of elements or indices in the collection. We use the term *unit of locality* to denote the building block for a computer system that is made of multiple similar components. Each component has memory and operation capabilities.

2.2 Motivation

Consider a high-performance computing architecture consisting of a large number of units of locality. Memory is collocated with the unit, causing local accesses to be less expensive than remote accesses. Examples of such configurations include non-uniform memory access (NUMA) architectures, clusters of processors, and emerging peta-scale architectures such as Cascade [4].

A data distribution partitions a collection of data and distributes its elements across units of locality. To achieve high performance, this means a distribution strategy must account for access locality as well as load balance. Although some approaches towards automatic data distribution have been explored in the past, in general it is useful to let the user choose the distribution strategy based on existing knowledge about a parallel application. However, once the distribution strategy is chosen, the system should take over the low-level, error-prone tasks such as communication generation and synchronization enforcement.

A distribution decision should specify, for a particular data item, its associated unit of locality and the layout within that unit. Sometimes, depending on the structure of data, the latter can be determined automatically.

Consider a program that solves a Poisson equation to compute the pressure field over the points of a discrete domain using the Finite Element Method. Depending on the concrete application, various discretization strategies may be considered for a given domain. Thus, the same algorithm may be used with a regular mesh described by cubic elements, or with an irregular mesh consisting of tetrahedral elements. In both cases, the underlying parallel structure of the application is identical. Moreover, the data access, and consequently, synchronization structures are identical as well. However, since in a conventional approach the different data representations are encoded in the algorithm, these similarities may not be detectable by looking at the source code.

Thus, we wish to introduce the appropriate abstractions to capture these similarities and separate the algorithm from details of its data representation. Specifically, we would like to decouple indices from collections of data items using index sets and domains. *Index sets* provide names for the components of collections. *Domains* are entities that specify an index set and its distribution. *Data collections* are defined over domains and will be distributed accordingly.

In the previous example, the regular mesh can be a three dimensional domain with an index set that is a regular Cartesian product. Collections of data items over the domain include the pressure vector which is defined at every point in the mesh domain and can be represented as an array over the domain. A data distribution specifies, for each component of the index set — `(i,j,k)` — the unit of locality it belongs to and, potentially, the offset within that unit.

The irregular mesh can be a domain indexed by references to element objects. In this case, the index set is a collection of element objects which name the components of data items defined over the irregular domain. A data distribution specifies, for each component of the index set (`Element e`), the unit of locality it belongs to and, potentially, the offset within that location.

In addition, we would like to capture the behavior of the distribution machinery via a common interface which has various implementations defining commonly used classes of distributions such as block, cyclic, and indirect, as well as novel user-defined distributions. Two key operations exported by the distribution interface are the definition of the mapping of indices to units of locality (`Map(Index)`) and their layout (`LocalLayout(Index)`) within the units of locality. In general, our approach provides a much richer capability that supports the needed sophisticated data representations that would, for example, be required for the implementation of distributed sparse data collections.

The domain interface includes a `Distribute` method which is parameterized by the distribution class. This method specifies the distribution for the domain it is invoked on.

2.3 Applicability

Distributions for data parallel applications have been extensively used in software tools, libraries and applications. The various approaches include manually specified distributions, automatically distributed applications with compiler and run-time support, component and object-oriented frameworks, and skeletons for scientific applications.

Our design can be used for data parallel applications when:

- a numerical algorithm should be reusable regardless of the geometry or physical structure of its input data,
- multiple distribution strategies need to be studied to investigate the best approach, and
- low-level communication and synchronization details should be concealed from the user.

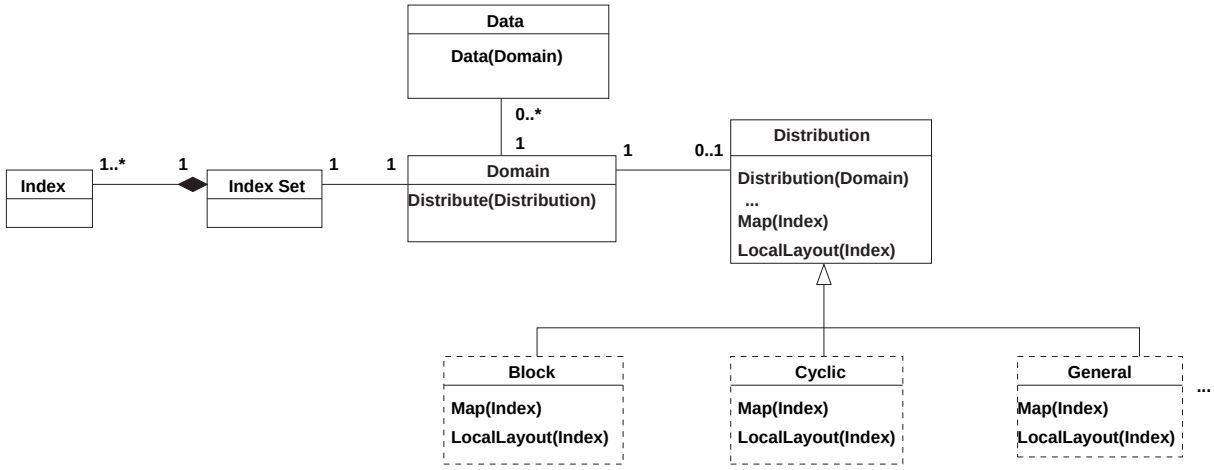

Figure 1: Distribution elements.

2.4 Structure

Figure 1 depicts the elements involved in defining a distribution. A `Domain` has one `Index` set associated with it and one `Distribution`. A `Distribution` class interface includes operations that allow the specification of the mapping of an index to a unit of locality. Thus, `Map(Index)` specifies the unit, while `LocalLayout(Index)` specifies the local address within that unit. The `Distribution` class can be extended to specify commonly used distributions or novel user-defined distributions. A data collection `Data` can be defined over that domain. The collection maps the domain index set to the variables in the collection. Since the index set is distributed, the addresses of the variables in the collections will be distributed as well.

2.5 Participants

- **Domain**
  is a description of a collection of names for data. These names are referred to as the indices of the domain. All indices for a domain are values with some common type. It consists of:
  - an index set,
  - a distribution of that index set, and
  - a set of associated data collections. All these collections share the index set and its distribution with the domain but can have different data types.
- **Index**
  defines the *type* of indices: this includes integer tuples (in the case of regular Cartesian product index sets) and object references, such as instances of a `node` class in a dynamic graph structure.
- **Distribution**
  is a mapping from index values to units of locality. A distribution allows a user to specify data locality and alignment by overriding its default behavior. Its interface includes:
  - a mapping from indices to units of locality
  - a mapping from indices to offsets within units of locality. We call this a *local layout*.
- **Concrete Distribution**
  overrides the mapping operations to implement a particular distribution (i.e. `Map` and `LocalLayout`). The concrete distribution is a specialization of the `Distribution` type.
- **Data Collection**
  are abstractions of mappings from index sets to variables. Arrays are one example of such mapping.

2.6 Collaborations

A user who wants to use distribution for a data parallel program, must create a concrete distribution by providing the mapping and local layout information.

When a domain is created, the user specifies its index set *type* and its distribution *type*. The `Distribute` method can be applied to the domain with the created distribution as parameter. As a consequence, all data collections defined on the domain will be distributed according to the specified distribution. Accesses to distributed data are implemented to handle any required communication transparently.

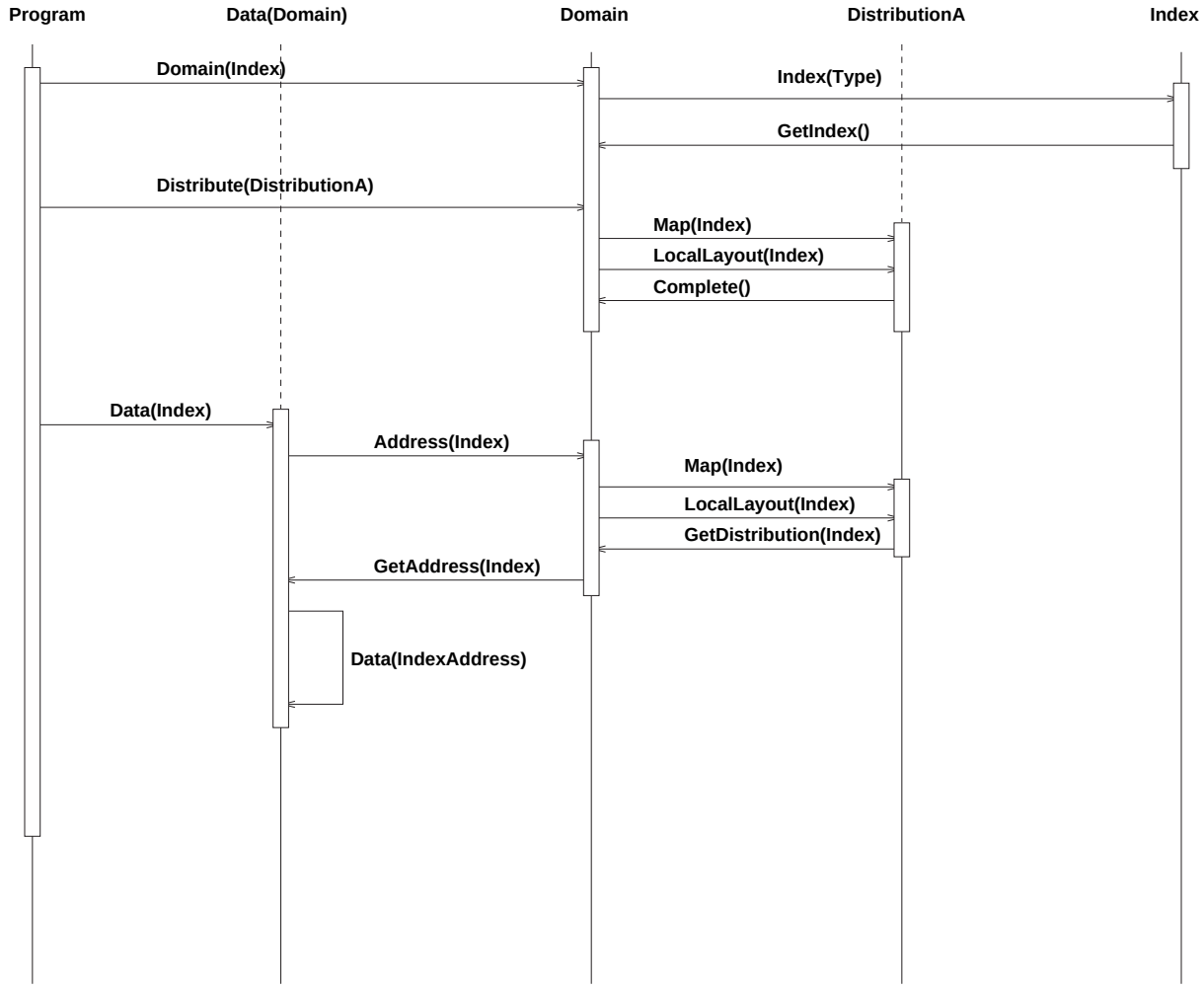

Figure 2: The interaction between distribution elements.

Figure 2 depicts the collaborations between a domain, its distribution and a data collection defined on the domain. A program may contain domain declaration statements which trigger the creation of a domain based on the user provided information on its index type and distribution type. In Figure 2 the `DistributionA` is a user-defined concrete distribution for a domain. When instantiating a domain with a particular distribution type, the index set of the domain is mapped according to the `Map` and `LocalLayout` operations for every index in the index set. `Data(Domain)` is a data collection defined on the domain. Each access to a data item given by an index is translated to reflect its distribution.

2.7 Consequences

There are a number of benefits and liabilities of our design for user-defined data distributions:

1. The distribution class hierarchy allows a user to define distributions which are more suitable for an application than hard-wired distributions. A user needs to specify the mapping of indices to units of locality and the local layout within the units. More sophisticated control of the data arrangement can be specified if required.

2. The domain abstraction allows the user to define domains which are closer to the physical domain by extending index types to include object references. Also, the index set of a domain may have an inherent linear order (as in the case of Cartesian products of integers), or may be arbitrarily ordered (as, e.g., object references).

3. Data collections are associated with domains, and thus their structure is defined in the domain rather than in the data itself. As a consequence, the programmer can define collec-

tions of data with complex and irregular structure without complicating the data representation itself.

4. Giving the user full freedom in specifying the distribution increases the burden on ensuring good run-time performance. Our approach provides support for the optimization of data distributions according to various criteria such as memory overhead, iteration strategies, and data access performance. Specialized distributions can be made part of standard libraries.

3. IMPLEMENTATION

Our design is implemented using the Chapel language and conceptually runs on a Chapel abstract machine. Most of the elements described in the previous sections are part of the language and therefore are implemented as first-class entities.

An execution of the *Chapel Abstract Machine* determines an *execution locale set*, which is an arrangement of identical **locales**. Locales represent the units of locality in Chapel. The size of the execution locale set is determined at the time the program begins execution, and remains invariant thereafter. Data and computations can be mapped to locales with the understanding that entities mapped to the same locale are closer to each other than when they are mapped to different locales. Different data objects that are mapped in the same way to a set of locales are said to be *aligned*. If a computation is mapped to the same locale as a data object accessed by it we say that there is an *affinity* between the computation and the data object, resulting in a **local** access; otherwise, the access is **remote**. In terms of the performance metrics, a local access is less expensive, i.e., burdened with less overhead, than a remote access.

A distribution maps indices to locales and a location within each locale. The execution locale set is globally declared as:

```
var Locales:  array [1..num_locales] of Locale;
```

3.1 Domains and Arrays

Domains are first-class entities that have a set of indices which may be bounded or unbounded. In the former case the domain is called *definite*, while in the latter, *indefinite*.

For each domain, there is a corresponding index type which includes primitive types, enumerated types, and class reference. There are two categories of domains:

1. Arithmetic domains have bounded index sets with an integer index type.
2. Indefinite domains have unbounded index sets and any of the index types listed above.

An example for an arithmetic domain declaration in Chapel is:

```
var rMesh :  domain(3) = [1..m, 1..n, 1..p];
```

This specifies a domain, `rMesh`, with an arithmetic index set of rank 3. The index set is initialized by a Cartesian product.

Let us define an irregular mesh as being composed of geometrical elements, their faces and vertices. The definition of such structure in Chapel is:

```
class Element {
    var idx : integer;
    var nfe : integer;
    var elemFaces[1..nfe] : Face;

    class Face {
        var idx : integer;
        var nvf : integer;
        var faceVerts[1..nvf] : Vertex;
    }

    class Vertex {
        var idx : integer;
        var dim : integer;
        var coords[1..dim] : float;
    }
}
```

An irregular domain over such mesh can be declared in Chapel as:

```
var iMesh :  domain(Element);
```

Elements can be added to a domain and removed from a domain by using Chapel predefined operations for indefinite domains `add(elem)` and `remove(elem)`.

Data collections are defined over domains. Chapel provides support for a generic notion of an array that includes Fortran arrays as a specific instance. Therefore, for the previously declared arithmetic and indefinite domains, we can define data collections as follows:

```
var regArray[rMesh] :  float;
var iregArray[iMesh] :  float;.
```

3.2 Distributions

Distributions are a means to exploit locality in Chapel. A distribution is a mapping from domain index sets to locales. A programmer

can describe affinity between data and computation by associating them with abstract *locales*.

A distribution type is defined as a class that can be extended to express user-defined distributions. The Chapel interface for a distribution is:

```
class distribution {
  ...
  function SetDomain(d1 : domain);
  function GetDomain() : domain;

  function SetTarget(t : locale[]);
  function GetTarget() : locale[];

  function Map(i : index) : locale;
  function LocalLayout(i : index) : location;
  ...
}
```

The Chapel compiler is written in the C++ programming language. Internally, a domain is represented as a C++ class `Domain`, an index as a C++ class `Index`, and a distribution as a C++ class `Distribution`. The user overrides the functionality of the `Distribution` class by providing a concrete domain to instantiate the distribution and concrete implementation for the `Map` and `LocalLayout` operations.

A user-defined block distribution can be written in Chapel as:

```
class block{
  implements distribution;
  block_size : integer;

  --la is the target locale array
  constructor create(d : domain, in la : locale[],
                     in  bs : integer) {
    this.SetDomain(d);
    this.SetTarget(la);
    block_size = bs;
    ...
  }

  function Map(in i: index) : locale {
    return this.GetTarget()(ceil((i-1)/block_size)+1);
  }

  function LocalLayout(var in i : index) : location {
    return (mod(i-1, block\_size)+1);
  }
```

To simplify the presentation we leave out the lower and upper bounds for the index sets. Thus, the code above implicitly assumes that the lower bound is 1. The block size can either be specified by the user or computed by the system.

Distributions can be specified for domains and, as a consequence, all data collections defined on a given domain will be distributed according to the specified distribution:

```
var rMesh:domain(3)=[1..m,1..n,1..p] dist(block,block,block)
on L3D;
var iMesh :  domain(Element) dist(general);.
```

The first declaration defines a *dimensional* distribution. Each dimension of the array `regArray` associated with the `rMesh` domain will be block distributed on the corresponding dimension of the three-dimensional locale array, `L3D`. The irregular array `iregArray` associated with the `iMesh` domain will be distributed according to a general distribution specified by the user on the available locales. The following code excerpt shows how this can be done:

```
var iMesh : domain (Element) dist(general);
  ...
  for i in iMesh {
    ...
    i.locale = f(i,...);
    -- this maps index i to a locale determined by function f
    ...
  }

  complete(D);
  -- this statement ‘‘completes’’ the distribution by defining
  -- the corresponding fields in the domain;
  -- it can only be called after all indices in D have been
  -- mapped;
}
```

Here, the function call `f(...)` references an arbitrary user-defined function that establishes the point-to-point mapping between a domain index and a locale.

Concurrent execution is supported via the Chapel `forall` construct. Distributed collections are iterated over using this statement. The iteration space is split according to the data distribution and local accesses are grouped within the same locale:

```
forall i in rMesh {
  ...
  regArray(i) = regArray(i-1) + ...;
}

forall i in iMesh {
  ...
  iregArray(i) = ...;
}
```

It is beyond of the scope of this paper to include details on code generation and optimization for the Chapel compiler. Because the implementation is an ongoing effort we are still in the process of evaluating our approach for user-defined distributions and its impact on efficiency.

4. RELATED WORK

There has been significant effort in the area of distributions for data parallel applications and we will review two of those approaches: (1) Early work on distributions in support of Fortran related languages and (2) Object-based systems, libraries and skeletons for scientific programs.

4.1 Distribution Support for Fortran and Related Languages

1 The first language to allow users to control the local layout of data was IVTRAN [20], which was developed for the SIMD machine ILLIAC IV. Kali [19] (and its predecessor BLAZE) were among the first languages to introduce distribution declarations in the context of distributed-memory systems. Kali allows the dimensions of an array to be orthogonally mapped onto an explicitly declared processor array using simple regular distributions such as *block* and *cyclic*, and more complex distributions such as *irregular* in which the mapping of each array element is explicitly specified. Simple forms of user-defined distribution were also permitted. Parallel computation was specified by means of *forall* loops within a global name space.

SUPERB [22] is an interactive restructuring tool, which translates Fortran 77 programs into message-passing Fortran for distributed-memory architectures. The user specifies the distribution of the program's data via an interactive language; based on compiler-provided analysis, the user selects a transformation strategy for the coarse-grain parallelization of the program for a distributed-memory machine.

The Fortran D project [9, 15] follows a slightly different approach to specifying distributions. The distribution of data is specified by first aligning data arrays to virtual arrays known as *decompositions*. The decompositions are then distributed across an implicit set of processors using relative weights for the dimensions. The language allows an extensive set of alignments along with simple regular and irregular distributions.

Vienna Fortran [5] is the first language to provide a complete specification of mapping constructs in the context of Fortran. In addition to simple regular and irregular distributions, Vienna Fortran defines a generalized block distribution which allows arbitrarily sized contiguous segments of data to be mapped to the processors. The language also proposes a mechanism for user-defined distribution and alignment functions, and defines multiple methods of passing distributed data across procedure boundaries.

HPF-2 [14] defines a set of directives for Fortran 95 largely based on previous work in Fortran D and Vienna Fortran. It provides significant support for irregular distributions (including *general block* and *indirect* mappings) as well as the possibility to map pointers, components of derived types, and objects to subsets of processors directly.

Newer developments, such as the *global array languages* Coarray Fortran, UPC, and Titanium take a reasonable intermediate approach, providing a higher level of abstraction than MPI but dealing only with standard distributions and requiring explicit control of communication.

The ZPL language supports a concept of dimensional distributions which are organized into five types, each of which has its own properties: block, cyclic, multi-block, non-dist, and irregular. These types give the compiler the information it needs to generate loop nests and communication, abstracting the details of the distribution from the compiler's knowledge. This strategy was detailed in [7], in which a few block distributions were implemented as a proof-of-concept.

4.2 Distribution Support in Object-Based Systems

Hawk [13] is a system based on ORCA [2, 1] that has the notion of partitioned objects for supporting regular, data parallel applications. There is one thread of control per data access. In this Distributed Shared Memory (DSM) software implementation the entire data is replicated on each address space and only parts of it are truly owned. The parts that are not owned are invalid and updated by a consistency protocol. Due to the replication strategy, the system is inefficient for intensive data applications.

EPEE [16, 21] is an Eiffel parallel execution environment aimed at offering a high level API developer a platform for incorporating new components as common behavioral patterns are detected. The environment provides the programmer with a set of classes for handling data distribution issues. For instance, mechanisms for distributing bi-indexable objects (e.g., arrays, grids, matrices, tables) based on a block-wise partitioning are encapsulated in a class Distribution 2D. This idea is similar to the approach taken by

CO_2P_3S [18]. This is a system that allows the user to specify parallel design patterns and, based on the specification, generates data parallel programs, including communication and synchronization code. The effort required to write such patterns may be significant, but once written, they can be reused.

Charm++ [17] is a concurrent object-oriented system based on C++. Parallelism is explicitly expressed as an extension to the C++ language. Parallel processes (chares) communicate through message objects that are explicitly packed/unpacked by user. The system also features special shared objects and remote accesses through remote procedure calls. Thus, communication and synchronization may be controlled by the programmer. Synchronization is ensured for shared objects.

ICC++ [6] is a C++ dialect for high performance parallel computing. Data collections are represented as arrays encapsulated within objects. Distribution can be explicitly specified by overloading the access `[]` operator to a collection object. Irregular distributions can also be manually specified by supplying a map file which is a sequence of integer indices along with virtual processor numbers. created for large

pC++ [3] is an object parallel language based on C++ and HPF-like. Unlike templates in HPF, distributions in pC++ are first class objects. A `Distribution` is characterized by its number of dimensions, the size in each dimension and the function by which the distribution is mapped to processors. Current distribution functions allowed in pC++ include BLOCK, CYCLIC, and WHOLE.

The Mentat [12] system provides data-driven support for object-oriented programming. The idea is to support a data-flow graph computation model in which nodes are actors and arcs are data dependences. The programmer must specify the classes whose member functions are sufficiently computationally complex to warrant parallel execution. The data-flow model is enhanced to support larger granularity and a dynamic topology. Parallelism is supported through having multiple actors executing on multiple processors.

HPC++ [11] is based on PTSL (Parallel Standard Template Library), a parallel extension of STL, Java style thread class for shared-memory architectures, and HPF like directives for loop level parallelism. A context is a virtual address space on a node. Parallelism within a context is loop level parallelism. Parallelism across multiple contexts allows one thread of execution on each context. Low level synchronization primitives (including semaphores and barriers) coexist with high level collections and iterators.

Distributed recursive sets [8] are nested data collections that allow expressing complex data structures in data parallel applications. The collections are automatically distributed by a system using a graph-based partitioning scheme.

There are two major developments in Chapel that distinguish it from previous work:

1. The generalization of the array concept and the ability to define domains indexed by arbitrary primitive and user-defined data types.
2. The generality of the distribution machinery which is a combination of system-supported and user-defined distributions. The distribution framework in Chapel allows user access to distribution decisions by letting the user define novel distributions based on a system provided distribution type.

5. CONCLUSION

This paper presented a highly reusable data distribution design for data intensive, high performance applications. The design can be used as-is by application writers, object-oriented frameworks and skeleton writers, and generally, high-level languages and tools writers. The elements of the distribution design pattern are novel concepts introduced by the Chapel high productivity language.

We showed how domains and their index sets allow the construction of complex data structures, with index types including a variety of primitive or user-defined types. Further, we showed how data collections can be associated with domains, inheriting their index set and its distribution.

We believe that our design balances the system support and user control over distributions having the potential of delivering both promises of high productivity and performance guarantees. As our Chapel compiler and run-time infrastructure evolves, we hope to provide empirical evidence on these aspects.

6. REFERENCES

[1] H. E. Bal and M. F. Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In A. Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, volume 28 of *SIGPLAN Notices*, pages 162–177, New York, NY, 1993. ACM Press.

[2] H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum, and J. Jansen. Replication Techniques for Speeding up Parallel

Applications on Distributed Systems. *Concurrency Practice and Experience*, 4(5):337–355, August 1992.

[3] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[4] D. Callahan, B. Chamberlain, and H. 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.

[5] B. M. Chapman, P. Mehrotra, and H. P. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[6] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ — A C++ dialect for high performance parallel computing. *Springer LNCS*, 1049:76–94, 1996.

[7] S. J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD thesis, University of Washington, 2004.

[8] R. E. Diaconescu. *Object Based Concurrency for Data Parallel Applications: Programmability and Effectiveness*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, August 2002. [NTNU 2002:830, IDI Report 9/02, ISBN 82-471-5483-8, ISSN 0809-103X].

[9] G. Fox, Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., and M.-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Houston, TX, December 1990.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing. Addison-Wesley, 18th printing edition, September 1999.

[11] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine. HPC++ and the HPC++Lib Toolkit. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 73–108, 2001.

[12] A. S. Grimshaw. The mentat computation model data-driven support for object-oriented parallel processing. Technical Report CS-93-30, University of Virginia, May 1993.

[13] S. B. Hassen, I. Athanasiu, and H. E. Bal. A flexible operation execution model for shared distributed objects. In *Proceedings of the OOPSLA'96 Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 30–50. ACM, October 1996.

[14] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Technical report, Jan. 1997.

[15] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling fortran d for mimd distributed-memory machines. *Commun. ACM*, 35(8):66–80, 1992.

[16] J.-M. Jzquel. An object-oriented framework for data parallelism. *ACM Computing Surveys*, 32(31):31–35, March 2000.

[17] L. V. Kale and S. 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.

[18] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Comput.*, 28(12):1663–1683, 2002.

[19] P. Mehrotra and J. V. Rosendale. Programming distributed memory architectures using Kali. In *Advances in Languages and Compilers for Parallel Computing*. MIT Press, 1991.

[20] R. E. Millstein. Control structures in illiac iv fortran. *Commun. ACM*, 16(10):621–627, 1973.

[21] N. Sato, S. Matsuoka, J.-M. Jezequel, and A. Yonezawa. A methodology for specifying data distribution using only standard object-oriented features. In *Proc. of International Conference on Supercomputing*, pages 116–123. ACM, 1997.

[22] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.