User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*

Bradford L. Chamberlain Steven J. Deitz David Iten Sung-Eun Choi
Cray Inc., 901 Fifth Avenue, Suite 1000, Seattle WA 98164
chapel_info@cray.com

Abstract

This paper introduces user-defined domain maps, a novel concept for implementing distributions and memory layouts for parallel data aggregates. Our domain maps implement parallel arrays for the Chapel programming language and are themselves implemented using standard Chapel features. Domain maps export a functional interface that our compiler targets as it maps from a user's global-view algorithm down to the task-level operations required to implement the computation for multicore processors, GPUs, and distributed memory architectures. Unlike distributions in HPF and ZPL, Chapel's domain maps are designed for generality and do not rely on hard-coding a fixed set of distributions into the compiler and runtime. The chief contributions of this paper are its description of our motivating principles and an overview of our framework.

1 Introduction

Chapel is a novel language being developed by Cray Inc. to support general, productive parallel programming. Chapel strives to be general by supporting arbitrary parallel algorithms on diverse parallel architectures. In terms of productivity, it aims to improve the programmability of large-scale parallel systems while matching or beating the performance and portability achieved by current programming models like MPI and OpenMP. Chapel is being developed in an open-source manner and the concepts described in this paper can be used in their current form by downloading Chapel [7].

One of Chapel's most promising concepts for improving parallel programmability is its support for *global-view arrays*—data aggregates whose elements may be stored using distinct memories, yet whose indices are expressed *in toto* using a single logical index set. While existing languages like High Performance Fortran (HPF), ZPL, and UPC also support global-view arrays [20, 4, 15], each supports at most a handful of distributions whose semantics are built into their compilers and runtimes. This helps with productivity when the built-in distributions suit a user's needs, but constitutes a linguistic dead-end when they do not.

In contrast, our work strives to support *user-defined* distributions that permit advanced users to specify their own global-view array implementations. We achieve this using a concept called a *domain map* which serves as a recipe for implementing parallel arrays. We refer to domain maps that target a single shared memory segment as *layouts*. Domain maps that target multiple distinct memories, as on large-scale distributed memory systems or heterogeneous node architectures, are called *distributions*. In both cases, domain maps permit the Chapel compiler to lower a user's high-level code down to the fragmented, per-node data structures and tasks that implement its parallel operations.

This paper represents the first published description of Chapel's strategy for user-defined domain maps. Previous work proposed a possible approach to distributions in Chapel [13], yet that proposal was never implemented and, in our opinion, was too minimalist in its design to support Chapel's performance goals. Our two approaches share some characteristics due to our common foundations [12, 25] and early collaborations. To our knowledge, this paper represents the first time that a programming language has supported natively-specified user-defined distributions via a functional interface.

This paper is organized as follows: The next section provides a brief overview of Chapel as background. Section 3 contrasts our work with the most relevant previous work. Section 4 outlines our philosophy and goals for user-defined domain maps in Chapel while Section 5 describes a number of motivating domain maps and their current status. Our software framework for domain maps is introduced in Section 6. Section 7 wraps up by summarizing and outlining our next steps.

2 Chapel Overview

This section provides a brief overview of Chapel for the purposes of understanding this paper. For more information, please refer to Chapel's documentation [5, 10, 8].

Chapel's features are organized in distinct layers, where higher-level concepts are built upon lower-level ones. For example, Chapel's domain maps are its highest-level concept, implemented in terms of lower-level features for expressing task parallelism and locality. Unlike HPF, Chapel's users can shed its high-level data parallel features and program at lower levels to obtain more explicit control over the computation. We refer to this as a *multiresolution language design*.

The lowest level of Chapel's feature set is a rich base language with support for compile-time computa-

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

tion, range and tuple types, objects, function overloading, CLU-style iterator functions [21], static type inference, and generic programming. By and large, this set of features was selected in order to provide good support for specifying user-defined domain maps in Chapel.

Chapel's locality-oriented features permit the programmer to refer to architectural regions of locality via an abstract *locale* type. For most parallel platforms, the Chapel compiler maps locales to multicore or SMP nodes. A Chapel program can refer to the set of locales on which it is executing using a built-in array of locale values. Programmers specify the number of locales at program execution time via command-line flags.

Chapel's data parallel features are based on a first-class representation of an index set called a *domain*. Domains are used to describe iteration spaces and to declare arrays. Chapel's domains are a generalization of the *region* concept pioneered by the ZPL language [4], supporting dense, sparse, strided, associative, and unstructured data aggregates. Domains support iteration, intersection, set-oriented queries, and operations for creating other domains. They are also used to declare, slice, and reallocate arrays. Chapel arrays are one-to-one mappings from a domain's indices to a set of variables of arbitrary but homogeneous type.

Each Chapel domain—and by extension its arrays—has an associated domain map which defines its implementation. Domain maps specify how domain indices and array elements are mapped to locales, how they are stored in memory, and how operations such as accesses, iteration, and slicing are implemented. If no domain map is explicitly specified by the user, the compiler uses a default domain map that targets a single locale, implementing parallel operations using its processor cores.

When multiple domains share a single domain map, they are considered to be *aligned* since a given index will map to the same locale within each domain. One of the benefits of exposing domains and domain maps as first-class language concepts in Chapel is that overheads associated with implementing arrays can be amortized across multiple aligned arrays and domains. In addition, domain maps improve the ability for compilers and users to reason about the semantics of parallel programs.

The following Chapel code declares a domain map *myDist* that is an instance of the distribution *DistClass*. This domain map is used to declare two aligned 1-dimensional domains, *D1* and *D2*, each of which is then used to declare a pair of arrays:

```
const myDist = new dmap(new DistClass(...));

const D1: domain(1) dmapped myDist = [1..m],
      D2: domain(1) dmapped myDist = [0..m+1];

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

3 Related Work

As mentioned previously, the most important predecessors for our work are the High Performance Fortran family of languages [19, 20, 16, 9, 1] and ZPL [22]. HPF supports arrays whose dimensions can each be distributed using regular block, cyclic, and block-cyclic schemes. HPF-2 added support for indirect distributions that could support arbitrary mappings of data to processors [25, 17], while other efforts proposed support for distributed compressed sparse row (CSR) arrays by having the programmer write code in terms of distributed versions of the underlying data structures [23].

ZPL is more closely related to our work due to its support for first-class index sets known as *regions* [4], designed to help the compiler and user reason about distributed array semantics. For most of its lifetime, ZPL only supported block distributions. Late in the ZPL project, a lattice of distribution types was designed to extend ZPL's generality [12]. However, only a few of these distributions were ever implemented.

ZPL and HPF implementations have relied on building the semantics of their distributions into compilers and runtime libraries. In contrast, our approach only requires the implementation to know about general operations that a domain map can support—like iteration or slicing—rather than any specific knowledge about its semantics—such as which locales will own neighboring elements in a block distribution. Moreover, we believe that our approach provides users with the ability to specify arbitrary distributions more efficiently in time and space than previous efforts like HPF-2's indirect distribution. We also assert that our framework can support more general distributions than previous approaches.

4 Chapel Domain Map Design Goals

In previous sections, we described the basic role of user-defined domain maps in Chapel. Here, we provide a number of additional goals and themes that have guided our design.

Support for General Array Formats A primary concern in our philosophy is that developers should be able to describe any reasonable array representation that they can envision. In particular, the domain map framework strives to avoid imposing arbitrary constraints on the author due to our potential shortsightedness. In carving out our design space we wanted to make sure to support a framework that was rich enough to support not only standard distributions as in HPF and ZPL, but also richer mappings such as various sparse matrix formats, recursive bisections, multidimensional multipartitionings [11], hierarchically tiled arrays [2], Morton-ordered arrays [24], graph partitioned data structures [18, 3], or any other mapping of data elements to locale memories that a computation may require.

Specification via a Functional Interface Due to our desire to support very general array implementations, we define our domain maps via an object-oriented interface in order to say as little as possible about how they should be implemented, focusing instead on the operations that they must support. Section 6 discusses this functional interface in more detail.

Multidimensional Distributions Unlike HPF in which each dimension of an array is distributed independently, in our work we choose to distribute multidimensional index sets holistically in order to support distributions like recursive bisection that cannot be defined as a composition of dimensional distributions. Our approach subsumes the per-dimension approach because multidimensional distributions can be defined that are themselves parameterized by a list specifying how each dimension should be partitioned.

Target Locale Sets In order to support coarse-grain task parallelism and coupled computation models, Chapel's distributions typically take a target set of locales to which the domain indices and array elements are mapped. By convention, locales that are not part of the target set do not store data for that distribution, and parallel loops over the distribution's domains and arrays will only utilize processors in the target locale set. In this way, distributions can target disjoint or overlapping sets of machine resources. This flexibility is enabled in part by Chapel's support for dynamic and nested parallelism rather than being limited to single-threaded SPMD execution models.

Plug-and-play Chapel's semantics for domain and array operations are intended to be independent of the domain maps used to implement them. To this end, changing a domain's domain map should only impact the implementation and performance of a program, not the correctness of its execution. This permits users to first develop and debug a single-locale program before incrementally adding support for multi-locale execution and performance optimizations simply by changing the domain maps used to declare their domains.

Separation of Roles Chapel domain maps are intended to separate the use of high-level parallel array operations from the low-level details required to implement global-view semantics on large-scale distributed-memory systems. In this sense, the clients of domain maps are intended to be applications programmers who understand the concept of using parallel operators on arrays, yet who need not be well-versed in the details of implementing parallel data structures. Such skills *are* required by authors of domain maps. This separation of concerns is designed to permit computational scientists and parallel computing experts to each focus on their area of expertise without unnecessarily intertwining their code within a Chapel program.

Domain Map Libraries Supporting libraries of domain maps will be important in order to reuse code across projects and reduce the need for users to develop domain maps from scratch. Chapel is designed to include a standard library of domain maps to support common data layouts and distributions. Like any standard library, the implementations of these domain maps can vary from one target platform to the next in order to tune for each system's capabilities. Additionally, we envision there being open-source community repositories for Chapel domain maps so that developers can share their code with the community and benefit other users.

Performance Chapel's multiresolution design is not meant to suggest that using higher-level features will necessarily result in a sacrifice of performance. While the use of lower-level features will always admit performance improvements through increased effort, our domain map framework is designed to result in good performance for global-view data structures. Our performance results in ZPL competed with and outperformed hand-coded Fortran+MPI [4, 12], giving us reason to believe that well-designed domain maps can support global-view abstractions without sacrificing performance. Our initial scalability results for the Chapel block distribution also support this belief, achieving 10.8 TB/s and 0.122 GUPs for two of the HPC Challenge benchmarks on 2048 quad-core processors [6, 14]. We are implementing Chapel's standard domain maps using the same mechanisms that an end-user would to ensure that our design does not result in a performance cliff between "built-in" domain maps and those defined by an end-user.

Parameterization of Domain Maps Chapel domain maps are parameterizable so that a single domain map implementation can be used in multiple contexts. Aspects of this parameterization may be encoded in the domain map for convenience and portability—for example, a domain map might query a locale's cache line sizes or number of processor cores in order to optimize loop structure. In addition, domain maps may support user-supplied arguments so that clients can manually tune the way a domain map is implemented rather than requiring a distinct domain map for each potential context.

Implemented Using Lower-level Features In the spirit of Chapel's multiresolution design, domain maps are authored using lower-level concepts in Chapel. For example, rather than introducing new concepts for generating parallelism within a program, domain maps are implemented in terms of Chapel's standard features for task parallelism and locality. As mentioned previously, most of Chapel's base language features are very useful for managing the complexity of implementing domain maps, including its support for OOP, generic programming, type inference, tuples, iterators, overloading, and the compile-time language.

Compiler-Known Concept While a good software engineer could implement domain maps like ours in an existing object-oriented language using a similar framework, we believe that it is crucial for global-view languages to support domain maps as a language concept in order to support compiler optimization of parallel operations, particularly those utilizing multiple arrays or domains within a single statement. As we will describe in Section 6, the Chapel compiler's role is not simply to manually rewrite global-view operations down to per-locale operations, but also to inspect a domain map's supported interface in order to perform optimizations that take advantage of its strengths.

Transparent Execution Models One of the common concerns in using global-view abstractions is that the high-level nature of the data structures and operations will prevent a performance-minded programmer from being able to reason about how their code will execute. For this reason, it is crucial that domain maps be well-documented so that the client can understand how indices are mapped to locales, how data is stored within a locale, how much parallelism is used for loops over a domain or array, and how communication is implemented. Without such information, a domain map's client will have a more difficult time obtaining good performance from their high-level code.

5 Sample Domain Maps and Status

In this section, we provide a survey of domain maps that we intend to support in Chapel using our framework. We start with simpler and more common domain maps and work our way toward cases that stretch the conventional notion of what a distribution is. We describe the current status of our implementation as we go.

Standard Layouts Chapel currently supplies a number of standard layouts in order to support diverse data representations within a locale's memory. Chapel's default layouts support dense row-major order arrays, sparse coordinate storage, and open-address hashing using quadratic probing. We have also developed standard layouts that implement column-major storage order and compressed sparse row (CSR) format. These layouts utilize runtime heuristics and user-supplied arguments to determine the number of tasks to create for each parallel loop. In several cases they already result in performance competitive with hand-coded C. These single-locale domain maps are not only useful for implementing parallel arrays on multicore processors, they also serve as building blocks for each locale's contribution to a multi-locale distribution. Looking ahead, we are interested in implementing hierarchically-tiled arrays, Morton-order arrays, and diverse implementations of sparse, associative, and unstructured domains and arrays.

Regular Distributions Chapel's standard domain map library will support common regular distributions such as block, cyclic, and block-cyclic. Our current block and cyclic distributions are feature-complete and our block-cyclic distribution is under development. We ultimately expect to support several variations of block and block-cyclic distributions in order to support interoperability with existing libraries and languages. As mentioned earlier, we also plan to implement dimensional distributions parameterized by per-dimension partitioning rules.

Irregular Distributions All of the distributions described so far divide an array's dimensions between a set of locales using regularly-spaced hyperplanes to partition complete dimensions. Our framework also supports the ability to define irregular distributions such as recursive bisections to distribute an index space in a data-sensitive way for improved load balance. Another example that we have started implementing is support for distributed hash tables in which elements are first hashed to a locale and then stored locally using a standard associative domain. For sparse and unstructured computations, we plan to support distributions that utilize standard graph partitioning techniques including geometric algorithms, spectral methods, and multilevel partitioning algorithms [18, 3].

Accelerator-based Domain Maps In collaboration with UIUC, we have recently been exploring the use of Chapel domain maps that target hardware accelerators. We have prototyped layouts that target NVIDIA GPUs by generating CUDA to implement array allocations, data transfers, and parallel operations. Our prototype work is promisingly competitive with hand-coded CUDA and has the benefit of providing users with a unified set of concepts for targeting CPUs and GPUs rather than relying on hybrid programming models.

Dynamically Load-Balanced Domain Maps Distributions are typically defined as a static mapping of indices or array elements to machine resources. However, nothing about our framework requires domain maps to be so static in nature. In particular, we are exploring the use of domain maps to implement parallel loops using dynamic work distribution patterns like master-worker or work-stealing. A user's parallel loops would appear unchanged while the underlying implementation would be far more dynamic than a traditional distribution.

Domain Maps for Resiliency Another non-standard use of our framework would be to support resilient computations by having a domain map utilize redundancy in storage and/or computation in order to tolerate hardware failures. For example, clients could specify the degree of redundancy that they are willing to invest, directing the domain map to store each array element or perform each loop iteration redundantly the specified number of times.

Domain Descriptors:

- Create new arrays
- Support whole-domain assignment of index sets
- Modify the domain's index set by adding, removing, and clearing indices (irregular domains only)
- Compute index set intersections and ordering
- Iterate over the index set sequentially, in parallel, and in a zippered manner with other iterable expressions
- Query index set size and membership

Array descriptors:

- Randomly access array elements
- Reallocate the array's data when its domain is modified
- Create array aliases to support slicing, reindexing, and rank change (rectangular index sets only for the last two)
- Get and set sparse "zero" values (sparse arrays only)
- Iterate over elements sequentially, in parallel, and in a zippered manner with other iterable expressions

Figure 1: Summary of the Required Interface for Chapel's Global Domain and Array Descriptors

6 Domain Map Framework

This section describes the software framework that we are using to implement domain maps. Due to space limitations, this description is necessarily high-level.

Chapel's domain maps are implemented using three descriptor classes: one to represent the domain map itself, one to represent its domains, and one for its arrays. These classes are called *global descriptors*, and they are instantiated for each domain map, domain, or array that the program creates. Multi-locale distributions typically use a second set of *local descriptors* to represent the indices and elements owned by a single locale.

Domain map authors define their descriptors by subclassing abstract base classes defined by Chapel's standard modules. The Chapel compiler makes no assumptions about how a domain map's descriptors store indices and data values; it only interacts with the classes through their functional interfaces. Descriptors are typically linked together so that domains can refer to their domain maps, arrays to their domains, and vice-versa. In addition, global descriptors need a way to find their constituent local descriptors in order to propagate method calls to the objects owning the indices in question.

Domain map descriptors support three interface types: a required interface, optional interfaces, and custom interfaces. Each is described below.

Required Interface The required interface is sufficient to support all of Chapel's operations on domains and arrays, though potentially in a suboptimal manner. If this interface is not implemented, the compiler cannot lower Chapel's global-view operations to the lower-level computations required to implement them.

The required interface for the global domain map descriptor is minimal: it must create new domains and, in the case of distributions, map from indices to locales. Figure 1 summarizes the required interfaces for the global domain and array descriptors.

Optional Interfaces Chapel domain maps can also implement a wide number of optional sub-interfaces that enable optimizations beyond what the compiler can achieve with the required interface alone. In lowering global-view code, the compiler looks for opportunities to profitably utilize these optional interfaces. When such opportunities exist, the participating domain maps are inspected to see whether the necessary optional interfaces are supported. If they are, the compiler applies the optimization to improve the quality of the generated code.

We have not yet done much optimization of domain maps and therefore have not defined many optional interfaces to date. We currently support one to replicate global descriptors and a second that supports optimized iteration for aligned domains. In the future we anticipate adding interfaces for structured communication idioms and efficient block copies of sub-array assignments.

Custom Interfaces The third class of domain map interface is one that is not known to the Chapel compiler. Since users are permitted to author their own domain maps, it makes sense to give them the ability to define and directly invoke methods on their domain maps, domains, and arrays. Such custom interfaces provide the user with a finer level of control and the potential for better performance. However, they also have the disadvantage that since they will not be supported by most domain maps, their use violates the plug-and-play goal of our approach, making a user's code more brittle.

7 Summary

In this paper, we have motivated, described, and reported on a unique framework for implementing user-defined layouts and distributions for global-view parallel computation. While our use of domain maps has been developed primarily for large-scale parallel computing, the mechanism is also general enough to support memory layouts and parallelization strategies for multicore processors and for heterogeneous nodes consisting of traditional processors coupled with GPU accelerators.

Next Steps The approach described in this paper is very much a work-in-progress, yet our experiences to date give us confidence that we are on a constructive path. Our next steps are to continue writing increasingly advanced domain maps while continuing to improve the performance of our current set. Longer-term, we are interested in exploring the role of domain maps in interoperability and on emerging exascale architectures.

Acknowledgments

The authors would like to thank David Callahan, Hans Zima, and Roxana Diaconescu for their early contributions to Chapel and their role in helping refine our user-defined domain map philosophy. We would also like to acknowledge David Bernholdt, Wael Elwasif, Robert Harrison, Jim Kohn, John Lewis, Albert Sidelnik, and David Wise for interesting discussions that encouraged us to pursue some of the more exotic domain map concepts described in this paper. Finally, we would like to thank all current and past Chapel contributors and users for their help in developing the language foundations that have permitted us to reach this stage.

References

[1] ALBERT, E., KNOBE, K., LUKAS, J. D., AND STEELE, JR., G. L. 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* (1988), ACM Press, pp. 42–56.

[2] BIKSHANDI, G., GUO, J., HOEFLINGER, D., ALMASI, G., FRAGUELA, B. B., GARZARAN, M., PADUA, D., AND VON PRAUN, C. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of parallel programming* (March 2006), ACM Press, pp. 48–57.

[3] BOMAN, E., DEVINE, K., HEAPHY, R., HENDICKSON, B., LEUNG, V., RIESEN, L., VAUGHAN, C., CATALYUREK, U., BOZDAG, D., MITCHELL, W., AND TERESCO, J. Zoltan v3: Parallel partitioning, load balancing and data-management services, users guide. Tech. Rep. SAND2006-4748W, Sandia National Laboratories, Albuquerque, NM, 2007.

[4] CHAMBERLAIN, B. L. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[5] CHAMBERLAIN, B. L., CALLAHAN, D., AND ZIMA, H. P. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications 21*, 3 (August 2007), 291–312.

[6] CHAMBERLAIN, B. L., CHOI, S.-E., DEITZ, S. J., AND ITEN, D. HPC Challenge benchmarks in Chapel. (available at http://chapel.cray.com), November 2009.

[7] Chapel source repository and website for downloading releases. http://sourceforge.net/projects/chapel.

[8] Chapel website. http://chapel.cray.com.

[9] CHAPMAN, B. M., MEHROTRA, P., AND ZIMA, H. P. Programming in Vienna Fortran. *Scientific Programming 1*, 1 (1992), 31–50.

[10] CRAY INC. *Chapel Specification*, 0.795 ed. Seattle, WA, April 2010. (http://chapel.cray.com).

[11] DARTE, A., MELLOR-CRUMMEY, J., FOWLER, R., AND CHAVARRÍA-MIRANDA, D. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *Journal of Parallel and Distributed Computing 63*, 9 (2003), 887–911.

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

[13] DIACONESCU, R., AND ZIMA, H. P. An approach to data distributions in Chapel. *International Journal of High Performance Computing Applications 21*, 3 (August 2007), 313–335.

[14] DONGARRA, J., AND LUSZCZEK, P. Introduction to the HPC Challenge benchmark suite. Tech. Rep. ICL-UT-05-01, ICL, 2005.

[15] EL-GHAZAWI, T., CARLSON, W., STERLING, T., AND YELICK, K. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, June 2005.

[16] FOX, G., HIRANANDANI, S., KENNEDY, K., KOELBEL, C., KREMER, U., TSENG, C.-W., AND WU, M.-Y. Fortran D language specification. Tech. Rep. CRPC-TR 90079, Rice University, Center for Research on Parallel Computation, December 1990.

[17] HIGH PERFORMANCE FORTRAN FORUM. *High Performance Fortran Language Specification Version 2.0*, January 1997.

[18] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1999), 359–392.

[19] KOELBEL, C., AND MEHROTRA, P. Programming data parallel algorithms on distributed memory using Kali. In *ICS '91: Proceedings of the 5th international conference on Supercomputing* (1991), ACM, pp. 414–423.

[20] KOELBEL, C. H., LOVEMAN, D. B., SCHREIBER, R. S., GUY L. STEELE, JR., AND ZOSEL, M. E. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, September 1996.

[21] LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *ACM SIGPLAN 12*, 7 (July 1977), 75–81.

[22] SNYDER, L. *The ZPL Programmer's Guide*. Scientific and Engineering Computation. MIT Press, March 1999.

[23] UJALDON, M., ZAPATA, E. L., CHAPMAN, B. M., AND ZIMA, H. P. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems 8*, 10 (October 1997).

[24] WISE, D. S., FRENS, J. D., GU, Y., AND ALEXANDER, G. A. Language support for Morton-order matrices. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practice of parallel programming* (2001), ACM, pp. 24–33.

[25] ZIMA, H., BREZANY, P., CHAPMAN, B., MEHROTRA, P., AND SCHWALD, A. Vienna Fortran — a language specification version 1.1. Tech. Rep. NASA-CR-189629/ICASE-IR-21, Institute for Computer Applications in Science and Engineering, March 1992.