Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson,
Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus
Chapel Team
Cray Inc.
Seattle, WA, USA
chapel_info@cray.com

***Abstract*—Chapel is a programming language whose goal is to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as combining the strengths of Python, Fortran, C/C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel wrapped up, and the team embarked on a five-year effort to improve Chapel's appeal to end-users. This paper follows up on our CUG 2013 paper by summarizing the progress made by the Chapel project since that time. Specifically, Chapel's performance now competes with or beats hand-coded C+MPI/SHMEM+OpenMP; its suite of standard libraries has grown to include FFTW, BLAS, LAPACK, MPI, ZMQ, and other key technologies; its documentation has been modernized and fleshed out; and the set of tools available to Chapel users has grown. This paper also characterizes the experiences of early adopters from communities as diverse as astrophysics and artificial intelligence.**

***Keywords*-Parallel programming; Computer languages**

I. Introduction

Chapel is a programming language designed to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as striving to create a language whose code is as attractive to read and write as Python, yet which supports the performance of Fortran and the scalability of MPI. Chapel also aims to compete with C in terms of portability, and with C++ in terms of flexibility and extensibility. Chapel is designed to be general-purpose in the sense that when you have a parallel algorithm in mind and a parallel system on which you wish to run it, Chapel should be able to handle that scenario.

Chapel's design and implementation are led by Cray Inc. with feedback and code contributed by users and the open-source community. Though developed by Cray, Chapel's design and implementation are portable, permitting its programs to scale up from multicore laptops to commodity clusters to Cray systems. In addition, Chapel programs can be run on cloud-computing platforms and HPC systems from other vendors. Chapel is being developed in an open-source manner under the Apache 2.0 license and is hosted at GitHub.[1]

[1]https://github.com/chapel-lang/chapel

The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems program (HPCS). HPCS wrapped up in late 2012, at which point Chapel was a compelling prototype, having successfully demonstrated several key research challenges that the project had undertaken. Chief among these was supporting data- and task-parallelism in a unified manner within a single language. This was accomplished by supporting the creation of high-level data-parallel abstractions like parallel loops and arrays in terms of lower-level Chapel features such as classes, iterators, and tasks.

Under HPCS, Chapel also successfully supported the expression of parallelism using distinct language features from those used to control locality and affinity—that is, Chapel programmers specify *which* computations should run in parallel distinctly from specifying *where* those computations should be run. This permits Chapel programs to support multicore, multi-node, and heterogeneous computing within a single unified language.

Chapel's implementation under HPCS demonstrated that the language could be implemented portably while still being optimized for HPC-specific features such as the RDMA support available in Cray® Gemini™ and Aries™ networks. This allows Chapel to take advantage of native hardware support for remote puts, gets, and atomic memory operations.

Despite these successes, at the close of HPCS, Chapel was not at all ready to support production codes in the field. This was not surprising given the language's aggressive design and modest-sized research team. However, reactions from potential users were sufficiently positive that, in early 2013, Cray embarked on a follow-up effort to improve Chapel and move it towards being a production-ready language. Colloquially, we refer to this effort as "the five-year push."

This paper's contribution is to describe the results of this five-year effort, providing readers with an understanding of Chapel's progress and achievements since the end of the HPCS program. In doing so, we directly compare the status of Chapel version 1.17, released last month, with Chapel version 1.7, which was released five years ago in April 2013.

At the end of the HPCS program, Chapel had a number of issues that might prevent a potential user from adopting it. Chief among these was concern about Chapel's performance and scalability, which were sorely lacking at the time. For that reason, users wondered whether Chapel would ever be able to compete with *de facto* HPC programming models like Fortran, C, or C++ with MPI and OpenMP. Over the past five years, performance improvements have been a major focus for our team, and we report on this effort in Section V, showing recent Chapel performance results for well-known benchmarks.

For other users who were less concerned about performance, other barriers remained. For some, it was the state of the base language. Although Chapel had demonstrated its unique features for specifying parallelism and locality by the end of HPCS, many of its more traditional language features were still in their infancy, reflecting the program's focus on research-oriented features that were unique to HPC and Chapel. Section III provides an overview of key improvements to the language's feature set. Other users expressed interest in a larger suite of libraries, similar to what they were accustomed to in C++, Java, or Python. In Section IV we describe improvements that have been made in Chapel's library support since version 1.7.

Still other users had reservations about Chapel's overall maturity and ecosystem, based on the state of its documentation, tools, memory leaks, user support, and/or community. These areas have also seen significant improvements over the past five years, and we review some highlights of these efforts in Section VI. After summarizing the improvements made over the past five years, the paper wraps up with perspectives from current and prospective Chapel users in Section VII.

In the next section, we provide a high-level comparison between Chapel and other prominent HPC programming models. Note that this paper does not seek to present an introduction to Chapel itself—for that, we refer readers to the Chapel website[2] and online documentation[3] including the Chapel language specification [1]. Readers who are interested in a more detailed history of Chapel's formative years under the HPCS program may be interested in our CUG 2013 paper [2] as well as the Chapel chapter in Pavan Balaji's excellent book, *Programming Models for Parallel Computing* [3].

II. Related Work

The most obvious and important point of comparison for Chapel is the message passing interface, MPI [4], since the vast majority of HPC programs are written using it. MPI programs are typically written in a Single Program, Multiple Data (SPMD) style in which users execute multiple copies of their program, having written it to cooperate with other instances of itself through the passing of messages and calls to collective routines. Unlike MPI, Chapel supports a *global view* of control flow. Each Chapel program starts as a single (conceptual) task that runs the user's `main()` procedure. From this original task, additional parallelism is created using data- and task-parallel constructs. Where the unit of parallelism in an MPI program is the process, Chapel's parallelism is expressed in terms of lighter-weight tasks that can run within a single shared-memory *locale* or distributed across several of them.

Chapel's global view also extends to its namespace which permits any task to refer to any variable within its lexical scope, as in most programming languages. This is true whether the variable is located within the memory of the locale on which the task is running or some other remote locale. There is a performance cost for accessing remote variables, due to the need to transfer them across the network, but the details of managing such communications are handled automatically by the compiler and runtime rather than by relying on explicit user calls, as in MPI. This makes Chapel part of the *Partitioned Global Address Space (PGAS)* family of languages. As a result of its global views of data and control, programming large-scale systems in Chapel tends to be far more similar to traditional desktop programming than it is in MPI, creating the potential for making HPC computation much more accessible to a broader community of programmers.

A leading alternative to MPI for distributed-memory programming is the SHMEM interface, governed by the OpenSHMEM community [5], [6]. SHMEM is similar to MPI in that it relies upon an SPMD programming model and user-specified communication between copies of the program. However, it differs in that its primary routines for data transfer are *one-sided* puts and gets that can make use of RDMA support in cases like the Aries network on Cray® XC™ series systems. Like SHMEM, Chapel also relies on single-sided RDMA, but implements it automatically within the compiler and runtime rather than by relying on the programmer to specify such transfers explicitly. In this respect, Chapel's global view has similar advantages over SHMEM as it does MPI.

Unified Parallel C (UPC) [7] and Fortran 2008, formerly Co-Array Fortran [8], [9], are two prominent examples of PGAS languages that handle communication automatically for the programmer, similar to Chapel. UPC supports 1D arrays that can be distributed across compute nodes in a block-cyclic manner, as well as *wide pointers* that support the creation of distributed data structures. Chapel is similar to UPC in its support for distributed global arrays, but adds support for multidimensional, sparse, and associative arrays to its feature set. Chapel also supports distributions other than block-cyclic and the ability for users to define distributions [10]. Like UPC, Chapel supports unstructured

[2]https://chapel-lang.org

[3]https://chapel-lang.org/docs

distributed data structures, but follows the approach of languages like Java by expressing them using class object references rather than pointers in order to avoid compilation challenges like understanding aliasing in the presence of pointer math.

Fortran 2008 takes a different approach to distributed data structures than UPC and Chapel, permitting variables to be declared with a *co-dimension*. Such co-dimensions are similar to traditional Fortran array dimensions except that they refer to copies of the program, or positions within the SPMD image space. By indexing into the co-dimension, remote copies of a variable can be accessed and communication is induced. This abstraction provides clean language support for communication, yet it requires the programmer to write data structures using a local, per-process view as in MPI. In contrast, Chapel's global-view data structures, like UPC's, make communication syntactically invisible for the sake of code reuse. That said, such communication can still be reasoned about semantically and programmatically. Despite being similar to Chapel in terms of supporting global namespaces, UPC and Fortran both require programs to be written in the SPMD model, and therefore retain some of the challenges of managing local-view control flow as in MPI and SHMEM.

In all of the SPMD models above—MPI, SHMEM, UPC, and Fortran—finer-grained parallelism may be introduced into a program's execution by mixing in additional parallel programming models such as POSIX threads, OpenMP, OpenCL, OpenACC, or CUDA [11], [12], [13], [14], [15]. This is often referred to as *MPI+X* programming, where MPI is used to create per-socket SPMD processes which can then express finer-grained parallelism to leverage other cores or accelerators via additional notations. The chief downside of this approach is that it requires programmers to understand multiple disparate notations and features in order to write and understand parallel programs in this style. In contrast, Chapel's task-based programming model is general enough to support SPMD patterns and local-view programming, yet rich enough to support more general and dynamic parallel patterns. As a result, Chapel removes the need to mix multiple programming models to express coarse- and fine-grain parallelism within a single program.

Chapel's closest siblings are two other languages that were developed under the HPCS program, IBM's X10 [16], [17] and Sun's (later Oracle's) Fortress [18]. X10's features were, in many respects, similar to Chapel's, yet made a much stronger distinction between operations on local versus remote data than Chapel does—in a sense making it more like Fortran 2008 to Chapel's UPC. Chapel chose not to expose such differences syntactically in order to promote code reuse and avoid having to specialize code for the cross-product of local versus remote variables. Fortress shared many high-level thematic goals to Chapel, such as creating a language in which many of the core features are defined as library code. However, Fortress was also more aggressive in many aspects of its design, including support for mathematical notation and units of measure. Neither language seemed to gain the level of user enthusiasm that Chapel has, and active development has essentially wrapped up on both of them.

Among more modern languages, Julia [19] is perhaps the closest to Chapel, since both strive to support parallel programming on large-scale systems using a Python-like language, yet without sacrificing performance. Based on our experiences, while Julia has a much larger user community, programmers from an HPC background tend to favor the way Chapel is engineered close to the iron, as well as its strong foundation for parallelism and locality upon which higher-level features of the language are based. In contrast, Julia seems to take more of a "trust the compiler" approach which arguably leaves expert programmers with no recourse when the compiler gets things wrong. That said, Julia also has tangible advantages over Chapel, such as interoperability with Python and interactive programming via REPLs and notebooks. These are areas in which we expect Chapel to improve in coming years.

III. Language Improvements

By the end of the HPCS program, Chapel's features for parallelism and locality were in fairly good shape. The core features for creating tasks, controlling locality and affinity, and expressing data parallel abstractions were in place and remain largely unchanged to this day. Sections III-A and III-B below describe the primary changes that have been made to these features since version 1.7.

In contrast, Chapel's base language features left a lot to be desired at the end of the HPCS program—and this should not be surprising given the research focus of HPCS and the modest size of the team. That said, these faults were a major concern for early users trying to write real programs, particularly larger ones. As a result, a significant effort was made during the five-year push to improve the base language in terms of robustness and feature completeness. This effort will be summarized in Section III-C.

A. Parallel Language Improvements

1) Task Intents: The primary change to Chapel's parallel features since HPCS is the introduction of *task intents*, also referred to as *forall intents*. This change was adopted to reduce the chances of unintentionally writing race conditions. As an illustration, consider the following naïve parallel loop which attempts to sum the squares of an array's elements while counting the number of positive values:

```
1 var A: [1..n] real;
2 var a2, total: real;
3 var pos: atomic int;
4 forall a in A {
5   a2 = a**2;
6   total += a2;
7   if (a > 1) then pos.add(1);
8 }
```

Under Chapel 1.7, all references to outer-scope symbols from within parallel constructs like this *forall-loop* simply referred to the original variables. Thus, the references to A, $a2$, $total$, and pos within the loop above refer to the original variables declared outside the loop.

For this loop, this results in two races: The first is that the assignment to $a2$ on line 5 causes the loop's tasks to all refer to the same shared $a2$ variable, permitting their values to overwrite one another. The second race is similar, but slightly subtler, and relates to the updates to $total$ on line 6. Because these updates are not synchronized, it's possible that multiple tasks will simultaneously read the same value from $total$, compute their updates, and then write the results back, effectively overwriting the other tasks' updates in a classic *read-read-write-write* race. Note that the update to pos is not problematic due to its atomic nature.

Recognizing the frequency with which new Chapel users were writing races like these, we decided to change Chapel's semantics for tasks, considering outer-scope variables to effectively be "passed into" the tasks, similar to how arguments are passed to procedures. Moreover, we decided to use the same rules that are employed by Chapel's procedure arguments, causing different types to be passed in different ways by default. As a result of these changes, when the program above is compiled with more recent versions of Chapel, compiler errors are generated for the assignments to $a2$ and $total$. This is because scalar values are passed to tasks by `const in` intent (as they are for normal procedures). This prevents the task-local shadow copies that are created by the compiler from being modified within the forall-loop and eliminates the accidental races.

Upon receiving these compiler errors, a user can choose to override the default task intents by adding a *with-clause* to the parallel construct to specify how outer variables should be passed to tasks. For instance, to get the behavior for this example that was originally intended, yet in a parallel-safe way, the user could write the forall-loop as follows:

```
forall a in A with (in a2, + reduce total) {
```

This specifies that each task should get a personal (non-`const`) copy-in of $a2$ that it can modify independently of other tasks, effectively giving each task the scratch variable that it desired. It also says that each task should get a private instance of $total$ whose value will be sum-reduced with those of other tasks' copies as they complete.

Meanwhile, the array A and the atomic variable pos are passed by reference (`ref`) by default, just as they would be for a subroutine taking them by default intent. This permits the forall loop's tasks to read or write the original variables without additional code changes. As with normal arguments, these choices are motivated by the principle of least surprise, to avoid potentially expensive array copies, and to support the common case for types designed for inter-task coordination, like atomic variables.

Note that this change to Chapel does not eliminate races: a parallel loop may still have a race on a shared variable like array A above, or the user may explicitly specify that $a2$ and $total$ should be passed in by `ref`, re-enabling the original races. That said, we have found that this change has significantly reduced the frequency with which such unintentional races occur, as intended. For further information on task intents, refer to the Chapel Language Specification [1] and online documentation.[4]

2) Standalone Parallel Iterators: The other main change to Chapel's parallel features that took place since Chapel 1.7 was to introduce the notion of *standalone parallel iterators* which are invoked when applying a forall-loop to an expression in a non-zippered context, such as the loop over array A in the previous section. Prior to this feature, such cases were handled by invoking the expression's leader-follower iterators, first described at the PGAS 2011 workshop [20]. As noted in that work, this approach added unnecessary complexity and bookkeeping overhead for the degenerate (but common) case of iterating over a single expression. By introducing standalone parallel iterators, we reduced the size of the generated code for such cases while also typically improving performance. For further information on standalone parallel iterators, refer to the parallel iterators primer in Chapel's online documentation.[5]

B. Locality Improvements

Chapel's core features for locality have similarly remained largely unchanged since Chapel 1.7. *Locales* are still used to refer to compute nodes, *on-clauses* are still used to direct computation to specific locales, and *domain maps* are still used to distribute an index set and its arrays across a set of locales. The major change that has taken place in this area has been the introduction of *user-defined locale models* [21] in response to the evolution of compute node architectures in HPC systems.

In more detail, in the HPCS timeframe, Chapel's locales were a type that was built into the language, where all details of managing tasks and memory within a locale were left to the compiler and runtime. This approach made sense in a world where compute nodes were fairly flat and homogeneous. However, as compute nodes with multiple NUMA domains and accelerators have become more common, this "flat" notion of locales became increasingly limited. Where before, one might have been willing to rely on a runtime to map a task or variable to an appropriate processor or memory bank within a compute node, as node architectures become increasingly heterogeneous and hierarchical, that approach seems increasingly naïve.

To this end, we introduced the notion of user-defined locale models to Chapel, enabling advanced users to create

[4]https://chapel-lang.org/docs/1.17/technotes/reduceIntents.html
[5]https://chapel-lang.org/docs/1.17/primers/parIters.html

their own locale definitions to describe compute node architectures that we had not anticipated. Such locale models are often hierarchical in nature, containing *sublocales* within the top-level locales to describe things like NUMA domains, or specific flavors of processors or memory. In this approach, the locale model author creates a set of objects to model the abstract architecture of their compute nodes. In doing so, the user satisfies interfaces that specify how to allocate memory and create tasks for each locale type. Ultimately, we expect that locale models will also specify how remote puts and gets should be implemented, though today these interfaces are still hard-coded between the compiler and runtime. Like Chapel's other *multiresolution features*—parallel iterators [20] and domain maps [10]—these locale models are written using Chapel code.

While most of Chapel's five-year push was skewed toward the "development" side of R&D, the introduction of hierarchical / user-defined locale models constituted one of our most significant research efforts during this period. In recent years, all compilations of Chapel programs have utilized locale models written in Chapel to map the language's requests for memory or tasks to a system's resources. At the time of this writing, prototypical locale models exist for multi-NUMA domain compute nodes and Intel® Xeon Phi™ "Knights Landing" (KNL) processors in addition to the "flat" locale model that we use by default. Further information on user-defined locale models can be found in Chapel's online documentation[6].

C. Base Language Improvements

As mentioned previously, Chapel's base language features lagged significantly behind those for parallelism and locality and were considered a major impediment to practical use. During the five-year push, a great deal of effort was spent improving them to help with the practical use and adoption of Chapel. This section highlights some of the most notable changes.

1) Object-Oriented Features: One particular area that was lacking in Chapel 1.7 was support for Object-Oriented Programming (OOP). While Chapel's design included both records (similar to structs in C/C++) and classes (similar to classes in Java or C#), in many respects, the original design was naïve. For example, Chapel's original design for records turned out to only be sufficient for supporting cases in which fields were restricted to Plain-Old Data (POD) types—*e.g.*, scalar values and other records or arrays of POD type. When records required more sophisticated memory management, as can happen when they store class fields, our original design was too simplistic. Similarly, Chapel's original implementation of class hierarchies, particularly those involving generic classes, was not nearly as robust as an expert programmer would want or need.

[6]https://chapel-lang.org/docs/1.17/technotes/localeModels.html

Since Chapel 1.7, these features have improved dramatically. The original constructor feature that resulted in many of these weaknesses has been replaced with a new *initializer* feature which supports much richer initialization of objects and addresses the lacking record semantics. Bugs related to generics and class hierarchies have been fixed, and our standard domain map hierarchy now makes greater use of these features to help lock them in.

In addition, new features have been added to Chapel's OOP types, such as the ability to create *type methods* that support invoking procedures and iterators directly on a class or record type rather than on objects of that type (similar to *static methods* in C++). Classes and records may now also declare *forwarding fields*, to which methods are dispatched when they cannot be handled by the containing object itself. This provides a means of reusing object methods using the "has a" nature of fields rather than the "is a" approach of inheritance.

One final feature related to OOP, whose development is still underway, is support for a *delete-free* style of programming in which the language and a compiler *lifetime checking* pass help manage memory allocated for class objects. This eliminates the need for programmers to explicitly `delete` their classes and reduces the chances of errors such as memory leaks, double-frees, and use-after-free errors.

2) Namespace Improvements: Though Chapel has always supported namespaces in the form of its *modules*, these features have been improved significantly since Chapel 1.7 by permitting module `use` statements to restrict which symbols are made visible in the current scope. This extension also provides a means for renaming such symbols as they are `use`d. In addition, module-level symbols can now be declared `public` or `private` as a means of restricting which symbols are available through a `use` statement. Finally, enumerated types can now be accessed in an unqualified manner by naming the type in a `use` statement just as a module would be.

3) Error-Handling: In addition to improving existing features, a few new base language features have been added since version 1.7 to improve Chapel's usability in production codes. One major example is error-handling, in which exceptional conditions can be dealt with gracefully as a program is running. Traditionally, Chapel programs have either halted when reaching an error condition, or they have used certain computational patterns to indicate a problem, such as passing error codes back through `out` arguments.

In this effort, we added support for error-handling using an approach inspired in part by Swift's error-handling features. Specifically, Chapel routines can now `throw` errors in addition to returning a value. When calling such routines within production code, the calling routine must wrap the call in `try...catch` blocks to handle the error or throw it further onwards. A `try!` form can be used to halt the program in the event that an error is not handled.

As part of the error-handling effort, Chapel also added support for a `defer` statement which can be used to specify cleanup actions to be taken when a given code block is exited in any way, including throwing an error. For further information on Chapel's error-handling features, refer to the online documentation.[7]

4) Other Base Language Improvements: Beyond the base language improvements described above, a few others are worth noting in this summary:

Strings: In Chapel 1.7, the `string` type was virtually unusable due to memory leaks, bugs, and the lack of a sophisticated string library. Since then, these issues have been vastly improved, making string-based computation viable in Chapel for the first time. In addition, a new uninterpreted string literal has been added, permitting multi-line strings to be embedded directly into a user's source code.

The `void` type: Chapel has also recently introduced a `void` type which is somewhat like that of C's `void`, yet more versatile and designed to support the folding away of code at compile-time. For example, the following record declaration sets the type of its *stop* field to be either `int` or `void` based on the compile-time `param` value *bounded*:

```
record sequence {
  param bounded: bool;
  var start: int;
  var stop: if bounded then int else void;
}
```

As a result, the field will be completely optimized away in cases where *bounded* is `false`, saving space. In addition to eliminating storage, `void` values can also be stored in variables of `void` type, passed to routines, and returned from functions and iterators. This propagation of `void` values through a computation permits users to fold code out of a program at compile-time, resulting in a powerful optimization technique that can be applied in generic programming scenarios.

And More: Beyond the improvements described in this paper, many others have been made to Chapel since version 1.7 in the areas of array support, interoperability improvements, operator precedence, and several others. For far more complete coverage of the improvements made in recent releases, please refer to Chapel's extensive release notes.[8]

IV. Chapel Libraries

Chapel's library support also improved significantly during the five-year push. Chapel 1.7 shipped with around two dozen library modules, which were documented using comments in the source code, if they were documented at all. By contrast, Chapel 1.17 contains 60 library modules, most of which are documented online, and many of which were contributed by developers external to Cray. This increase in libraries was aided in part by Chapel's improved support for tools and language features that help with C interoperation, since many Chapel libraries wrap their C counterparts to avoid reinventing the wheel.

The following list notes some of the more prominent libraries that have been added since Chapel 1.7:

- **Math Libraries:** Chapel now supports `FFTW`, `BLAS`, and `LAPACK` modules which support calls to routines from the familiar community libraries. In many cases, the Chapel interfaces to these libraries are simpler than their C counterparts since Chapel's arrays carry multidimensional information in their descriptors and can therefore pass these values to the C routines without user involvement. Recent releases have also included a new `LinearAlgebra` module which supports cleaner interfaces to linear algebra operations than LAPACK or BLAS.
- **Communication Libraries:** Chapel now supports `MPI` and `ZMQ` (ZeroMQ) modules that provide access to these standard modes of inter-process communication. Chapel's MPI support can either be used to perform message passing between a number of single-locale Chapel programs executing cooperatively, or within a multi-locale Chapel program, as an alternative to relying on the language's global namespace and implicit communication. The ZMQ module supports inter-process message queues and can be used to coordinate between multiple concurrent Chapel programs, to interoperate between Chapel and other languages with ZeroMQ support (like Python), and/or to permit processes running on the login node to coordinate with those running on compute nodes.
- **Parallel / Distributed Patterns:** Chapel has a growing number of library modules that support common patterns for parallel and distributed computing. For example, the `DistributedIters` module supports dynamic, distributed work-sharing iterators that can be used to load balance computations across locales. Distributed collections are supported by the `DistributedBag` and `DistributedDeque` modules. And Chapel programmers can make use of barrier synchronization and futures via the `Barrier` and `Futures` modules, respectively.
- **File System Modules:** Users can interact with the file system using new `FileSystem` and `Path` modules which support core operations like renaming and deleting files along with more interesting cases like iterating in parallel over the files in a directory hierarchy or those that match a given *glob* pattern. Since version 1.7, Chapel has also introduced an `HDFS` module which may be of interest to users who compute with data stored using the Hadoop Distributed File System.

[7] https://chapel-lang.org/docs/1.17/technotes/errorHandling.html

[8] https://chapel-lang.org/releaseNotes.html

Benchmark	Full Problem Size	Reduced 1.7 Problem Size	Compiler Flags for Reference Version	Reference Version Programming Model
HPCC STREAM	32 GB/node	32 GB/node	-O3 -fast-math -funroll-loops -fprefetch-loop-arrays	MPI+OpenMP
HPCC RA	32 GB/node, $N\_U = 2^{31}$	32 GB/node $N\_U = 10,000,000$	"	MPI
PRK Stencil	16 GB/node	2 GB/node	-O3	MPI+OpenMP
ISx	$n = 2^{27}$	$n = 65536$ $perBucketMultiply = 1$	-O3 -mavx	SHMEM

Table I
COMPILATION AND EXECUTION DETAILS FOR BENCHMARKS

- **Other Notable Modules:** Some other notable new library modules in Chapel include:
  - `BigInteger`: provides a nicer interface to Chapel's traditional GMP routines for arbitrary precision integers.
  - `Reflection`: permits users to reason about static aspects of their program code such as whether or not a given call can be resolved or what fields an object contains.
  - `Crypto`: supports various cryptographic routines based on OpenSSL.
  - `DateTime`: enables computations on dates, times, and timezones.
  - `Spawn`: supports spawning subprocesses and interacting with them.
  - `Curl`: provides standard support for data transfers via URLs.

Recent Chapel releases have distinguished between *standard modules*—those that are considered to be a core part of the Chapel distribution—and *package modules*—those that are either more tangential to the language, or which are not yet considered mature enough to be incorporated into the standard set. Chapel's online documentation contains pages describing both standard[9] and package[10] modules.

V. PERFORMANCE OPTIMIZATIONS AND IMPROVEMENTS

If you were to ask a random HPC programmer for their impressions of Chapel, a typical response might be something like "it's that elegant language which doesn't perform very well." And for most of Chapel's history, this characterization has been accurate. However, as a result of the five-year push, Chapel's performance and scalability have now improved to the point that it is competitive with hand-coded C+MPI+OpenMP for a variety of computational styles. In this section, we present recent Chapel benchmark results, comparing to version 1.7 and reference implementations, and highlighting a few key optimizations that helped us narrow the gap.

[9]https://chapel-lang.org/docs/1.17/modules/standard.html
[10]https://chapel-lang.org/docs/1.17/modules/packages.html

A. Experimental Methodology

All performance results in this paper were gathered on Cray XC systems. Scalability results comparing Chapel 1.17 vs. reference versions of benchmarks were gathered on up to 256 compute nodes with 36 cores per node. Due to limited machine availability, Chapel 1.7 vs. 1.17 comparisons and single-node LCALS comparisons were gathered on a distinct XC system with 28 cores per node. Because of the poor performance of Chapel 1.7, these cases were only run from 1–32 compute nodes to avoid wasting system resources. The two systems we used have identical software stacks, identical memory types, and similar Intel "Broadwell" processors. The following table provides a more detailed listing of the hardware and software used to gather our results.

28-core Hardware	Cray XC with Aries interconnect w/ CLE6 28-core (56T) 2.6 GHz "Broadwell" CPUs 128 GB DDR4 RAM
36-core Hardware	Cray XC with Aries interconnect w/ CLE6 36-core (72T) 2.1 GHz "Broadwell" CPUs 128 GB DDR4 RAM
Chapel Software	Chapel 1.7.0 and 1.17.1 GCC 6.3 as back-end compiler 16M hugepages
Reference Software	GCC 6.3 cray-mpich 7.6.2 and cray-shmem 7.6.2

Table I provides experimental details for the scalability studies in this section. Problem sizes for the Chapel 1.17 versus reference scalability studies are given in the first data column. Due to the poor performance of Chapel 1.7, in some cases we had to dial down the problem sizes for its scalability studies so that they would complete in a reasonable amount of time. These sizes are given in the second data column. In all experiments, we ran ISx in its weak-ISO scaling mode. The next column indicates the compiler flags used for the reference versions, based on the Makefiles that ship with them. All Chapel programs were compiled with the `--fast` flag. The final column indicates whether the reference version uses MPI, SHMEM, or MPI+OpenMP. All runs involving Chapel or OpenMP

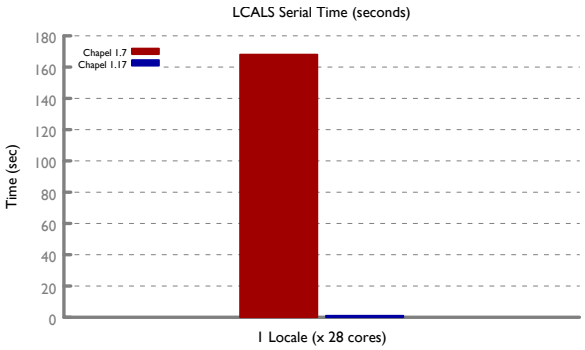

Figure 1. Timings for LCALS serial pressure calc kernel

were run with a process per node and a task per core. Non-OpenMP cases were run with a process per core. Not shown in the table, all LCALS results are for the *long* problem size.

B. Arrays

Arrays are a fundamental building block for HPC applications and a first-class language concept in Chapel. Chapel arrays have a feature set that is significantly richer than arrays in C/C++, or even Fortran. Chapel supports arbitrary indexing (instead of fixed 0- or 1-based indexing), true multi-dimensional arrays, and parallel iteration as well as built-in support for slicing, rank-change, and many other operations. A naïve implementation of these features can incur a significant performance penalty, and that was certainly the case in the 1.7 release of Chapel. Since then, we have implemented several key optimizations which permit Chapel's arrays to perform as well as their C/C++ equivalents. Specifically, we have:

- Implemented a "shifted data" optimization which stores a shifted reference to the start of an array to eliminate the overhead of arbitrary indexing
- Implemented a loop-invariant code motion optimization which hoists array meta-data to avoid dereferences and extra memory accesses compared to C arrays
- Eliminated a multiply in indexing operations for the innermost dimension of array accesses (required for computing offsets into multi-dimensional arrays)
- Improved affinity for parallel operations over arrays
- Made significant improvements to loop code generation
- Eliminated reference counting for arrays

The impact of these optimizations can be seen by examining Chapel's performance for the LCALS benchmark suite. LCALS is a collection of loop kernels that can be used to assess the performance of array operations. Overhead incurred by array features and a general lack of optimizations in Chapel 1.7 led to performance being significantly behind the reference version. Figure 1 shows that Chapel is now over 170× faster than version 1.7 for the serial *pressure_calc* kernel. Appendix A shows the C code generated for this kernel by Chapel 1.7 along with the significantly cleaner code generated by version 1.17. More importantly, Chapel's serial performance is now on par with the C version, and for a majority of the kernels, parallel performance is competitive with the C+OpenMP version as shown in the LCALS graph in Figure 4.

C. Runtime

Chapel's runtime includes support for memory allocation, task-spawning, topology discovery, and communication. Over the past five years, these components have received significant attention in order to increase their speed and scalability.

Chapel 1.7 defaulted to portable, but slow, runtime components. Memory allocations were satisfied by the system allocator, which typically has poor performance for parallel allocations. Chapel tasks were directly mapped to system pthreads with no regard for task placement. This incurred high task creation and switching times as well as suboptimal task affinity. Communication was implemented in a portable manner using GASNet [22] over an MPI substrate, which did not make best use of the underlying network.

In contrast, Chapel 1.17 uses a concurrent and highly scalable memory allocator built on top of jemalloc [23]. Chapel tasks are mapped to user-level qthreads [24] that have extremely fast task creation and switching times as well as affinity-awareness via hwloc [25]. Communication on Cray systems is now mapped directly to a *ugni*-based implementation to provide optimized RDMA operations for gets, puts, active messages, and network atomic operations with very little software overhead.

These runtime optimizations, combined with array optimizations, have led to dramatic improvements for several HPCC benchmarks. STREAM Triad is a synthetic benchmark that measures sustainable memory bandwidth. RandomAccess (RA) measures the rate of random updates to a large, distributed array of integers. As seen in Figure 2, performance for STREAM has improved by over 3×, primarily due to array/task affinity improvements and faster task spawning times. RA performance has improved by several orders of magnitude as a result of mapping puts, gets, and atomics directly to RDMA operations.

Figure 4 shows that STREAM performance is competitive with the reference MPI+OpenMP version, and that RA outscales the reference MPI version at 256 locales. Note that neither Chapel nor the reference version of STREAM were compiled with flags to force non-temporal or "streaming" stores. Switching to Intel or Cray compilers and compiling with the appropriate flags will result in both Chapel and the reference version attaining the "true" peak hardware memory bandwidth, though we tend to run without these flags because we don't believe that bypassing the cache is practical for real applications. In addition, we compare our version of

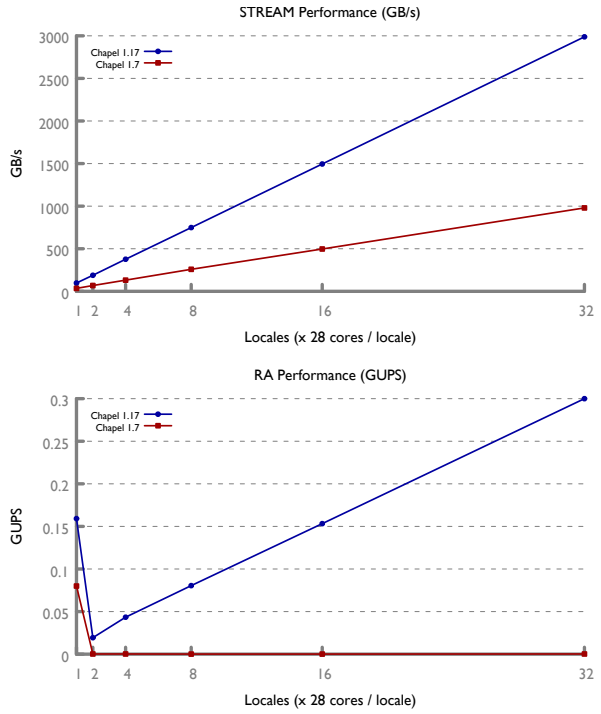

Figure 2. Performance for STREAM and RA benchmarks

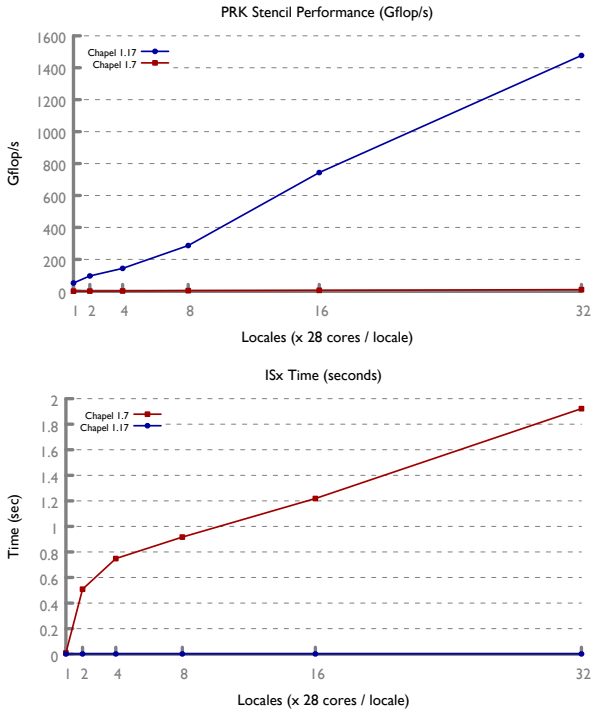

Figure 3. Performance for PRK Stencil and time for ISx benchmarks

RA to the standard HPCC RA. The Chapel implementation does not take advantage of the "lookahead" permitted by the benchmark and instead updates are performed one at a time. The *bucketing* reference implementation takes advantage of the lookahead, while the *no bucketing* version is a more direct comparison to the Chapel implementation. The bucketing version outperforms the Chapel version at lower locale counts, but due to Chapel's 1-sided RDMA support, it outperforms both versions at 256 locales and higher.

D. Communication

We have also implemented several communication optimizations in the compiler and standard library. Array assignments, including slices and strided arrays, are now transformed into large bulk communication operations instead of a communication operation per element. Loop invariant code motion is able to hoist many communication operations out of loop bodies. A remote-value-forwarding optimization bundles remote data with active messages in order to reduce communication. Existing distributions have been improved to reduce communication, and a new stencil distribution has been developed for optimized stencil computations.

As discussed in section II, the Chapel compiler and runtime are responsible for managing communication for remote data. The compiler must detect potentially remote data and insert appropriate runtime calls. We have significantly revamped the analysis to determine which accesses are provably local, dramatically reducing the number of locality checks that Chapel requires at runtime.

These communication optimizations have significantly improved the performance of benchmarks with structured communication patterns such as PRK Stencil and ISx. PRK Stencil is a Parallel Research Kernel that tests the performance of stencil computations, while ISx is a scalable integer sort application punctuated by an all-to-all bucket exchange. Figure 3 shows the dramatic performance improvements that have taken place since Chapel 1.7 for both of these benchmarks. PRK Stencil benefited primarily from array optimizations, use of the stencil distribution, communication optimizations, and array affinity improvements. ISx performance and scalability improved due to bulk array assignment, array optimizations, communication layer improvements, the addition of a scalable barrier implementation, and a wide variety of other improvements. Compared to reference versions in Figure 4, we can see that PRK Stencil performance is on par with the reference MPI+OpenMP version. ISx scales nearly as well as the reference SHMEM version, though raw performance is still behind.

E. Performance Summary and Next Steps

These array, runtime, and communication optimizations, as well as various other performance improvements, have dramatically boosted the performance of Chapel codes. Five years ago, it was nearly impossible to write Chapel code that

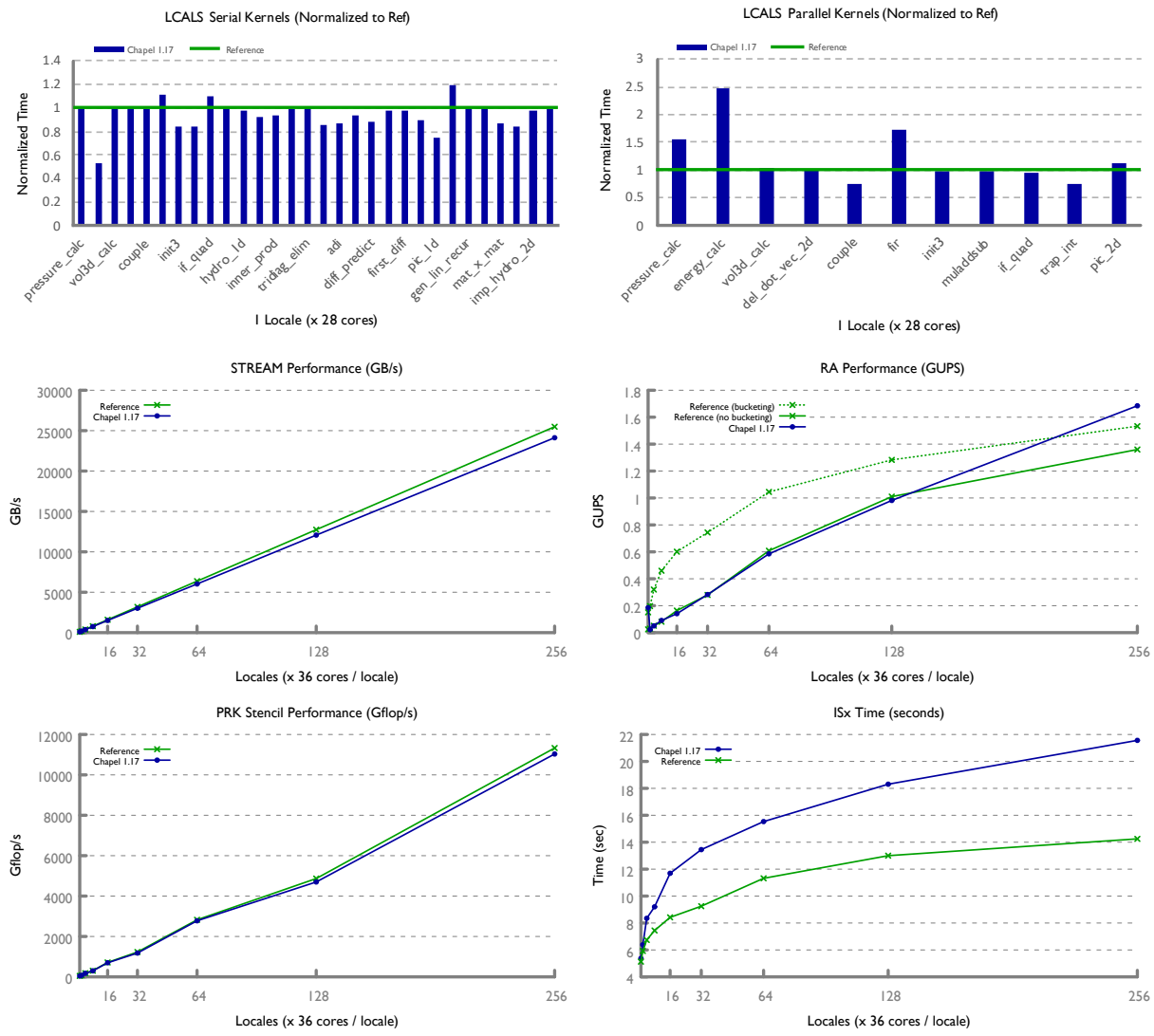

Figure 4. Chapel vs Reference performance for several benchmarks

could compete with a reference version. Generally speaking, today it is possible to write elegant Chapel codes that can perform and scale as well as reference C+MPI+OpenMP codes.

While Chapel performance has improved significantly, it is still possible to write Chapel programs that perform poorly. In some cases, this reflects a problem on the user's part, just as naïve programmers might write MPI programs that perform or scale poorly due to not fully understanding the ramifications of their choices. Yet in other cases, additional optimizations are required in our implementation to make reasonable Chapel code compete with hand-coded approaches. We believe our recent performance improvements demonstrate that there is nothing inherent in Chapel's design that inhibits performance; rather it's just a matter of fleshing out performance optimizations in the compiler, runtime, and library implementation. As such, we encourage Chapel users who are seeing poor performance to contact us so that we can help them work through issues in their code or prioritize additional optimization opportunities in ours.

Currently, we are pursuing optimizations that close the remaining performance gap between ISx implementations in Chapel and SHMEM. Additionally, now that we have demonstrated the ability to compete at modest locale counts, we are starting to study performance at much larger scales. At present, we have performed preliminary performance studies on Edison at NERSC for up to 1,024 locales. Generally speaking, Chapel's performance and scalability trends at these scales are promising. For benchmarks like PRK Stencil, scalability starts to suffer at higher locale

counts, lagging behind the reference version by roughly 10% at 1,024 locales. This is because Chapel's global-view approach involves remote task-spawning in timed sections, yet our remote task-spawning implementation has not yet received significant attention. Currently, remote tasks are spawned one at a time, which has a non-trivial overhead at high locale counts. To address this, we plan on implementing a parallel tree-based spawning mechanism.

VI. Other Improvements

In addition to the major improvements to language features, libraries, and performance described so far, Chapel has seen improvements in a number of other areas over the past five years. This section briefly summarizes a few of the most notable cases.

A. Documentation

At the end of the HPCS program, Chapel had a fairly minimal set of documentation. There was the language specification, a quick-reference sheet summarizing major features, a set of 22 primer examples explaining various aspects of the language, a command-line man page for the compiler, and a large number of text README-style files scattered throughout the release. Today, as a result of a concerted effort, Chapel's documentation consists of over 200 searchable hyperlinked webpages generated from ReStructured Text files and Chapel source code using the Sphinx documentation tool[11]. All Chapel documentation can be found online at https://chapel-lang.org/docs/.

B. Tools

Chapel's tool support was in a similarly primitive state at the end of HPCS, where the primary tools available were (a) highlighting modes for **vim** and **emacs** and (b) an early draft of **chpldoc**—a tool for generating HTML documentation from source comments. While tools remain an area where Chapel would benefit from additional effort, the five-year push produced a number of new tools that benefit Chapel users today:

- **chpldoc**: responsible for producing a significant fraction of the online documentation mentioned in the previous section
- **chplvis**: a GUI-based tool for visualizing the communication and tasking in a Chapel program to help debug performance issues [26]
- **c2chapel**: a script for generating Chapel `extern` declarations from C header files in order to simplify the task of interoperating with C and wrapping existing libraries
- **mason**: a package manager for Chapel inspired by Rust's package manager, Cargo[12]
- **chpltags**: indexes Chapel modules to support quick code searches within **vim** and **emacs**
- **bash** tab completion: provides command-line help when invoking the Chapel compiler, **chpl**

[11] http://sphinx-doc.org/

[12] https://doc.rust-lang.org/cargo/index.html

C. Memory Leaks

Although performance has historically been the primary factor preventing programmers from using Chapel, other users have encountered the fact that Chapel has historically leaked memory, potentially causing long-running programs to fail. There were many sources of these leaks, though the vast majority were caused by core data types like distributed arrays and strings. The root cause of many of these leaks can be blamed on our poor memory management discipline around Chapel's records, as described in Section III-C1. As these features have improved, many of the most significant sources of leaks have been resolved. In other cases, we have put a concerted effort into tracking down and closing sources of leaks.

To understand the magnitude of this effort, consider the following table which summarizes memory leaks in our nightly testing for version 1.7 vs. 1.17. The columns indicate the number of nightly correctness tests we ran on the release date, the fraction of them that leaked memory, and the total amount of memory leaked across all tests:

Chapel version	total tests	leaking tests	% tests that leak	total memory leaked
1.7	4410	3128	70.9%	2137.0 MiB
1.17	8478	302	3.6%	0.237 MiB

As this table shows, the total amount of memory leaked in our nightly testing has decreased by several orders of magnitude over the past five years. But perhaps more importantly, where previously most tests had leaked memory, now just a small fraction of them do, reflecting the effort that has been made to plug leaks. In addition, many past leaks have been due to user-level leaks in the tests themselves, most of which have now been resolved. As mentioned in Section III-C1, we are exploring the development of delete-free features in order to reduce the chances of such user-level leaks in the future.

As of the 1.17 release, a third of the remaining leaked memory is due to a single test, though it is not yet clear whether the blame lies with Chapel or the test itself. Meanwhile, $2/3$ of the leaking tests leak less than 256 bytes each while $1/3$ leak less than 64. As a result, we believe that Chapel is quickly approaching a leak-free state, and in the meantime, we no longer expect leaks to be a serious problem for production codes.

D. Configuration and Installation

Though Chapel has traditionally been considered to be relatively easy to install compared to many open-source packages, several additional steps were taken during the five-year push to make things even simpler and more robust.

One such step was to create two variants of each of our longstanding **setchplenv** shell scripts. The first supports a *quickstart mode* in which many features that rely on third-party packages are disabled, to simplify and accelerate the build. The second variant is designed for *production mode* where these third-party packages are built and used in order to provide better performance and a richer set of libraries.

For most of its history, Chapel has supported the ability to select between various configurations, such as tasking implementations or memory allocators, using `CHPL_`-prefixed environment variables. While Chapel continues to support these variables, it now also permits them to be specified via per-installation or per-user configuration files. These support the creation of default configurations without having to set variables in each shell. New compiler flags also permit these defaults to be overridden on a per-compile basis, as desired.

We also made our build process a bit more uniform with other projects, by adding a **configure** script along with `make install` and `make check` rules to support common steps that many developers expect from open-source builds. Notably, we have continued to avoid using autotools as part of our build process, which many users consider a positive attribute of the Chapel project.

Finally, we have added support for even simpler ways of installing Chapel, including a homebrew package for Mac OS X users and a Docker image for users of Docker.

E. Project Improvements

While most of this paper has focused on technical improvements to the Chapel language and compiler, the five-year push also included many improvements to the process of developing Chapel and interacting within the community. In this section, we mention some of the more notable improvements.

At the end of HPCS, Chapel's development was hosted at SourceForge using a Subversion repository. It was released under the BSD license and a Cray-specific contributor agreement. This made Chapel simple to use, but challenging for many developers to contribute to. User support and community interactions were primarily done through mailing lists. Nightly testing was managed via a number of crontab-driven scripts and email reports.

Since that time, this process has been significantly revamped and modernized. The project switched to the Apache 2.0 license and contributor agreement to reduce barriers for external contributors. Chapel development is now done in a git repository hosted at GitHub. Simple sanity checks are run on each pull request using Travis before they are flagged as being OK for merging. Additional longer-running smoke tests are run after merging to avoid major surprises for users or developers working from the master branch. Nightly testing is now coordinated through Jenkins, which also provides web interfaces for browsing the historical testing logs. The results of nightly performance tests are available online as interactive graphs, permitting developers and users to track improvements and backslides in performance as the project proceeds.[13]

GitHub has also helped organize Chapel's user support, as the team now manages public bug reports and feature requests through GitHub's issue tracker. The team also now monitors StackOverflow questions tagged with "chapel", providing a means for users to get help with questions whose answers may also be of interest to future users. For much of the five-year push, the team has maintained IRC channels for asking questions in real-time, but just this year has modernized that support by upgrading it to a Gitter channel.[14]

Beyond user support, the Chapel project has also improved its online presence through social media. During the five-year push, we established feeds on both Twitter and Facebook where we post regularly to note milestones, events, improvements, and deadlines that may be of interest to the broad Chapel community. We also established a YouTube channel as a hub for collecting videos related to Chapel, primarily in the form of footage from technical talks.

The Chapel community has also grown through the establishment of an annual workshop. In May 2014, the first Chapel Implementers and Users Workshop (CHIUW) was held in conjunction with IPDPS in Phoenix, AZ. Since then, the workshop has been held annually, providing an opportunity for researchers and developers to gather and discuss work they've been doing in, or on, Chapel. Over the course of those five years, dozens of speakers and institutions have been represented, and the number and quality of submissions has grown over time. At the time of this writing, we are just a week away from CHIUW 2018, which will be held at IPDPS 2018 in Vancouver, BC.

For those interested in more information on any of the above, the best place to start is on the Chapel project's webpage which has recently moved from its traditional home at http://chapel.cray.com to https://chapel-lang.org.

VII. User Perspectives

Throughout Chapel's development, we have worked closely with users and prospective users to get their feedback, and to improve Chapel's utility for their computations. In preparing this paper, we sent a short survey to a number of current and prospective Chapel users so that we could convey their perspectives on Chapel in their own words. This section summarizes a few of the responses that we received. We start with two current users of Chapel from the fields of Astrophysics and Artificial Intelligence (AI).

Nikhil Padmanabhan is an Associate Professor of Physics and Astronomy at Yale University, and a self-described

[13] https://chapel-lang.org/perf-nightly.html

[14] https://gitter.im/chapel-lang/chapel

Chapel enthusiast. Nikhil's research is in cosmology, specifically related to using surveys of galaxies to constrain cosmological models. In his response, Nikhil explains that his interest in Chapel developed from a desire to have a lower barrier to writing parallel codes. He explains, "I often find myself writing prototype codes (often serial), but then need to scale these codes to run on large numbers of simulations / datasets. Chapel allows me to smoothly transition from serial to parallel codes with a minimal number of changes." He goes on to say, "Other important issues for me are my 'time to solution' (some measure of productivity vs. performance). Raw performance is rarely the only consideration."

Another notable Chapel user is Brian Dolan, who is the co-founder and chief scientist of Deep 6 AI, a startup focused on accelerating the process of matching patients to clinical trials. Brian and Deep 6 use methods from dozens of fields in pursuit of their product needs, including Natural Language Processing, AI, and Machine Learning. As specific examples, they do network analysis and community detection on graphs as well as reinforcement learning in the form of Deep-Q networks. Brian explains that his team uses Chapel for rapid prototyping of compute-intensive methods that require huge amounts of data and rapid computations, often done on the fly. He explains that he's used Fortran, R, Java, and Python extensively, but that he strongly prefers Chapel "due to the extreme legibility and performance."

Brian goes on to say, "We've now developed thousands of lines of Chapel code and half a dozen open-source repositories for database connectivity, numerical libraries, graph processing, and even a REST framework.[15] We've done this because AI is about to face an HPC crisis, and the folks at Chapel understand the intersection of usability and scalability. It's unrealistic to integrate languages like Fortran into a commercial environment. It's also unrealistic to rely on Python to do HPC." In his responses, Brian also states that this quarter they expect to push their first Chapel application into production.

Turning from current Chapel users to prospective ones, Jonathan Dursi is a Senior Research Associate at the Hospital for Sick Children in Toronto Canada, as well as a Technical Lead for the Canadian Distributed Infrastructure for Genomics. His current computational needs cover a broad range of human genomics, bioinformatics, and medical informatics, such as nanopore sequencing, genome assembly, large-scale biostatistics, and text mining. Jonathan has tracked the Chapel project over a number of years and has blogged about it along with other HPC programming models aimed at improving productivity. He explains, "My interest in Chapel lies in its potential for bioinformatics tools that are currently either written in elaborately crafted, threaded, but single-node C++ code, or in Python. Either has advantages and disadvantages (performance vs. rapid development cycles), but neither has a clear path to cross-node computation, for performance as well as larger memory and memory bandwidth. Chapel has the potential to have some of the best of both worlds in terms of C++ and Python, as well as having a path to distributed memory."

Anshu Dubey, a Computer Scientist at Argonne National Laboratory is another prospective user who has shown interest in Chapel over the past year or so for the purpose of designing and developing Multiphysics software that can serve multiple science domains. She explains that "In Multiphysics applications, separation of concerns and use of high level abstractions is critical for sustainable software. Chapel combines language features that would enable this for clean implementation." She goes on to say, "HPC scientific software is made more complex than it needs to be because the only language designed for scientific work, Fortran, is losing ground for various reasons. Its object-oriented features are clunky and make it nearly as unsuitable as other languages for scientific work. Chapel appears to be parallel and modern Fortran done better, and therefore has the potential to become a more suitable language."

In these surveys, we also asked each person which areas of Chapel they think need the most attention in the years to come, and their responses included faster compilation times, better interoperability with Python, better support for operations on sparse arrays and matrices, and support for irregular distributed data structures like associative arrays and DataFrames.

VIII. Summary and Next Steps

Over the past five years, the Chapel community has implemented a vast number of improvements targeting traditional pain points that have thwarted Chapel's adoption, including: performance, language features, library support, memory leaks, tools, documentation, and community. While Chapel is by no means complete, it has now reached a stable point where it can be used in production by certain programmer profiles, as the responses from our users indicate.

All programming languages that are in use evolve, and our plan is to continue to improve and support Chapel in response to feedback and requests from users. As we look to the future, we expect to continue strengthening the language, seeking specifically to stabilize its core features with the goal of avoiding backwards-breaking changes. Specific areas of focus for the near-term will be the delete-free features of the language, improved sparse array support, and support for partial reductions. We also plan to maintain and improve upon Chapel's portability by targeting the OFI libfabric interface, improving support for GPUs, and increasing our reliance on Chapel's LLVM back-end.

As the world of AI continues to grow, we anticipate putting additional emphasis on features and technologies that will address the pain points of data scientists. As part of this focus, we anticipate making improvements to Chapel's

[15] https://github.com/Deep6AI

interoperability features, focusing particularly on calls to and from Python and C++. We will also be investigating support for Chapel programs within Jupyter notebooks in order to meet data scientists where they're working rather than forcing them to resort to the command-line. In addition, we plan to improve our support for data ingestion from common file formats, and to support richer data structures for storing and manipulating that data, such as DataFrames to complement Chapel's current support for distributed arrays.

We encourage users who are intrigued by Chapel's features and progress to give the language a try, and to share their feedback about ways in which it could better serve their workloads and address their productivity challenges. We also encourage developers to partner with us in making Chapel even more productive than it is today. Looking back on what the Chapel community has accomplished over the past five years, it is exciting to speculate about what the next five might hold!

Acknowledgments

This paper's authors represent the current team of developers working on Chapel at Cray Inc., each of whom has made significant contributions to the accomplishments described in this paper. That said, this paper's results reflect the efforts of a much larger community over the past five years. To that end, we would like to thank our colleagues at Cray, past and present, who helped support and contribute to the five-year push, including Paul Cassella, Benjamin Robbins, Tony Wallace, Kyle Brady, Sung-Eun Choi, Martha Dumler, Tom Hildebrandt, John Lewis, Tom MacDonald, Michael Noakes, and Thomas Van Doren. We'd also like to acknowledge the summer interns and Google Summer of Code students who have worked with our project these past five years: Sean Geronimo Anderson, Ian Bertolacci, Laura Delaney, Andrea Francesco Iuorio, Louis Jenkins, Engin Kayraklioglu, Przemyslaw Lesniak, Cory McCartan, Sarthak Munshi, Sam Partee, Kushal Singh, and Tim Zakian. Finally we'd like to thank a few notable external developers whose contributions have exceeded the norm: Phil Nelson, Nikhil Padmanabhan, and Nicholas Park. See the Chapel website for a full list of contributors to the Chapel project.[16]

We are grateful to Brian Dolan, Anshu Dubey, Jonathan Dursi, and Nikhil Padmanabhan for allowing us to summarize their perspectives in this paper.

We would like to dedicate this paper to the memory of our dear and recently departed colleague, Burton Smith, whose vision and leadership got the Chapel project off to a strong start, and whose past mentorship continues to influence its direction today.

References

[1] The Chapel Team, *Chapel Language Specification (version 0.985)*, Cray Inc., Seattle, WA, USA, April 2018. [Online]. Available: https://chapel-lang.org/docs/1.17/language/spec.html

[2] B. L. Chamberlain, S.-E. Choi, M. Dumler, T. Hildebrandt, D. Iten, V. Litvinov, and G. Titus, "The state of the Chapel union," in *Cray User Group (CUG) 2013*, Napa Valley, CA, May 2013.

[3] B. L. Chamberlain, "Chapel," in *Programming Models for Parallel Computing*, P. Balaji, Ed. MIT Press, November 2015, ch. 6, pp. 129–159.

[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*. High Performance Computing Center Stuttgart (HLRS), June 2015.

[5] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010.

[6] *OpenSHMEM: Application Programming Interface (version 1.4)*, December 2017. [Online]. Available: http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf

[7] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, June 2005.

[8] A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty, "Fortran 2008 coarrays," *SIGPLAN Fortran Forum*, vol. 34, no. 1, pp. 10–30, April 2015.

[9] R. W. Numerich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[10] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov, "Authoring user-defined domain maps in Chapel," in *Cray User Group (CUG) 2011*, Fairbanks, AK, USA, May 2011.

[11] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[12] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, October 2007.

[13] *OpenCL 2.2 Specifications*, May 2018. [Online]. Available: https://www.khronos.org/registry/OpenCL/

[14] *The OpenACC Application Programming Interface (version 1.0)*, November 2011.

[15] *CUDA C Programming Guide (version 9.2.88)*, NVIDIA Corporation, May 2018.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2005, pp. 519–538.

[16] https://chapel-lang.org/contributors.html

[17] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, *X10 Language Specification (version 2.6.1)*, IBM, June 2017.

[18] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, *The Fortress Language Specification, version 1.0*, Sun Microsystems, Inc., March 2008.

[19] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[20] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro, "User-defined parallel zippered iterators in Chapel," in *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS 2011)*, Galveston Island, TX, USA, October 2011.

[21] G. Titus, "Chapel hierarchical locales: Adaptable portability for exascale node architectures," November 2014, poster and talk presented in the Emerging Technologies track of SC14. [Online]. Available: https://chapel-lang.org/presentations-archived.html

[22] D. Bonachea, "GASNet specification v1.1," U.C. Berkeley, Tech. Rep. UCB/CSD-02-1207, 2002, (newer versions also available at http://gasnet.lbl.gov/).

[23] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *BSDCan*, 2006, pp. 1–14.

[24] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*. IEEE, 2008, pp. 1–8.

[25] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a generic framework for managing hardware affinities in HPC applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[26] P. A. Nelson and G. Titus, "Chplvis: A communication and task visualization tool for Chapel," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016)*, May 2016, pp. 1578–1585.

APPENDIX A.
GENERATED CODE IMPROVEMENTS

As mentioned in Section V-B, Chapel's generated code has improved significantly since version 1.7, which has been one contributing factor to the performance improvements reported in this paper. In this appendix, we provide an example of how Chapel's generated code has improved by examining one of the serial loop kernels in LCALS.

Consider the following Chapel loop which implements the *pressure_calc* kernel:

```
for i in 0..#len {
  bvc[i] = cls * (compression[i] + 1.0);
}
```

Using the Chapel 1.7 compiler, the generated code for this loop was as follows, requiring ∼170 seconds to execute:

```
end3 = T22;
call_tmp94 = (ret54 != T22);
T23 = call_tmp94;
while (T23) {
  // get (bvc + i)
  ret58 = bvc;
  ret59 = (ret58)->origin;
  ret_11 = &((ret58)->blk);
  ret_x111 = *(*(ret_11) + 0);
  call_tmp95 = (i3 * ret_x111);
  call_tmp96 = (ret59 + call_tmp95);
  ret60 = (ret58)->factoredOffs;
  call_tmp97 = (call_tmp96 - ret60);
  call_tmp98 = (ret58)->data;
  call_tmp99 = (call_tmp98 + call_tmp97);

  // get (compression + i)
  ret61 = compression;
  ret62 = (ret61)->origin;
  ret_12 = &((ret61)->blk);
  ret_x112 = *(*(ret_12) + 0);
  call_tmp100 = (i3 * ret_x112);
  call_tmp101 = (ret62 + call_tmp100);
  ret63 = (ret61)->factoredOffs;
  call_tmp102 = (call_tmp101 - ret63);
  call_tmp103 = (ret61)->data;
  call_tmp104 = (call_tmp103 + call_tmp102);

  // compute cls * (*(compression + i) + 1.0)
  ret64 = *(call_tmp104);
  call_tmp105 = (ret64 + 1.0);
  call_tmp106 = (cls * call_tmp105);

  // store computation
  *(call_tmp99) = call_tmp106;

  // advance index/induction variable
  call_tmp107 = (i3 + 1);
  i3 = call_tmp107;
  call_tmp108 = (call_tmp107 != end3);
  T23 = call_tmp108;
}
```

In contrast, using version 1.17, the generated code is as follows and runs in ∼1 second:

```
_ic__F1_high = ((int64_t)((len - INT64(1))));
i = INT64(0);
coerce_tmp = (&bvc)->_instance;
coerce_tmp2 = (coerce_tmp)->shiftedData;
coerce_tmp3 = (&compression)->_instance;
coerce_tmp4 = (coerce_tmp3)->shiftedData;
for (i = INT64(0); ((i <= _ic__F1_high));
     i += INT64(1)) {
  call_tmp2 = (coerce_tmp2 + i);
  call_tmp3 = (coerce_tmp4 + i);
  *(call_tmp2) = ((_real64)((cls *
          ((_real64)((*(call_tmp3) + 1.0))))));
}
```

Applying some manual edits to omit hoisted meta-data and clean up variable names, the relation to the original loop is made even clearer:

```
for (i = 0; i <= len-1; i += 1) {
  bvc_p_i = (bvc_p + i);
  compression_p_i = (compression_p + i);
  *(bvc_p_i) = cls * ((*compression_p_i) + 1.0);
}
```