6 Chapel

Bradford L. Chamberlain, Cray Inc.

Chapel is an emerging parallel language designed for productive parallel computing at scale. Chapel originated as part of Cray Inc.'s participation in the DARPA High Productivity Computing Systems (HPCS) program, which ran from 2002–2012. At present, Chapel development is focused on transforming the research prototype produced under HPCS into a production-grade implementation. Cray leads the effort of designing and developing Chapel, in collaboration with members of the research and open-source communities.

Chapel supports a multithreaded execution model, permitting the expression of far more general and dynamic styles of computation than the typical single-threaded Single Program, Multiple Data (SPMD) programming models that became dominant in the 1990's. Chapel is designed such that higher-level abstractions, such as those supporting data parallelism, can be built in terms of lower-level concepts in the language, permitting the user to select between various levels of abstraction or control as necessitated by their algorithm or its performance requirements.

This chapter provides a brief introduction to Chapel, starting with a condensed history of the project (Section 6.1). It then describes Chapel's motivating themes (Section 6.2), followed by a survey of its main features (Section 6.3), and a summary of the project's status and future work (Chapter 6.4).

6.1 A Brief History of Chapel

6.1.1 Inception

DARPA's HPCS program was launched in 2002 with five teams, each led by a hardware vendor: Cray Inc., Hewlett-Packard, IBM, SGI, and Sun. The program challenged the teams to develop technologies that would improve the productivity of HPC users in terms of performance, portability, programmability, and robustness. The vendors were encouraged to reconsider all aspects of their system stack with the goal of delivering technologies that would be revolutionary and distinct from their established roadmap. Along with changes to their processor, memory, and network architectures, the vendor teams also proposed new and enhanced software technologies, including novel programming languages.

In 2003, the HPCS program transitioned to phase II, and a programmatic downselect occurred, enabling the Cray, IBM, and Sun teams to pursue their proposed research plans. At the outset of this phase, the initial designs of the new programming languages began to emerge, with the Cray team pursuing the Chapel language, IBM starting work on X10 [76, 246], and Sun (later acquired by Oracle) developing Fortress [10].

Cray's HPCS project was named *Cascade* after the prominent mountain range just east of its corporate headquarters in Seattle. The project was led by Burton Smith, Chief Sci-

entist of Cray at the time. Though he believed that existing HPC programming models were a productivity limiter for high-end systems, Burton was initially hesitant to pursue a new programming language under HPCS, due to skepticism about whether languages designed by lone hardware vendors could be successful. He soon reconsidered this position, however, after an enumeration of well-established programming languages in both HPC and mainstream computing revealed that most of them had originally been developed by a single hardware vendor. In most cases, the key to a language's long-term success involved a transition to a broader, more community-oriented model at an appropriate point in its life cycle. In January–February 2003, the Cascade team announced its intention to pursue a new language at various HPCS reviews and meetings. Work on Chapel began in earnest that year under the leadership of David Callahan.

The Chapel language took its name as an approximate acronym for *Cascade High Productivity Language*, coined by Callahan. The team generally felt lukewarm-to-negative about the name, in large part due to its possible religious implications. However, nobody came up with a preferable alternative quickly enough, and the name stuck. When asked about it, team members would occasionally quip, "We'll wait until we've gotten the language to a point that we're thoroughly happy with it and then switch to a truly great name."

6.1.2 Initial Directions

Chapel's initial design was shaped primarily by four people who set the language on the path that it continues to follow today: David Callahan, its chief architect from Cray Inc.; Hans Zima, an academic partner within the Cascade program representing CalTech/JPL; Brad Chamberlain, a recent graduate from the ZPL project at the University of Washington; and John Plevyak, an independent contractor who joined the Chapel project in late 2003, bringing with him a strong background in iterative flow analysis and type inference [233].

To a great extent, Chapel's feature set reflects a combination of the backgrounds of these four initial architects: David Callahan established the overall vision for the language and, from his extensive experience with the Tera MTA (Multi-Threaded Architecture), brought the notion of a general, multithreaded execution model with lightweight, data-centric synchronization [11]. Hans Zima was a founding contributor to the High Performance Fortran (HPF) language in the 1990s, and brought with him the lessons learned from that high-profile endeavor [161]. Brad Chamberlain's dissertation focused on the benefits of supporting first-class index set concepts in parallel languages [65], so he contributed an alternative model for data parallelism with the goal of improving upon the array abstractions supported by HPF and ZPL. And finally, John Plevyak's experience filled an expertise gap in the group that Callahan correctly believed would be crucial for the language's

success—supporting static type inference with the goal of making parallel programming more productive, particularly as a means of supporting generic functions and classes.

With this combined background, the initial team set off to define Chapel. Much of the early brainstorming was guided by explorations of parallel computations that had posed challenges for previous parallel languages. Examples included finite element methods, fast multipole methods, connected components algorithms, n-body simulations, and the like. Draft codes and documents to explain them were circulated, punctuated by marathon design summits where the virtues and flaws of various options were debated, often passionately and at length. In the fall of 2004, David Callahan took on the task of writing an initial draft of the language specification, which served as a straw man for subsequent debate, refinement, and implementation. With that, the Chapel project was off and running.

6.1.3 Phases of Development under HPCS

The Chapel project's history during the course of HPCS can be thought of as falling into three distinct periods: For the first period, from 2003 to early 2006, the project was in a molten state, with team members splashing around and trying to construct a common vision of the language that they could all agree upon and conceivably implement. This period saw the first publication describing Chapel [57], as well as the formation of the initial development team who would get Chapel up and running.

The second period, from 2006–2008, marks the timeframe in which both Chapel's design and the compiler architecture began stabilizing [66], permitting a number of milestones to be achieved at regular intervals: In April 2006, task-parallel Chapel codes were run for the first time. In December 2006, the first release was made available to external users and evaluation teams on a by-request basis. July 2007 saw the execution of the first distributed-memory task-parallel programs. In June 2008, the first data-parallel constructs started working, and by September 2008, the first distributed-memory data-parallel codes were executing. During this period the core Chapel development team at Cray kept their heads tucked down, to move the implementation along as far and fast as possible.

The third period, from 2008–2012, constitutes the time when the Chapel team began to increasingly look outward in an attempt to attract users to Chapel and get feedback on its design and implementation. During these years, Chapel moved to a SourceForge-based open-source control repository, switched to a public release mechanism, started supporting early users, and established a number of collaborations with academics, lab staff, and members of industry outside of Cray. The Chapel team also stepped up its level of outreach during this period, particularly in terms of giving tutorials on Chapel in forums like the annual SC conference series and PRACE community events. All the while, improvements

were made to the implementation to flesh out missing features and boost performance in order to make progress and retain the interest of early users.

6.1.4 Life After HPCS

At the time of publication, the Chapel project remains an active and ongoing effort. By the end of the HPCS program in late 2012, Chapel had successfully achieved its programmatic requirements and, more importantly, had sparked substantial interest among HPC users. As a result, the Cray team embarked on a five-year effort to improve Chapel from the research prototype that was developed under HPCS to a production-grade implementation [69]. The goals of this effort are: to improve Chapel's performance and scalability; to address immature aspects of the language; to port Chapel to emerging node architectures involving deeper memory hierarchies and heterogeneous processors; to improve its interoperability features; to nurture the Chapel user and developer communities; and to explore the transition of Chapel's governance to a neutral body external to Cray.

With this overview of Chapel's history in mind, we now move on to describe some of the motivating themes and concerns that helped shape Chapel's design.

6.2 Chapel's Motivating Themes

To understand Chapel's features, it can be helpful to understand the themes that influenced what would or would not be included in the language. In this section, we provide an overview of these themes to establish a framework for the language features described in Section 6.3.

6.2.1 Express General Parallelism

One of the first and most important themes in Chapel is the concept of supporting general parallel programming. In particular, Chapel's goal is to be a language in which users will never hit a point where they conclude "Well, that was fun while I was trying to do x and y; but now that I want to do z, I'll have to go back to using MPI," (or whatever technology they had been using). This approach is in strong contrast to the host of parallel languages from the 1990s that focused on a specific type of parallelism, to the exclusion of other styles—e.g., HPF and ZPL's overriding focus on data-parallelism to the detriment of task-parallelism and nested parallelism. Chapel's founders believed that while focusing on a single style of parallelism was a prudent approach for an academic project, for a new language to truly be adopted within a field as diverse as HPC, it had to support a wide variety of computational styles.

To this end, Chapel was designed with features to support data parallelism, cooperative task parallelism, and synchronization-based concurrent programming. In addition, these styles were designed so that they could be composed arbitrarily to support nested parallelism.

In addition to permitting diverse styles of parallelism, Chapel was also designed to support general granularities of parallelism, both in the user's program and the target hardware. In practice, algorithms tend to contain parallelism at multiple levels: computational models, function calls, loop iterations, and even individual statements or expressions. Meanwhile, modern hardware typically supports parallelism across multiple machines or cabinets, network nodes, and processor cores, as well as vector operations that support parallelism in the instruction set. Most conventional parallel programming models target only a subset of these software and hardware granularities, and often just one. As a result, programmers must use hybrid programming models that mix multiple concepts and notations in order to take full advantage of all available parallelism in their algorithm and hardware. For example, a parallel program wanting to take full advantage of a petascale system today might use MPI to express executable-level parallelism across the nodes, OpenMP (Chapter 12) to express loop or task parallelism across the processor cores, and CUDA (Chapter 15), OpenCL (Chapter 16), or OpenACC [1] to offload parallel kernels to an accelerator. In contrast, Chapel strives to support the expression of all parallelism in a user's program while targeting all available hardware parallelism with a single, unified set of language concepts.

6.2.2 Support a Multithreaded Execution Model

At the time of Chapel's inception, like today, most of the deployed distributed-memory programming languages and notations exposed programming and execution models based on single-threaded cooperating executables, with SPMD models as a particularly common case. The Chapel team attributes much of the lack of productivity and generality within HPC programming to this restriction since it forces users to take a process-centric view of their computation rather than describing and orchestrating the parallelism as a whole.

Chapel chose instead to adopt a multithreaded execution model in which each process will typically be composed of multiple threads. Users express parallelism within their programs in terms of *tasks* that represent units of computation that can, and should, execute in parallel. These tasks are then executed by the threads, potentially creating additional tasks, either within the same process or a remote one. The result is a programming and execution model that is far more dynamic and general than traditional single-threaded SPMD models.

6.2.3 Enable Global-View Programming

Another way in which Chapel diverges from most adopted HPC notations is by supporting what its designers refer to as a *global view* of data structures and control flow. The concept is to move away from requiring computations on distributed data structures, like arrays, to be expressed in terms of the subarrays owned by each process, as is typical in conventional approaches like MPI or Fortran 2008's *co-arrays*. Instead, the programmer can declare and compute on distributed data structures as they would for a completely local version. Such variables are referred to as *global-view data structures* since they can be declared using a global problem size and accessed using global indices rather than via per-processor extents and local indices. Since large-scale programs are almost always data-intensive, Chapel also supports a wide variety of global-view array types, including multidimensional rectangular arrays, sparse arrays, associative arrays, and unstructured arrays.

Chapel's *global view of control* refers to the fact that a Chapel program begins executing using a single task and then introduces parallelism through the use of additional language constructs. This is in contrast to SPMD programming models in which users write their program with the assumption that multiple copies of `main()` will execute simultaneously.

It is important to note that while previous languages, like HPF and ZPL, also supported global-view concepts, they did not provide a rich means of escaping these abstractions in order to exert more control over execution details. Recognizing that high-level abstraction is not ideal for every scenario, Chapel was designed such that users could selectively avoid using global-view abstractions and drop down to more explicit local-view constructs. As a specific example, Chapel programmers can choose to write traditional single-threaded SPMD programs using manually-distributed data structures, and even message passing, if they so choose. In summary, providing a global view of computation for programmability should not preclude the expression of more explicit parallelism, since having a greater degree of control can also play a crucial role in productivity.

6.2.4 Build on a Multiresolution Design

The previous section's notion of programming at higher levels of abstraction or greater degrees of control as needed is part of what Chapel refers to as its *multiresolution design philosophy*. The idea is to have the language support higher- and lower-level features, permitting the user to benefit from abstractions like global-view arrays when appropriate, while still being able to do more explicit low-level programming when desired or necessary. Moreover, Chapel was designed such that its higher-level abstractions are implemented in terms of the lower-level ones. This ensures that they are all compatible, permitting users to mix and match between different levels arbitrarily.

As a specific example, Chapel's data-parallel loops and global-view arrays are higher-level features that are implemented in terms of its lower-level features like task-parallelism and explicit control over locality. Another example of a multiresolution feature is Chapel's support for *hierarchical locale models*, which permit advanced users to specify how Chapel is mapped to a target architecture by creating classes representing the processors and memories of a system's compute nodes. The user implements tasking and memory interfaces for these classes which are targeted by the compiler.

6.2.5 Enable Control over Locality

Because Chapel was designed to execute on large-scale systems where locality and affinity are crucial for performance, locality is considered a core concept in Chapel along with parallelism. Chapel's locality features provide control over where data values are stored and where tasks execute so that users can ensure parallel computations execute near the variables they access, or vice-versa.

Chapel supports a Partitioned Global Address Space (PGAS) memory model [288] in which a user's code can refer to any lexically visible variable regardless of whether it is stored in a local or remote memory. If the variable is remote, the compiler and runtime are responsible for implementing the communication that is required to load or store the variable over the network. Users can reason about the location of a variable statically using Chapel semantics, or dynamically using a variety of execution-time queries.

Chapel supports expression of locality using distinct language concepts from those used to introduce parallelism. This contrasts sharply with SPMD programming models in which each copy of the executable serves as both the unit of parallelism and of locality for the program. By separating these concerns into two distinct feature sets, Chapel permits programmers to introduce additional parallelism within a single process, or to execute code on a distinct compute node without introducing parallelism. This orthogonal design results in a clean, natural separation between parallelism ("what should run simultaneously?") and locality ("where should it run?").

6.2.6 Support Data-Centric Synchronization

Another motivating theme in Chapel is the expression of synchronization in a data-centric manner. This has two primary benefits. The first is that by associating synchronization constructs with variables, the locality of the abstraction is well-defined since each variable has a specific location on the target machine. The second is that since most synchronization is designed to guard access to data structures or values, combining the synchronization constructs with the variables being accessed typically results in a more elegant expression of the algorithm.

6.2.7 Establish Roles for Users vs. the Compiler

Chapel has been designed so that the responsibility of identifying parallelism and managing locality rests on the user rather than the compiler. Although Chapel is often characterized (correctly) as being a large and feature-rich language, it was intentionally designed to avoid reliance on heroic compilation or promises that the compiler would automatically manage everything for the user. To this end, Chapel was designed to avoid relying on the compiler to introduce parallelism and manage locality. While nothing in the language precludes an aggressive compiler from performing such transformations automatically, such technology is not expected of a Chapel compiler (and conversely, identifying parallelism and locality *is* expected from users).

Owing to its PGAS nature, one of the main roles of the compiler (and runtime libraries) is to implement the global namespace in a manner that transparently and efficiently transfers data values between their stored location and the tasks that reference them. This communication management forms the Chapel compiler's biggest role, along with traditional compiler concerns of scalar code generation and optimization.

Chapel's multiresolution design also serves to distinguish between different user roles. For example, a parallel programming expert can work at lower levels of the language, implementing parallel loop schedules and distributed data structures that can then be used by an applied scientist who does not need to be exposed to all of the implementation details. Chapel's user-defined *domain maps* are a key example of this philosophy.

6.2.8 Close the Gap Between Mainstream and HPC Languages

When polling students about which programming languages they are most familiar and productive with, responses typically focus on modern languages like Python, Java, and Matlab, along with experience with C and C++ from those who have worked in more systems-oriented areas. Meanwhile, the HPC community uses Fortran, C, and C++ almost exclusively along with technologies like MPI and OpenMP with which most students have no experience. Part of Chapel's goal for improving productivity is to narrow this gap in order to make better use of the graduating workforce while also leveraging productivity advances enjoyed by mainstream programmers.

To this end, the Chapel design team selected features and themes in productive mainstream languages and sought ways of incorporating them into a language suitable for HPC programming. In doing so, the goal was to support features that would neither undermine the goal of scalable performance nor alienate traditional HPC programmers.

Chapel's type inference capability is an example of a mainstream language feature that was customized to support good performance. Chapel chose to support type inference in order to give users the ability to quickly prototype code and benefit from polymorphism

in a manner similar to scripting languages like Python and Matlab. However, unlike most scripting languages, Chapel's type inference is implemented in the compiler, resulting in a fixed static type for each variable in order to avoid the overheads of dynamic typing at execution time. The use of type inference is completely optional in Chapel so that programmers who prefer using explicitly-typed languages (e.g., for clarity or robustness in library interfaces) can still program in a more traditional style.

Chapel's object-oriented programming (OOP) capabilities are an example of a mainstream feature that was included in a manner that would keep it palatable to more traditional HPC programmers. In Chapel's early years, the design team spoke with HPC programmers who would offer opinions like "I'm simply not accustomed to using object-oriented features. If I had to rewrite my code in an object-oriented style it would create a lot of work for me because it's not how I've been trained to think." To this end, Chapel supports object-oriented features for all of the productivity and modularity benefits that they provide, yet intentionally avoids basing the language on a pure object-oriented paradigm (as with Smalltalk or Java) so that C and Fortran programmers can opt to ignore the OOP features and write more traditional block-structured imperative code.

6.2.9 Start From Scratch (but Strive for Familiarity)

The decision Chapel's designers made that has probably been called into question most often was the choice to design Chapel from a blank slate rather than as an extension to an existing language. There are many reasons why Chapel took this approach; perhaps the simplest is that all adopted languages carry with them a certain amount of baggage which reflects their original goals of supporting something other than general, large-scale parallel computing. As a result, most extension-based parallel languages tend to be a subset of a superset of a sequential language, making them incompatible with preexisting source code. Moreover, such languages still require a significant learning curve from users who must not only learn the new features that have been added, but also remember which ones have changed. The Chapel team's attitude is that the intellectual effort involved in learning a new parallel language stems primarily from learning the semantics of its parallel constructs, not their syntax. Thus, starting from scratch and designing features that express those new semantics as clearly as possible can have great value when compared to trying to force them into a language that was not originally designed with parallelism in mind.

That said, Chapel has also tried to avoid inventing concepts simply for the sake of it. In designing Chapel, the team studied successful (and unsuccessful) languages in order to learn from them, selecting features that would work well together. Chapel's primary influences include C, Modula, Fortran, C++, Java, C#, CLU [180], Scala [219], ML [134],

Perl, Matlab, ZPL [70, 255], HPF [161], and the Cray MTA™/XMT™extensions to C and Fortran [241].

Chapel's developers believe that in order to preserve the community's investment in legacy applications and libraries, it is more important to interoperate with existing languages than to extend them. To that end, Chapel directly supports interoperability with C, and has also worked with the Babel project at Lawrence Livermore National Laboratory to support a greater number of languages [234], including Fortran, Java, and Python.

The Chapel team likes to joke that they chose not to extend an existing language in order to offend all user communities equally rather than favoring one at the risk of alienating others. Joking aside, one of the encouraging results of Chapel's approach is that users from diverse language backgrounds—Fortran, C, Java, Python—have described Chapel as being familiar. That so many users find aspects of Chapel that are familiar and comfortable to them, while considering others an improvement over what they are used to, is an indication that the melting pot approach taken by Chapel can help with adoption rather than hindering it.

6.2.10 Shoot for the Moon

Another early criticism of Chapel was that the project bit off more than it could hope to complete under HPCS funding alone. This observation accurately reflects the team's intention. Chapel's founders believed that a truly successful, general parallel language would need to be very broad in its feature set and would need to involve a larger community than simply the Cray Chapel team. To this end, many of the original features were intentionally open research topics as a means of striving for a more productive solution and encouraging collaborations with the broader community. Examples of such features include user-defined data distributions (which are supported today), distributed software transactional memory (which resulted in collaborative research [44, 256] that never made it back into the master branch), and distributed garbage collection (which has not been pursued significantly, due to a lack of interested collaborators combined with a growing sense of skepticism about its value to Chapel).

6.2.11 Develop Chapel as Portable, Open-Source Software

The final theme in this discussion is the choice to develop and release Chapel as portable, open-source software. This decision was made primarily due to the fact that it is nearly impossible to get any new parallel language broadly adopted, let alone one that is not freely and generally available. Making the project open-source has also lowered barriers to involving external collaborators and helps potential users be less wary about what

might happen if support for Chapel ends. As a result, the Chapel project is being developed as a GitHub project,[1] and it is implemented and released under the Apache License, version 2.0.[2]

The portability of Chapel—both in terms of the compiler and its generated code—was also considered strategically crucial since nobody would adopt a language that only runs on systems from a single vendor. Moreover, making Chapel's implementation portable permits users to develop parallel programs on their desktops and laptops, and then move them to large-scale machines as the programs mature and resources become available. In order to maximize portability, the Chapel compiler has been developed in ISO C++ and generates ISO C99. All parallelism in Chapel is implemented using POSIX threads, and all communication can be implemented using the portable GASNet communication library's support for one-sided communication and active messages (Chapter 2). As a result of this approach, Chapel runs on most parallel systems, whether custom or commodity.

In Summary. The themes in this section have been crucial to defining the Chapel language and setting it apart from most conventional and competitive technologies. Readers who are interested in more detailed coverage of Chapel's motivating themes and philosophies are referred to various online blog articles [64, 63]. The following sections provide an overview of Chapel's main features, which have been designed with these themes in mind.

6.3 Chapel Feature Overview

This section gives an introduction to Chapel's primary features in order to provide an overview of the language. By necessity, this description only presents a subset of Chapel's features and semantics. For a more complete treatment of the language, the reader is referred to the Chapel language specification [89], materials on the Chapel website,[3] and examples from the Chapel release.[4] This section begins with the base language features and then moves on to those used to control parallelism and locality.

[1] https://github.com/chapel-lang/chapel (note that the Chapel repository was previously hosted by SourceForge and the University of Washington).

[2] http://www.apache.org/licenses/LICENSE-2.0.html (note that earlier versions of Chapel were released under the BSD and MIT licenses).

[3] http://chapel.cray.com

[4] Located in $CHPL_HOME/examples

6.3.1 Base Language Features

Chapel's base language can be thought of as the set of features that are unrelated to parallel programming and scalable computing—essentially, the sequential language on which Chapel is based. As mentioned in Section 6.2.9, Chapel was designed from scratch rather than by extending an existing language, and the base language can be thought of as those features that were considered important for productivity and for supporting user-specification of advanced language features within Chapel itself. Overall, the base language is quite large, so this section focuses on features that are philosophically important or useful for understanding Chapel code in subsequent sections.

Syntax. Chapel's syntax was designed to resemble C's in many respects, due to the fact that so many adopted languages at the time tended to utilize C syntax to greater or lesser degrees. Like C, Chapel statements are separated by semicolons, and compound statements are defined using curly brackets. Most Chapel operators follow C's lead, with some additional operators added; Chapel's conditionals and while-loops are based on C's; and so forth.

In other areas Chapel departs from C, typically to improve upon it in terms of generality or productivity. One of Chapel's main syntactic departures can be seen in its declarations which use more of a Modula-style left-to-right, keyword-based approach. For example, the following declarations declare a type alias, a variable, and a procedure in Chapel:

```
type eltType = complex;   // 'eltType' is an alias for the complex type

var done: bool = true;    // 'done' is a boolean variable, initialized to 'true'

proc abs(x: int): int {   // a procedure to compute the absolute value of 'x'
  if (x < 0) then
    return -x;
  else
    return x;
}
```

In this example, the `type` keyword introduces a new type identifier, `var` introduces a new variable, and `proc` introduces a new procedure, as noted in the comments. Other declaration keywords are used to create compile-time constants (`param`), run-time constants (`const`), iterators (`iter`), and modules that support namespace management (`module`).

Chapel uses the left-to-right declaration style in part because it better supports type inference and *skyline arrays*—arrays whose elements are themselves arrays of varying size. In addition, adopting a left-to-right declaration style aids productivity by making declarations easier for a nonexpert to read.

Basic Types. Chapel's basic scalar types include boolean values (`bool`), signed and unsigned integers (`int` and `uint`), real and imaginary floating point values (`real` and `imag`), complex values (`complex`), and strings (`string`). All of Chapel's numeric types use 64-bit values by default, though users can override this choice by explicitly specifying a bit width. For example, `uint(8)` would specify an 8-bit unsigned integer. All types in Chapel have a default value that is used to initialize variables that the user has not initialized. Numeric values default to zeroes, booleans to false, and strings to empty strings.

Chapel supports record and class types, each of which supports the creation of objects with member variables and methods. Records are declared using the `record` keyword and result in local memory allocation. Classes are declared using the `class` keyword and use heap-allocated storage. Records support value semantics while classes support reference semantics. For example, assigning between variables of record type will result in a copy of the record members by default. In contrast, assigning between variables of class type results in the two variables aliasing a single object. Records can be thought of as being similar to C++ structs while classes are similar to Java classes.

Chapel also supports tuple types that permit a collection of values to be bundled in a lightweight manner. Tuples are useful for creating functions that generate multiple values, as an alternative to adopting the conventional approach of returning one value directly and the others through output arguments. Chapel also uses tuples as the indices for multi-dimensional arrays, supporting a rank-independent programming style. The following code illustrates some simple uses of tuples in practice:

```
var t: (int, real) = (1, 2.3);     // a tuple 't' with int and real components

var (i, r) = t;                    // de-tuple 't' into new variables 'i' and 'r'

...t(1)...                         // refer to 't's first (integer) component

var coord: (real, real, real),     // a homogeneous 3-tuple of reals
    coord2: 3*real;                // an equivalent way to declare coord
```

Range and Array Types. Another built-in type in Chapel is the *range*, used to represent a regular sequence of integer values. For example, the range "`1..n`" represents the integers between 1 and *n* inclusive, while "`0..`" represents all of the nonnegative integers. Chapel's ranges tend to be used to control loops, and also to declare and operate on arrays. Ranges support a number of operators including intersection (`[]`), counting (`#`), striding (`by`), and setting the alignment of a strided range (`align`). The following Chapel code illustrates some range values and operators:

```
1..9                  // represents 1, 2, 3, ..., 9
1..9 by 2             // represents 1, 3, 5, 7, 9
1..9 by -1            // represents 9, 8, 7, ..., 1
9..1                  // represents an empty range
1..9 # 3              // represents 1, 2, 3
1..9 # -3             // represents 7, 8, 9
(1..9)[6.. by 2]      // represents 6, 8
lo..hi by 2 align 1   // represents the odd integers between 'lo' and 'hi' ( inclusive )
0..#n                 // represents 0, 1, 2, ..., n-1 (the first 'n' elements in 0..)
```

Chapel has extensive support for arrays, described in greater detail in Section 6.3.3. However, to introduce the concept, the following declarations create three array variables.

```
var Hist: [-3..3] int,                // a 1D array of integers
    Mat: [0..#n, 0..#n] complex,      // a 2D array of complexes
    Tri: [i in 1..n] [1..i] real;     // a 'triangular' skyline array
```

The first example declares a 1D array, *Hist*, whose indices range from -3 to 3, and whose elements are integers. The second declaration creates a 2D $n \times n$ array of complex values, *Mat*, which uses 0-based indexing. The final example is a 1D skyline array named *Tri* that uses 1-based indexing. Each of *Tri*'s elements is a 1-based 1D array of reals whose length is equal to its index in the outer array. This essentially creates a "triangular" array of arrays.

Type Inference. Chapel supports type inference as a means of writing code that is both concise and flexible. For example, the type specifier of a variable or constant declaration can be elided when an initialization expression is provided. In such cases, the Chapel compiler infers the type of the identifier to be that of the initialization expression. The following code illustrates some examples:

```
param pi = 3.1415;        // '3.1415' is a real, so 'pi' is too
var count = 0;            // '0' is an integer, so 'count' is too
const perim = 2*pi*r;     // if 'r' is a real/complex, 'perim' will be too
var len = computeLen();   // 'len' is whatever type computeLen() returns
```

The first two declarations are fairly straightforward—the type of each initializing literal expression is well-defined by Chapel (`real` and `int`, respectively), so the identifiers being declared have matching type. In the third line, *perim*'s type is based on the type resulting from multiplying *r* by an integer and a real. If *r* were a `real`, *perim* would be a `real`; if it were a `complex`, *perim* would be a `complex`; etc. In the final line, *len* will be

whatever type the procedure `computeLen()` returns. Note that these final two forms have the advantage of making these declarations flexible with respect to changes in the types of *r* and `computeLen()` at the cost of making the declarations a little less self-documenting—a reader would need to know the types of *r* and `computeLen()` in order to determine the types of *perim* and *len*.

Chapel's type inference also applies to function declarations: a function's argument and return types may be omitted. Omitted argument types are inferred by the compiler by inspecting the function's callsites and adopting the types of the corresponding actual arguments. Such function declarations are generic, and the compiler will create distinct instantiations of the routine for each unique callsite type signature, resulting in a capability much like C++'s template functions. If a function's return type is omitted, it is inferred by unifying the types of the expressions generated by its `return` statements.

As a simple example, consider the `abs()` function shown previously, but written in its type-inferred form:

```
proc abs(x) {          // 'x's type and the return type of abs() will be inferred
  if (x < 0) then
    return -x;
  else
    return x;
}
```

In this version of `abs()`, the formal argument *x* has no type specifier, and no return type is given. As a result, `abs()` may be called with any type that supports less-than comparison against integers and the unary negation operator—e.g., integers, floating point values, or any user-defined type that supports these operators. The compiler infers the return type of `abs()` by noting that both of the returned expressions have the same type[5] as *x*, in which case the return type will match the argument type. If the function is called within a user's program as `abs(3)` and `abs(4.5)`, the compiler would create both `int` and `real` instantiations of `abs()`.

For-loops and Iterators. Chapel's for-loops are different from C's, both syntactically and semantically. In Chapel, for-loops are used to iterate over data structures and to invoke iterator functions. Chapel's for-loops declare *iteration variables* that represent the values yielded by the *iterand expression*. These variables are local to a single iteration of the loop's body. The following statements demonstrate some simple for-loops:

[5] . . .assuming that unary negative preserves *x*'s type—if not, the compiler will attempt to find a unifying type that supports both returned expressions and throw an error if it cannot.

```
for i in 1..n do      // print 1, 2, 3, ..., n
  writeln(i);

for elem in Mat do   // double all elements in 'Mat'
  elem *= 2;
```

The first loop iterates over the range "`1..n`", referring to the individual integer values using the iteration variable *i*. Each iteration of the loop's body gets its own private local copy of *i*, so it cannot be used to carry values across distinct iterations. In addition, a range's iteration variables are constant, and therefore may not be reassigned within the loop body.

The second loop iterates over the *Mat* array, referring to its elements using the iteration variable *elem*, which is once again local and private to the loop body. When iterating over an array, the iteration variable refers to the array's elements; thus, assignments to it will modify the array's values. Here, the loop has the effect of iterating over all of the array's values, doubling each one.

Chapel loops can also be used to iterate over multiple iterands in a lockstep manner, known as *zippered iteration*. As an example, the following loop iterates over the elements of *Hist* and the unbounded range "`1..`" in a zippered manner:

```
for (elem, i) in zip(Hist, 1..) do
  writeln("Element #", i, " of Hist is: ", elem);
```

In addition to looping over standard data types, Chapel programmers can write their own iterator functions that can be used to drive for-loops. As a simple example, the following declaration creates an iterator which generates the first *n* elements of the Fibonacci sequence:

```
iter fib(n) {            // generates 'n' Fibonacci numbers
  var current = 0,       // 'current' and 'next' store two consecutive values
      next = 1;          //    from the sequence

  for i in 1..n {
    yield current;       // yield the current value
    current += next;     // increment it by the next
    current <=> next;    // swap the two values
  }
}
```

Iterator functions generate results for their callsites using *yield* statements. For example, in the Fibonacci iterator above, each iteration yields its value of *current* back to the callsite.

Execution continues after the yield statement until the iterator returns (either via a `return` statement or by falling out of the function).

Iterator functions are typically invoked using for-loops. For example, the following loop would print out the first *n* Fibonacci numbers:

```
for (i,f) in zip(1..n, fib(n)) do
  writeln("fib(", i, ") = ", f);
```

In this example, the iteration variable *f* takes on the values generated by `fib()`'s yield statements.

Iterators were included in Chapel for their benefit in abstracting loop-nest implementation details away from the loops themselves, providing reuse and customization benefits similar to what traditional functions do for straight-line code. While new users often worry that iterators may incur unnecessary performance overheads, it is important to note that most iterators, like the Fibonacci example above, can be implemented simply by inlining the iterator's body into the loop invocation and then replacing the `yield` statement with the loop body.

Other Base Language Features. In addition to the features described here, Chapel's base language also supports a number of additional constructs, including: enumerated types and type unions; type queries; configuration variables that support command-line options for overriding their default values; function and operator overloading and disambiguation; default argument values and keyword-based argument passing; meta-programming features for compile-time computation and code transformation; modules for namespace management; and I/O to files, strings, memory, and general data streams.

6.3.2 Task Parallelism

As alluded to in Section 6.2.2, all parallelism in Chapel is ultimately implemented using *tasks*—units of computation that can and should be executed in parallel. All Chapel programs begin with a single task that initializes the program's modules and executes the user's `main()` procedure. This section provides an overview of Chapel's features for creating tasks and synchronizing between them.

Unstructured Task Parallelism. The simplest way to create a task in Chapel is by prefixing a statement with the `begin` keyword. This creates a new task that will execute the statement and then terminate. Meanwhile, the original task goes on to execute the statements that follow. As a trivial example, the following code uses a `begin` statement to

create a task to execute the compound statement while the original task continues with the `writeln()` that follows it.

```
writeln("The original task prints this");
begin {
  writeln("A second task will be created to print this");
  computeSomething();           // it will then compute something
  writeln("The second task will terminate after printing this");
}
writeln("The original task may print this as the second task runs");
```

Because the two tasks in this example can execute concurrently, the final `writeln()` could be printed before the second and third `writeln()`s, between them, or after them, depending on how the tasks are scheduled.

Tasks in Chapel are anonymous, so there is no way to name a task directly. The two ways in which a user can check for task completion are through the `sync` statement or by coordinating through shared synchronization variables, described below.

The Sync Statement. Chapel's `sync` keyword prefixes a statement and causes the task encountering it to wait for all tasks created within the statement's dynamic scope to complete before proceeding. As an example, the use of the `sync` statement in the following code will wait for all the tasks generated by a recursive binary tree traversal to complete before the original task continues.

```
sync { traverseTree(root); }
writeln("All tasks created by traverseTree() must now be done");

proc traverseTree(node) {
  processNode(node);

  if (node.left != nil) then          // If there is a left child ...
    begin traverseTree(node.left);    // ... create a task to visit it

  if (node.right != nil) then         // Ditto for the right child ...
    begin traverseTree(node.right);
}
```

As can be seen, the `sync` statement is a big hammer. For finer-grain interactions between tasks, programmers can use special variable types that support data-centric coordination—Chapel's synchronization and atomic variable types—described in the following sections.

Synchronization Variables. A Chapel *synchronization variable* is like a normal variable, except that in addition to storing its value, it also stores a *full/empty state* that is used to guard reads and writes. As mentioned in Section 6.1.2, this concept was adopted from the similar Tera MTA and Cray XMT features [11, 241]. By default, a read of a synchronization variable blocks until the variable is full, reads the value, and leaves the variable in the empty state. Similarly, a write blocks until the variable is empty, writes the new value, and then leaves it full.

As a simple example, the following code implements a bounded-buffer producer/consumer idiom using an array of synchronization variables to implement the buffer:

```
1  var buff$: [0..#buffsize] sync real;

3  begin producer(numUpdates);   // create a task to run the producer
4  consumer();                   // while the original task runs the consumer

6  proc producer(numUpdates) {
7    var writeloc = 0;
8    for i in 1..numUpdates {
9      buff$[writeloc] = nextVal();  // this write blocks until 'empty', leaves 'full'
10     writeloc = (writeloc + 1) % buffsize;
11   }
12   buff$[writeloc] = NaN;          // write a sentinel to indicate the end
13 }

15 proc consumer() {
16   var readloc = 0;
17   do {
18     const val = buff$[readloc];   // this read blocks until 'full', leaves 'empty'
19     processVal(val);
20     readloc = (readloc + 1) % buffsize;
21   } while (val != NaN);
22 }
```

In this program, line 1 declares an array, *buff$*, whose elements are of type `sync real`. Thus, each element is a synchronized floating point value that carries a full/empty state along with its value. Because the array's declaration does not contain an initialization expression, its elements start in the empty state. Since incorrect accesses to synchronization variables can result in deadlock, Chapel programmers typically name them using a $ by convention, in order to alert readers to their presence and avoid introducing inadvertent reads or writes that may never complete.

Continuing the example, line 3 creates a task to execute the producer while the original task continues on to line 4 where it executes the consumer. The two tasks each sit in a tight loop, writing (lines 8–11) or reading (lines 17–21) *buff$*'s elements, respectively. Note that

the typical safety checks required to prevent the producer from overwriting elements or the consumer from getting ahead of the producer are not required in this implementation—the full/empty state associated with each *buff$* element naturally prevents these error cases from occurring.

In addition to the default read/write semantics, synchronization variables support a number of methods that permit other modes of reading/writing their values. For example the `readFF()` method provides a way to read a synchronization variable, blocking until it is full, but leaving it full rather than empty. Similarly, `readXX()` permits the task to peek at a synchronization variable's value regardless of the full/empty state.

In addition to providing a controlled way of sharing data, synchronization variables also play an important role in defining Chapel's memory consistency model. Typical Chapel variables are implemented using a relaxed memory consistency model for the sake of performance, which makes them an unreliable choice for coordinating between tasks. By contrast, loads and stores cannot be reordered across synchronization variable accesses, which also serve as memory fences. This permits synchronization variables to be used as a means of coordinating data sharing for larger, more relaxed data structures.

As an example, the following code fragment hands off a buffer of data (*buff*) between two tasks:

```
1  var buff: [1..n] real;
2  var buffReady$: sync bool;

4  begin {
5    fillBuffer(buff);
6    buffReady$ = true;          // signal the buffer is filled by making buffReady$ 'full'
7  }

9  {
10   const val = buffReady$;    // block until buffReady$ becomes full
11   processArray(buff);        // the implicit memory fence guarantees 'buff's readiness
12 }
```

The first task (lines 4–7) fills *buff* and then signals to the other task that the buffer is ready by filling the synchronization variable *buffReady$*. Meanwhile the original task (lines 9–12) blocks on the *buffReady$* flag until it is full (line 10) and only accesses the buffer once it is. Note that using a normal variable for *buffReady$* would not be guaranteed to work since it would be subject to relaxed consistency and therefore could have its loads/stores reordered with respect to *buff* by either the compiler or architecture. Making *buff* into an array of synchronization variables would also achieve the desired result, but would add significant overhead to every access of *buff*.

Chapel supports a variation of synchronization variables called *single-assignment variables*. They are almost identical except that once their full/empty state is set to full, it can never be emptied. For this reason, default reads of single-assignment variables use the `readFF()` semantics described above.

Single-assignment variables (and synchronization variables, for that matter) can be used to express future-oriented parallelism in Chapel by storing the result of a `begin` statement into them. As an example, consider the following code snippet:

```
var area1$, area2$: single real;

begin area1$ = computeArea(shape1);
begin area2$ = computeArea(shape2);

doSomethingElse();

const totalArea = area1$ + area2$,
      areaDiff = abs(area1$ - area2$);
```

This program creates two single-assignment variables, *area1$* and *area2$*. It then uses `begin` statements to create a pair of tasks, each of which computes the area of a shape and stores the result into its respective single-assignment variable. Meanwhile, the original task goes on to do something else. When it is done, it computes the total area by reading the two single-assignment variables. If the helper tasks have not yet generated their results, it will block due to the full/empty semantics of the single-assignment variables. Due to the single-assignment semantics, the variables can then be read again without blocking, for example to compute *areaDiff*, the magnitude of the difference between the areas.

Atomic Variables. Chapel also supports data-centric coordination between tasks using *atomic variables*. These are variables that support a set of common atomic operations which are guaranteed to complete without another task seeing an intermediate or incomplete result. Chapel's atomic variables are modeled after those of the C11 standard and benefit from the design work done there.

As an example of using atomic variables, consider the following program which uses atomic variables to compute a histogram in a manner that ensures updates will not be lost due to read-read-write-write ordering issues:

```
var hist: [0..#histSize] atomic int;
forall elem in Mat {
  const bucket = computeBucket(elem);
  hist[bucket].add(1);
}
```

This program uses a forall-loop (to be introduced in Section 6.3.3) to perform a parallel iteration over an array named *Mat*. For each element, *elem*, the corresponding bucket is computed and incremented. The increment is performed using the `add()` method for atomic variables, causing the argument value to be accumulated atomically into the histogram element. Since multiple tasks may update a single bucket value simultaneously, using a normal array of integers and incrementing them using addition may cause two tasks to read the same value before either had written its update, causing one of the updates to be lost. Proper use of atomic variables can guard against such races in a reasonably lightweight manner, given appropriate hardware support.

Structured Task Parallelism. In addition to the `begin` keyword, Chapel supports two statements that create groups of tasks in a structured manner. The first of these is the `cobegin` statement—a compound statement in which a distinct task is created for each of its component statements. The `cobegin` statement also makes the original task wait for its child tasks to complete before proceeding. Note that this differs from the semantics of the `sync` statement in that only the tasks created directly by the `cobegin` are waited on; any others follow normal fire-and-forget semantics. Although the `cobegin` statement can be implemented using `begin` statements and synchronization variables, that approach adds a considerable cost in verbosity for the user and fails to convey the intent as clearly to the compiler for the purpose of optimization.

As a simple example, the producer/consumer tasks from the earlier bounded buffer example could have been created with a `cobegin` as follows:

```
cobegin {
  producer(numUpdates);
  consumer();
}
writeln("We won't get here until producer() and consumer() are done");
```

Chapel's other form of structured parallelism is the *coforall-loop* which is like a traditional for-loop except that it creates a distinct task for each iteration of the loop body. Like the `cobegin` statement, `coforall` has an implicit *join* that causes the original task to wait for all of its children to complete before proceeding.

As an example, the following loop creates a distinct task for each element in an array:

```
coforall elem in Mat do
  processElement(elem);
writeln("We won't get here until all elements have been processed");
```

For very large arrays, coforall-loops tend to be overkill since you would not typically want to create a distinct task for every array element. In such cases, programmers would typically use the data-parallel constructs of the following section instead. In practice, the `coforall` loop tends to be used when the number of iterations is close to the target hardware's natural degree of parallelism, or when true concurrency between iterations is required (e.g., if distinct iterations synchronize with one another).

6.3.3 Data Parallelism

Chapel's task-parallel features support very explicit parallel programming with all the related hazards, such as race conditions and deadlock. In contrast, Chapel's data-parallel features support a more abstract, implicitly parallel style of programming that is typically easier to use. The primary features for data parallelism are forall-loops, ranges, domains, and arrays, described in this section.

forall-loops. The *forall-loop* is Chapel's data parallel loop construct. Syntactically, it is similar to for-loops and coforall-loops. As a simple example, the following code uses a forall-loop to iterate over a range in parallel:

```
forall i in 1..n do
  A[i] += 1;
```

The net effect of this loop is to increment elements 1 through n of array A in parallel.

Unlike for-loops, which are executed using a single task, and coforall-loops, which use a task per iteration, forall-loops use an arbitrary number of tasks, as determined by the loop's iterand. For example, in the forall-loop above, the range value "`1..n`" determines the number of tasks used to execute this loop. For typical iterands, this choice is based on the amount of hardware parallelism available. Many parallel iterators also have arguments that permit the user to specify or influence the number of tasks used to execute the loop. Like all loop forms, Chapel's forall-loops support zippered iteration in which corresponding elements are generated together in parallel.

Because the number of tasks used to implement a forall-loop is not known *a priori*, forall-loops must be *serializable*. That is, it must be legal to execute the loop using a single task. A consequence of this is that there can be no synchronization dependences between distinct iterations of the loop, since there is no guarantee that they would be executed by distinct tasks.

Forall-loops also support an expression-level form, as well as a shorthand syntax that makes use of square brackets. For example, the forall-loop above could have been written:

`[i in 1..n] A[i] += 1;`. The syntactic similarity between this shorthand and array type specifiers is intentional—one can read an array type like `[1..n] string` as "for all indices from 1 to *n*, store a string."

As part of Chapel's multiresolution approach, advanced users can implement their own parallel iterators which can be invoked using forall-loops. This is done by writing parallel iterators that create the tasks to implement the loop and then determine how the iteration space will be divided amongst them. These iterators are themselves implemented using Chapel's lower-level features, such as task parallelism and base language concepts. With this mechanism, users can write very simple iterators that statically partition the iteration space, as well as more complex ones that decompose the iteration space dynamically. The details of authoring parallel iterators are beyond the scope of this chapter; interested readers are referred to published work [68, 32] and the Chapel release for further details and examples.

Domains and Arrays. In Chapel, a *domain* is a first-class language concept that represents an index set. Domains are used to drive loops and to declare and operate on arrays. The following code creates constant domains that describe the size and shape of the arrays declared in Section 6.3.1:

```
const HistSpace: domain(1) = {-3..3},
      MatSpace = {0..#n, 0..#n},
      Rows = {1..n},
      Cols: [Rows] domain(1) = [i in Rows] {1..i};
```

The first line declares a 1-dimensional domain describing the index set from -3 to 3, inclusive. The second and third lines use Chapel's type inference to declare a 2D $n \times n$ domain and a 1D n-element domain. The final declaration creates an array of domains, using a forall-loop to initialize each element based on its index.

Given these domains, the original array declarations of Section 6.3.1 could be rewritten as follows:

```
var Hist: [HistSpace] int,
    Mat: [MatSpace] complex,
    Tri: [i in Rows] [Cols[i]] real;
```

The original declarations were equivalent to these ones; they simply resulted in the creation of *anonymous domains*. The benefit of naming domains is that it permits an index set to be referred to symbolically throughout a program, providing readers and the compiler with a

clearer indication of the relationships between arrays and iteration spaces. As an example, the following loop can be proven to require no array bounds checks since *HistSpace* is the domain used to declare *Hist*.

```
forall i in HistSpace do
  Hist[i] = 0;
```

In addition to dense rectangular domains and arrays, Chapel supports a variety of other domain types including *associative*, *sparse*, and *unstructured* domains. Associative domains store a set of index values of arbitrary type, such as strings, floating point values, or class object references. An associative array can be thought of as providing a hash table or dictionary capability, mapping the domain's indices to array elements. Unstructured domains have anonymous indices and are designed to represent pointer-based data structures like unstructured graphs. Sparse domains represent arbitrary subsets of a parent domain's index set. Their arrays store an implicit "zero" value for any index that is within the parent domain but not the child.

All of Chapel's domain types support a rich set of operations including serial and parallel iteration, membership tests, and intersection. Regular domains also support operators and methods that permit new domains to be created from them; for example, one such method simplifies the creation of boundary conditions.

Chapel's arrays support a rich set of operations including serial and parallel iteration, random access, slicing, reshaping, aliasing, and reindexing. Arrays can also be logically reallocated by reassigning their domain variables. When a domain's index set is modified, all arrays declared in terms of that domain are logically reallocated to reflect its new index set. Array values corresponding to indices that persist between the old and new domain values are preserved.

Promotion. In addition to explicit forall-loops, data-parallelism in Chapel can also be expressed using *promotion* of scalar functions and operators. When a domain or array argument is passed to a function or operator that is expecting a scalar argument, the function is invoked in parallel across all of the domain's indices or array's elements. These promotions are equivalent to forall-loops, but often result in a more compact expression of parallelism. As an example, the forall loop shown earlier to zero out *Hist* could be written as `Hist = 0;`—effectively, a promotion of the scalar assignment operator.

When multiple scalar arguments are promoted, the resulting expression is equivalent to a zippered forall-loop. For example, given the standard `exp()` function for exponentiation, the call `exp(A, B)` with conforming arrays *A* and *B* would be equivalent to the following forall-loop expression:

```
forall (a, b) in zip(A, B) do exp(a, b);
```

Note that both standard and user-defined functions and operators can be promoted in this way.

Reductions and Scans. Chapel's other major data-parallel features are *reduction* and *scan* expressions. Reductions can be used to flatten one or more dimensions of a collection of values while scans are used to compute parallel prefix operations. As an example, the following statement computes the largest sum of squares value over corresponding elements of *A* and *B*:

```
var biggest = max reduce (A**2 + B**2);
```

Note that the exponentiation and plus operators are promoted in this example.

Chapel provides a number of standard reduction and scan operators, such as sum, product, logical and bitwise operations, and max/min (with or without location information). Users can also write their own reduction and scan operators by specifying functions to accumulate and combine input and state values. Though this topic is beyond the scope of this paper, our approach can be viewed in the release or read about in published work [95].

6.3.4 Locality Features

Chapel's final feature area permits a programmer to control and reason about locality. At the low level, a Chapel programmer can explicitly specify the system resources on which a task is run or a variable is allocated. At a higher level, Chapel programmers can specify how domains and arrays are distributed across a system, resulting in distributed-memory data-parallelism. This section touches on both styles.

The Locale Type. The core of Chapel's locality features is the *locale* type. Locales represent units of the target system architecture that are useful for reasoning about locality and affinity. For most conventional parallel architectures, a *locale* tends to describe a compute node, such as a multicore or SMP processor. Due to Chapel's PGAS memory model [288], tasks executing within a given locale can access lexically visible variables whether they are allocated locally or on a remote locale. However, Chapel's performance model indicates that variable accesses within a task's locale will be cheaper than remote ones. This approach supports productivity through Chapel's global namespace, while still

supporting the ability to obtain scalable performance by being sensitive to where tasks execute relative to the data they access.

When executing a Chapel program, users specify the number of locales on which it should run using an execution-time command-line flag. Within the Chapel source code, these locales can be referred to symbolically using a built-in, zero-based 1D array named *Locales*, which stores *numLocales* locale values. These values represent the system resources on which the program is executing and permit the user to refer to them and query their properties. Like any other array, *Locales* can be reshaped, sliced, reindexed, etc.

As a simple example, the following statement computes the total amount of memory available to the locales on which a Chapel program is running:

```
const totalMem = + reduce Locales.physicalMemory();
```

This idiom uses a `physicalMemory()` method that is supported by the locale type, promoting it across the entire *Locales* array. It then uses a reduction to sum the individual memory sizes into a single value, *totalMem*. Other locale methods support queries such as the number of processor cores, the number of tasks or threads executing, the callstack limit, the locale's name, its ID, and so forth.

On-Clauses. Chapel programmers specify that a statement should execute on a specific locale using an *on-clause*. The on-clause takes a single operand that specifies which locale to target. If the expression is a variable, the statement will execute on the locale in which the variable is stored. As an example, consider the following statements:

```
on Locales[numLocales-1] do
  writeln("Hello from the last locale");

on node.left do
  traverseTree(node.left);
```

The first statement causes a message to be printed from the last locale on which the program is executing. The second statement specifies that the `traverseTree()` function should execute on whichever locale owns *node*'s left child. In practice, data-driven on-clauses like this tend to be preferable since they make the code more independent of the number of locales on which the program is running.

It is important to emphasize that on-clauses do not introduce parallelism into a program, keeping with Chapel's theme of using distinct concepts for parallelism and locality. However, on-clauses and parallel constructs compose naturally. For example, to launch an asynchronous remote task to traverse the left subtree above, we could have used:

```
begin on node.left do
  traverseTree(node.left);
```

Another common idiom is to launch a task per locale via a coforall-loop like the following:

```
coforall loc in Locales do
  on loc do
    writeln("Hello from locale ", loc.name);
```

This loop effectively generates traditional SPMD-like parallelism.

Within a Chapel program, the locale on which a variable is stored, or a task is running, can be queried. All variables support a *.locale* method that returns the locale in which it is allocated. For tasks, a built-in variable, *here*, can be used to query the locale on which the current task is executing.

Domain Maps, Layouts, and Distributions. Chapel's locales can also be used to create global-view, distributed arrays. Every Chapel domain is defined in terms of a *domain map* that specifies how it, and its arrays, should be implemented. When no domain map is specified (as in the preceding sections), a default domain map is used. It maps the domain's indices and array's elements to the current locale (*here*). Domain maps like these which target a single locale are referred to as *layouts* since they only specify how domains and arrays are stored in local memory. Domain maps can also target multiple locales as a means of storing distributed index sets and arrays; these are referred to as *distributions*.

As a simple example of a distribution, the following redefinition of *MatSpace* from Section 6.3.3 would result in a distributed Block-Cyclic implementation of its index set and of the *Mat* array that was declared in terms of it:

```
const MatSpace = {0..#n, 0..#n}
                 dmapped BlockCyclic(startIdx=(0,0), blocksize=(8,8));
```

This declaration says that *MatSpace*'s domain map should be an instance of the standard *BlockCyclic* distribution. Its arguments specify that 8×8 blocks should be dealt out to the locales, starting at index $(0, 0)$. By default, it will target all of the locales on which the program is running, reshaping them into a square-ish virtual array of locales. The user can also pass an optional array of target locales to the *BlockCyclic* constructor as a means of precisely controlling which locales the distribution should target and how they should be arranged logically.

Note that because of Chapel's global namespace and global-view arrays, a change to a domain declaration like this is the only thing required to convert a shared-memory parallel program into one that supports distributed-memory execution. Forall-loops over distributed domains/arrays are typically implemented such that each locale iterates over the indices/elements that it owns locally, providing a natural model for affinity.

User-Defined Domain Maps. As part of Chapel's multiresolution design, advanced users can author their own domain maps as a means of controlling the distribution and layout of domains and arrays, as well as the implementation details of their iterators, accessor functions, etc. Creating a domain map requires creating three descriptor classes, one to represent the domain map itself, a second to represent one of its domains, and a third to represent one of its arrays. These descriptors must support a required interface that the compiler targets when lowering high-level global-view operations down to the per-locale data structures and operations required to implement them. They can also support optional interfaces that the compiler can use for optimization purposes, when present.

Domain maps are considered the highest-level concept in Chapel's feature set because they tend to be written using data parallel features, task parallel features, locality features, and the base language. As an example, the *BlockCyclic* domain map shown above uses on-clauses to create local descriptors on each target locale to represent its individual portion of the global domain or array. It uses local domains, arrays, and data parallel operations to implement each locale's portion of the global array. It uses task parallelism and on-clauses to implement parallel iterators that execute using the target locales. And it uses classes, iterators, generics, and type inference from the base language to do all of this in a productive way.

As part of the project's research goals, all arrays in Chapel are implemented using the same domain map framework that an end-user would. This is done to avoid a situation in which standard "known" domain maps perform well but user-defined domain maps result in a significant performance penalty. Instead the Chapel team has chosen to use the same framework for all arrays as a forcing function to ensure that user-defined domain maps can achieve competitive performance.

A more detailed description of user-defined domain maps is beyond the scope of this chapter. Interested readers are referred to the Chapel release and published work for more information about, and examples of, user-defined domain maps [71, 67].

6.4 Project Status

As mentioned previously, Chapel is an active and ongoing effort. At the time of publication, all of the features described in this chapter are implemented and work correctly,

with one exception: skyline arrays as described in Section 6.3.1 are not yet implemented. As a result, today's arrays of arrays must have inner arrays that share a common domain. Workarounds for this feature exist for users who require such data structures today.

The Chapel team creates two Chapel releases per year, each spring and fall. Highlights of recent releases have included adding support for vectorization of forall-loops in cooperation with the back-end compiler; a growing standard library of common operations, including FFTW routines and file system utilities; a *chpldoc* utility for source-based documentation; and improvements to portability and performance. Other nascent efforts include support for an interpreted Chapel environment, a tool for visualizing communication and tasking intensity within Chapel programs, and Python interoperability.

Chapel's performance remains hit-or-miss at present, depending on the scenario. Generally speaking, performance has recently been improving significantly, particularly since the HPCS program wrapped up.[6] For single-locale programs, execution is increasingly competitive with hand-coded C+OpenMP [32]. Multi-locale executions can be more or less competitive with conventional approaches, depending on the computational idioms and target architecture, but have also been improving with time. Generally speaking, more work is required to optimize Chapel's communication, both in terms of applying traditional global-view communication optimizations [255] and simply by reducing the compiler's tendency to insert communication conservatively, which tends to thwart back-end compilers' serial code optimizations.

The user community's reaction to Chapel has been increasingly positive as the project has progressed. Initially, HPC users expressed a great deal of skepticism about the decision to pursue a new language, largely due to lingering disenchantment around HPF's failure in the 1990's [161]. As the community learned more about Chapel and grew to understand its philosophical and practical divergences from HPF, pessimism gave way to curiosity and cautious optimism. At present, many potential users believe that a mature implementation of Chapel would be very attractive to them, and the number of earnest users who are trying it out for their projects has grown markedly with recent releases.

Future Directions. Although Chapel's primary features are implemented and working, other aspects of the language are still being improved, both in the implementation and specification of the language.

In the base language, a major lack is a capability for handling error conditions in large-scale codes, whether using exceptions or some other feature more specialized for parallel execution. This was a known lack in the original design, but one that was deferred due to resource constraints, and which has become increasingly important as the number of

[6] http://chapel.sourceforge.net/perf/

Chapel users grows. Other areas for base language improvements include constrained generic interfaces, additional support for interoperability with other languages, and improved constructor/destructor features.

As part of Chapel's task parallel features, we want to add a notion of identifying logical teams of tasks. This capability would give users the ability to identify and operate on subsets of tasks using collective operations such as barriers, broadcasts, reductions, and *eurekas*—the ability for one task to signal that its team members can stop executing. Task teams may also serve as a means of assigning distinct execution policies to tasks, such as "these tasks may be work-stolen and load-balanced" versus "these tasks should be bound to their own threads and run to completion."

Within Chapel's data parallel features, a major lack is support for *partial reductions*, which are crucial for implementing many algorithms effectively, particularly in linear algebra. We would also like to add support for replicated array dimensions similar to ZPL's flood and grid dimensions [94]. At a lower-level, improvements are planned for Chapel's parallel iterator framework to support increased performance and flexibility.

The main effort that is underway in the area of locality is support for *hierarchical locale models* [274]. Chapel's classic definition of locales is very adept at describing *horizontal locality* such as that which exists between nodes of a commodity cluster. However, Chapel programmers have traditionally had no way to target specific processors or memories on compute nodes involving Non-Uniform Memory Access (NUMA) domains or heterogeneous resources. Hierarchical locales are designed to address this, using Chapel's multiresolution philosophy to permit programmers to model their target architectures in terms of objects that support a standard tasking and memory interface. Programmers can then access such sublocales using traditional on-clauses and distributions. At present, this framework is in use within every Chapel program, though ongoing work strives to efficiently target nonflat compute node architectures.

Summary. Overall, the Chapel language and compiler have demonstrated a great deal of promise with respect to general, productive, multiresolution parallel programming. We encourage potential users to give Chapel a try and to report back with feedback and areas for improvement. In evaluating Chapel, we suggest focusing less on what Chapel's current performance happens to be, and more on whether you agree that the language will be able to generate competitive performance as the implementation matures. We believe that a revolutionary and scalable parallel programming language is unlikely to materialize overnight, so urge parallel programmers to exercise patience with new languages like Chapel rather than giving up hope prematurely. Moreover, being an open-source project, we encourage programmers to help become part of the solution rather than simply sitting on the sidelines.