The State of the Chapel Union

Bradford L. Chamberlain, Sung-Eun Choi, Martha Dumler
Thomas Hildebrandt, David Iten, Vassily Litvinov, Greg Titus
Cray Inc.
Seattle, WA 98164
chapel_info@cray.com

***Abstract*—Chapel is an emerging parallel programming language that originated under the DARPA High Productivity Computing Systems (HPCS) program. Although the HPCS program is now complete, the Chapel language and project remain very much alive and well. Under the HCPS program, Chapel generated sufficient interest among HPC user communities to warrant continuing its evolution and development over the next several years. In this paper, we reflect on the progress that was made with Chapel under the auspices of the HPCS program, noting key decisions made during the project's history. We also summarize the current state of Chapel for programmers who are interested in using it today. And finally, we describe current and ongoing work to evolve it from prototype to production-grade; and also to make it better suited for execution on next-generation systems.**

***Keywords*-Chapel; parallel languages; project status**

I. Introduction

Chapel is a parallel programming language that has been developed by Cray Inc. under the DARPA High Productivity Computing Systems (HPCS) program over the past decade. Our primary goal for Chapel under HPCS was to improve programmer productivity on high-end parallel systems. At the same time, we have worked to make Chapel a portable programming model that is attractive for commodity clusters, desktop computing, and parallel programmers outside of the traditional High Performance Computing (HPC) community.

While the success of any new language can never be guaranteed, Chapel has succeeded in making a very positive impression on the parallel programming community. Most programmers who have invested the time to learn about Chapel are very interested in using it once its implementation becomes more robust and optimized. Perhaps more than any previous parallel language, there is a strong desire among a broad base of users to see Chapel succeed and evolve from its current prototype status to become product-grade. Chapel has been downloaded over 6,000 times since its first public release in November of 2008, and it is also available as a standard module on most Cray systems. It has been experimented with by typical HPC users in government and industry, nationally and internationally, as well as by research and educational groups within Computer Science departments.

Cray's entry in the DARPA HPCS program concluded successfully in the fall of 2012, culminating with the launch of the Cray XC30™ system. Even though the HPCS program has completed, the Chapel project remains active and growing. This paper's goal is to present a snapshot of Chapel as it stands today at the juncture between HPCS and the next phase of Chapel's development: to review the Chapel project, describe its status, summarize its successes and lessons learned, and sketch out the next few years of the language's life cycle. Generally speaking, this paper reports on Chapel as of version 1.7.0[1], released on April 18th, 2013.

The rest of this paper is organized as follows: Section II provides a brief history of the Chapel project under HPCS for background. A short introduction to the language's features is given in Section III, for those unfamiliar with Chapel. It is followed by a summary of Chapel's status in Section IV. In Section V, we report on Chapel's greatest successes and lessons learned under the HPCS program. In Section VI, we describe the next steps in Chapel's design and implementation before wrapping up in Section VII.

II. Chapel's Origins

This section provides a historical overview of the Chapel project. For a more detailed history of Chapel, please refer to [1].

A. A Brief History of Chapel

In 2002, at the outset of the HPCS program, the Cray team (codenamed *Cascade*) began considering ways to improve user productivity across the entire system stack: from processor and memory architecture, to network technology and topology, to operating system and runtime responsibilities, to programming environment features. As part of this investigation, the team wrestled with whether or not to develop a new programming language, considering the status quo to be far from ideal from a productivity perspective. Initially, there was concern that a language developed by a lone hardware vendor could not possibly be successful and was therefore not worth the effort; however, we soon realized that many of the most successful languages had been developed in precisely this way and had simply transitioned

[1]Available for download at https://sourceforge.net/projects/chapel/.

to a consortium-based model as they grew in maturity. Having overcome that initial hesitancy, the Chapel project was initiated during the first half of 2003. From the outset, it was determined to be a portable, open-source effort that would also support the ability to leverage Cray architectural features.

From 2003 through 2006, Chapel existed in a reasonably molten state. The team at that time was simultaneously working to define the language and kick off the implementation effort. Much of this period was characterized by wrestling with different language concepts and design philosophies, striving to find a set of features that would work well together. In early summer 2006, a turning point was reached when the current type inference semantics and compiler architecture were formulated. From that point on, progress became much more monotonic and stable. In December 2006, the first Chapel release (version 0.4) was made available on a by-request basis and included support for multitasking in a single-locale (shared memory) setting.

In July 2007, the first multi-locale (distributed memory) Chapel programs began running, and by March 2008, this support was stable enough to be included in the fourth and final request-only release of Chapel (version 0.7). From there, the implementation began focusing on Chapel's data parallel features, both in single- and multi-locale settings. That fall, Chapel was mature enough to run versions of the HPCC Stream and Random Access (RA) benchmarks written with a user-defined *Block* distribution that scaled to hundreds of compute nodes. This milestone earned Chapel a place in the three-way tie with Matlab and UPC/X10 for "most elegant" language in the 2008 HPC Challenge competition. That fall also saw our first public release of Chapel (version 0.8) and our first tutorial at SC08.

The following several years saw Chapel increase in maturity and prominence. In April 2009, Chapel's source code repository moved from being hosted in an invite-only manner at the University of Washington to an open repository at SourceForge, where it continues to be hosted today. A few months later, the Chapel website was launched at http://chapel.cray.com. At that year's HPC Challenge competition, Chapel emerged from the pack and was named the "most productive" language—a title that it has retained each year it has participated since then (2011 and 2012).

From 2009 onward, the number of collaborations and user interactions undertaken around Chapel increased significantly, resulting in the first signs of a broader open-source Chapel community outside of Cray [2], [3], [4], [5], [6]. Chapel also came of age this year in that it began to be considered an expected participant within the HPC space rather than yet another contender doomed to failure. At SC10, the Chapel community met for the first in a series of annual *CHUG* (Chapel Users Group) happy hours. In August 2011, the Chapel logo was unveiled, based on the winning entry in a contest held amongst Cray employees and the broader Chapel community. At SC11, the first Chapel "swag" was developed—a USB stick bearing the Chapel logo. SC11 also saw the first in a series of "Chapel Lightning Talks" sessions, highlighting work being done on Chapel by a half dozen groups outside of Cray.

In October 2012, Chapel completed its final deliverables for the DARPA HPCS program: a scalable execution of the SSCA#2 benchmark on the prototype Cray XC30 system and a final productivity report. In the ensuing time, the team has continued working on improvements to Chapel while simultaneously planning for the next phase in Chapel's development (described in Section VI).

B. Chapel By the Numbers

The following list provides some numbers characterizing aspects of Chapel's history at the time of this writing:

- 21,334: Commits against the Chapel public repository
- 6116: Downloads of Chapel releases from SourceForge
- 1024: Messages sent to the chapel-users mailing list
- 192: Unique, non-Cray subscribers to Chapel mailing lists hosted at SourceForge
- 154: Chapel talks given by the Cray team
  - 80 at conferences and workshops
  - 32 at milestone reviews
  - 18 at universities
  - 16 at government labs
  - 8 in industry settings
- 24: Notable collaborations external to the Cray team
  - 10 with national labs
  - 10 with academic groups
  - 4 with international teams
- 17: Number of Chapel tutorials given
  - 6 at SCxy
  - 5 in Europe
  - 3 at government labs
  - 3 at CUG
- 14: Major releases of Chapel
  - 10 public releases
  - 4 by request only
- 0: Language modifications due to changes in the Cascade architecture during the HPCS program

This last item is notable because Chapel was designed to be a very general, portable parallel programming language; so the fact that it did not need to change as the Cascade architecture evolved can be considered an indicator of success in that regard.

III. Chapel Overview

For readers who are unfamiliar with Chapel, this section provides a brief summary of the language features. Those who are familiar with Chapel can safely skip to the following section. This overview is necessarily high-level and not intended to be a replacement for a thorough introduction to the

language. More specifically, this paper refrains from Chapel code excerpts due to space constraints and the desire to avoid repetition with previous publications. For a more thorough introduction to Chapel's feature set, please refer to previous papers, tutorials, and the language specification [1], [7], [8], most of which can be found at http://chapel.cray.com.

We typically characterize the Chapel language as having features related to the *base language*, *task parallelism*, *data parallelism*, and *locality*. The locality features can further be thought of as providing both higher- and lower-level abstractions for mapping data and computations to machine resources. Each of the following sections provides a brief summary of the features in each concept area.

A. Base Language Features

Chapel's base language features can be thought of as those which are completely unrelated to issues of large-scale parallel computing. In essence, they can be thought of as forming the C-, Fortran-, or Java-level language on which Chapel is based; except that rather than extending an existing language, Chapel was designed from scratch (albeit with influences from a number of successful practical and academic languages).

Chapel's base language features were designed very much around the theme of providing the productivity-oriented conveniences of a scripting language, like Matlab or Python, while also providing the analysis and optimization benefits afforded by a traditional HPC scalar language, like Fortran or C. As an example of this theme, Chapel supports *type inference*, permitting the programmer to optionally omit type specifications for convenience in most contexts. This capability supports the user's ability to sketch out code quickly, as in a dynamically typed scripting language. However, to avoid the execution time overheads associated with such languages, Chapel is statically typed, meaning that the compiler will determine a single type for each variable, argument, and function, possibly by cloning code to deal with multiple type signatures. This compromise between the two traditional approaches provides much of the convenience available in a scripting language while also supporting the compiler's ability to generate optimized scalar code as in C and Fortran. Moreover, it provides a natural means of writing generic code without incurring the syntactic overhead of C++ templates.

Another key feature of the base language is its support for *ranges*, which represent regular sequences of integers. Chapel supports bounded and unbounded ranges, as well as a number of operators and methods on ranges that permit single-dimensional iteration spaces to be represented clearly. Ranges also serve as a crucial building block for higher-dimensional computations using features described in the data parallelism section below.

A third base language feature worth mentioning here is Chapel's support for *iterators*, inspired by the CLU language [9]. Unlike traditional procedures which return a single time during their execution, iterator functions have the ability to *yield* values back to the callsite while logically continuing execution until their return point. Iterators are typically used to drive loops, and can be thought of as providing analogous software engineering benefits for loops as traditional procedures do for straight-line code.

Other Chapel base features include zippered iteration, in which multiple iterators (or iterable values) may be traversed simultaneously; tuples, which support a lightweight means of passing and returning groups of values from procedures; compile-time computation and configuration features; rank-independent loop and array syntax; value- and reference-based object-oriented features; rich function call semantics including overloading, default arguments, and name-based argument matching; features for generic programming; and modules for namespace management.

B. Task-Parallel Features

Chapel's task-parallel features permit users to specify explicit concurrency within their programs. Task parallelism can be defined in an unstructured manner by identifying statements that should be executed concurrently with the originating task. This provides an arbitrary *fire-and-forget* style of task parallelism. Alternatively, tasks may be defined in a more structured manner, via stylized forms of compound statements or loops.

Synchronization between Chapel tasks is typically expressed in a data-centric manner, using *synchronization variables* or *atomic variables*. The former store a full/empty state along with the variable's value; this state prevents reads and writes of the variable from occurring until the full/empty state is in the appropriate mode, supporting producer-consumer synchronization in a natural manner. Chapel's atomic variables are similar to those recently added to C++11, providing support for common atomic operations supported by hardware. These provide support for operations such as mathematical and bitwise operators as well as *compare-and-swap*-style operations that safely execute instantaneously with respect to other tasks.

C. Data-Parallel Features

Chapel's data-parallel features provide users with a way to specify parallel computations at a higher level, by expressing operations on logical data sets such as arrays and iteration spaces. The core concept for Chapel's data parallel features is the *domain*, a language-level representation of an index set based on the *region* concept in ZPL [10]. Domains are used in Chapel to define iteration spaces and to describe the index sets of arrays.

Chapel's standard *rectangular domains* provide support for Fortran 90-style multidimensional rectangular arrays; they are defined using lists of ranges. Chapel also supports *strided domains* whose arrays are stored densely in memory;

sparse domains that represent an arbitrary subset of its parent domain's indices; *associative domains* that provide a hash table-like capability by supporting arbitrary index value types; and *opaque domains* that are designed to support data parallel operations on unstructured data.

Data parallelism within Chapel can be introduced in a number of ways: Explicit data parallelism is typically expressed via *forall loops*, commonly used to iterate over domains and arrays. An implicit form of data parallelism can be achieved by applying scalar functions and operators to array arguments whose element types match the expected scalar argument types. This results in a parallel execution of the operation that is semantically equivalent to a forall loop, yet with a more concise and elegant notation.

In addition to these operations, Chapel's data parallel features also include user-defined reductions and scans [11]; domain slicing to refer to array sub-expressions; and reshaping/reindexing/reallocation operations that modify the size, shape, or indices that define an array.

D. Locality Features

Locality features in Chapel are those used to map computations and data down to the target hardware. The core concept for reasoning about locality in Chapel is the *locale*, a primitive type used to represent locality within the target architecture. For example, on a Cray system, each compute node is typically represented as a locale. Chapel users specify the number of locales to use on the generated executable's command line. Programmers can refer to these locale values and query them symbolically within their program text as a means of referring to the machine resources.

Locales can be targeted in higher- and lower-level ways. The low-level way is to prefix statements with an *on-clause*, which is a means of specifying where the statement should execute. On-clauses can either name a specific locale value directly, or they can use a data-driven form as a means of moving the task to execute wherever the variable in question is stored.

The other concept for mapping Chapel computations to locales is the *domain map*. Domain maps provide a recipe for mapping a domain's indices to the machine resources to be used in storing and computing on its indices. Domain maps also specify how array elements should be mapped to locales and stored in memory, as well as how loops over domains and arrays should be parallelized. An end result of using domain maps is that data parallel operations can be expressed in a manner that is independent of the policies used to map the operations down to the target architecture. This supports a user's ability to switch between vastly different implementation strategies without rewriting their code or thwarting compiler optimizations (the LULESH implementation described in Section V-A4 provides a compelling example of this).

IV. Chapel Status

By and large, the features described in the previous section are implemented and available for use in the current Chapel release. The base language, task-parallel, data-parallel, and locality features have been used to write numerous benchmarks and mini-applications that span a wide variety of computational styles, targeting shared- and distributed-memory execution. The features have also been used extensively within the Chapel implementation itself, constituting over 21,000 Source Lines of Code (SLOC) within Chapel's standard and internal modules.

While the central features of the language are quite solid, a few dark corners remain. In most cases, these represent areas of the language or implementation that failed to receive sufficient attention within the HPCS program due to our need to move on to the next demonstration or milestone. In other cases, they represent feature areas that were intentionally left as future work during HPCS, with the knowledge that we had more than enough on our plate already. In still other cases, user-provided feedback has suggested changes and improvements that we have not yet had time to incorporate.

The following subsections describe incomplete items in Chapel's current implementation and definition. In each section, items are listed in an approximate priority ordering. While this list of outstanding work is nontrivial, it is worth noting that the list of working features is significantly larger.

A. Implementation Issues

1) Performance: In most discussions about Chapel's likelihood of practical adoption within the HPC community, the elephant in the room tends to be the performance of its generated code. While Chapel generates competitive and scalable performance for many benchmarks and computations, in most cases it is not yet a suitable replacement for performance-critical C/Fortran + MPI code.

Many in the HPC community are of the belief that moving to a higher-level language like Chapel necessarily implies a compromise in performance. As argued in more detail elsewhere, we do not agree with this perspective [12], [13]. In most cases, lacks in Chapel's performance today are related less to inherent limitations in its design, and more due to lack of maturity in its implementation. Throughout the HPCS program, the Chapel team focused primarily on making Chapel as correct and feature-complete as possible in order to get early feedback from users and ensure that we were creating a language whose features they would find useful and productive. This focus proved to be the right one, as the feedback we received was quite valuable and has helped make the language far more viable than it otherwise would be. As we often argue, it is easier to improve the performance of a productive parallel language designed for performance than it is to add productivity to one that performs well today yet was not initially designed with productivity in mind.

None of this is to say that achieving competitive performance with Chapel is a lost cause. In many cases, shared-memory parallel performance is competitive with technologies like C and OpenMP today [14], [15]. Distributed memory programs that communicate infrequently or in very unstructured ways can also perform and scale reasonably [16]. Where the current Chapel implementation tends to suffer most today is in distributed memory settings that are amenable to programmer optimization of communication, as that is an area where we have not yet focused.

In any new language, the inherent challenge to achieving optimal performance is that, by nature, it has a multiplicative property in which one poorly implemented feature can undermine the code's performance as a whole. Thus, as one strays from the code paths that received the most attention during the HPCS program, it is often the case that performance falls off as well. As a specific example, stencil codes are a key idiom for many HPC computations, yet were not at all a focus for the HPCS program or HPC Challenge competition. For this reason, Chapel performance is far from optimal for stencils, in spite of the fact that members of our group have achieved stellar results for stencils in higher-level languages in previous work, even beating Fortran+MPI in some cases [17], [10].

We assert that achieving good performance for a broad range of Chapel programs is a question of *when*, not *if*. To this end, we regularly challenge skeptics to identify aspects of Chapel's design that are inherent barriers to achieving Fortran/C levels of performance; the response is typically silence.

It is also worth noting that each user's definition of acceptable performance is relative. To someone currently using Fortran+MPI, Chapel performance is not yet sufficient to be a viable replacement. Yet for a traditional Python or Matlab user, Chapel can represent a dramatic leap forward, by moving from an interpreted, serial environment to one that is compiled and capable of distributed memory parallelism.

Arguments and rationalizing aside, it's important to note that Chapel performance and scalability *are* improving over time—sometimes dramatically. As an example, our 2012 HPC Challenge entries improved upon our 2011 results by $2.6\times$ for STREAM EP, $103\times$ for Global RA, $1741\times$ for HPL, and $600,000\times$ on a $16,384\times$ larger graph for SSCA#2 [16]. Continual performance improvements remain a primary goal for our team, and will be an area of increasing focus going forward.

2) Memory Leaks: Another way in which Chapel can result in suboptimal performance in the current implementation is through compiler-introduced memory leaks. The most common leaks in Chapel programs come from strings, which were implemented hastily with the knowledge that most HPCS demonstrations would only make modest use of strings for things like filenames, output, and error messages. As a result, the current compiler takes some simplifying shortcuts in the implementation of strings and leaks them as a side-effect. Not surprisingly, real-world programmers often have greater computational demands for strings, and for such programs, the memory used by Chapel's leaked strings can be detrimental to performance and long-running programs. We are currently working on re-implementing strings to plug these leaks as we move away from an HPCS focus.

A second source of memory leaks comes from our implementation of distributed arrays. Again, simplifying assumptions were made within the HPCS program given that distributed arrays in the programs we studied tended to be created at program start-up and to live for the entire program; such leaks are not an issue because they are cleaned up by the operating system when the program is torn down. Again, in a production code, it's likely that a user would want to use local-scope distributed array variables. For such programs, these memory leaks clearly need to be plugged in order to avoid problems.

3) Compiler Checks and Error Messages: Another implementation issue that's particularly frustrating to new Chapel programmers relates to the quality of the compiler's semantic checks and error messages. While correct Chapel code compiles and executes as expected, incorrect code can often result in confusing error messages or internal compiler errors, which are not helpful to the novice user. In other cases, semantic checks are not yet fully implemented, such as checking to make sure that constant fields are not reassigned by a user. These types of problems are common to new languages, but represent areas where greater effort is required to make Chapel viable for adoption by new users.

4) Unimplemented Features: A final category of lacks in the current implementation consists of features that are defined for Chapel, but which have never been implemented. For example, the language syntax cleanly supports the definition of *skyline arrays*—arrays of arrays where the size and shape of the array elements may vary from one to the next—but these have never been implemented. Another example involves applying an on-clause to a variable declaration as a means of allocating data remotely without introducing a new lexical scope. These are features that we anticipate adding in the next phase of Chapel's development.

B. Language Design Improvements

Where the previous subsection described issues with Chapel's current implementation, this one enumerates aspects of the current language design that need to be fleshed out better or revisited. Language changes that are more forward-looking in nature are described in Section VI-B.

One area where Chapel's initial design fell short can be seen in its support for user-defined constructors, particularly in the face of generic class hierarchies. The original Chapel design took a fairly lightweight approach to constructors that was sufficient for the HPCS program's needs, but is ultimately a bit simplistic for what an expert object-oriented

programmer would want. Somewhat related, Chapel's definition of copy semantics with respect to records (value classes) was similarly simplistic. This design will also need to be revised for optimal performance and management of user-defined types. At the time of this writing, we have working notes that revise the definitions of these features, and we intend to start implementing them this year.

Another general area for improvement relates to Chapel's definition of iterator functions. While basic serial and parallel iterators work well in the language, a number of extensions are in order that would benefit performance and language semantics. Such changes include: capturing an iterator instance in order to advance it manually; supporting iterators that can generate specific result shapes and index sets; and extending Chapel's parallel iterator interface [18] to support improved performance and semantics.

Other minor areas for improvement in the current language definition include: adding support for fixed-length strings to support in-place allocation of string data; and improvements to Chapel's interoperability features to more easily support additional data types and languages. We are also investigating changes to the semantics of variable references that cross task creation boundaries to help avoid race conditions and improve performance in common cases.

C. Standard Library Improvements

A final area for improvement in the current implementation can be found in Chapel's standard libraries, implemented via Chapel modules. The main area for improvement here is simply that the standard libraries are not particularly extensive compared to other productive languages such as Java, Matlab, and Python. To some extent, this situation will improve as the language continues to grow in popularity, permitting the broader community to more easily contribute library code to back to the implementation. In addition, many of the language improvements described in the previous section will be useful in writing new Chapel libraries and wrapping existing libraries in Chapel interfaces.

One specific area in the standard libraries that would benefit from further Chapel-specific work is to expand the set of standard domain maps to include more options, such as: irregular distributions of rectangular domains, multi-locale implementations of sparse and associative domains, and optimized implementations of opaque domains.

V. Chapel Takeaways

The Chapel experience during the HPCS program was a valuable one, with lessons that are worth sharing. This section recaps some of Chapel's greatest successes in HPCS, along with some valuable lessons learned.

A. Greatest Hits

1) Multiresolution Language Philosophy: Chapel's biggest achievement was perhaps the successful implementation of its *multiresolution philosophy* [19]. This was a major theme in Chapel's design, intended to simultaneously support higher-level ZPL- and HPF-like concepts for productivity while also supporting lower-level, more explicit concepts for the purposes of generality or programming closer to the target architecture.

As part of its multiresolution philosophy, Chapel took the approach of defining higher-level features in terms of the lower-level ones, and of permitting advanced end-users to provide such implementations themselves [20], [21], [18]. This has the benefit of guaranteeing that Chapel's features are compatible between levels, permitting users to move between them within a single program or statement as needed or desired. It also means that any perceived gap in the language's standard set of data structures, memory layouts, distributions, or parallelization strategies can be filled by a motivated user. To ensure that these features were practically useful, the Chapel team used the same framework that an end-user would to implement default and standard features. To that end, the C-style 1D dynamic array built into the Chapel compiler is only used to implement a single array layout: Chapel's default, local rectangular array. All other local and distributed array implementations are built strictly in terms of Chapel arrays.

Chapel's use of the multiresolution approach had some nice benefits beyond those described above. One impact is that data- and task-parallelism are very well-unified in Chapel, particularly in contrast with previous parallel languages that typically only supported one or the other, or grafted the two styles together in an awkward manner. Another impressive result was the addition of OpenMP-style dynamic iterators to the implementation without requiring any changes to the compiler or language [18].

Of course, our adherence to the multiresolution philosophy could also be blamed for some of Chapel's current performance issues: if we had built a small set of array types directly into the compiler and runtime, we would have been able to achieve better performance faster, as in previous work like ZPL [10], [17]. But to be successful in the long-term, we believe that it's far more important to support a flexible parallel framework that supports long-term optimization than to optimize an inflexible approach for the benefit of short-term gains.

2) Distinct Concepts for Parallelism and Locality: Another key philosophical decision in Chapel was to use distinct language concepts for parallelism—"what should execute concurrently?"—versus locality—"where should things be allocated and executed?" While Chapel is certainly not the first language to make this distinction, it has become one of the more successful ones; and in any case, the philosophy deserves much more widespread adoption than it has had, especially given emerging compute node architectures.

In contrast, most adopted distributed memory models utilize a *Single Program, Multiple Data (SPMD)* programming

and execution model. This has the downside of conflating parallelism and locality, since the executable constitutes the basic unit of both properties. As a result, new parallelism cannot be injected into a program without involving an additional parallel programming model or creating a new locality domain. Other approaches, like OpenMP, support much more general parallelism, yet without providing any direct control over locality (a situation that the OpenMP community is presently working to address).

The downside of these traditional approaches is that they tend to require mixing multiple programming models in order to express all the parallelism and locality within an algorithm in a manner that maps well to the parallelism available in the target hardware. In contrast, we believe that languages like Chapel which tease these two concerns apart are much better suited for expressing general parallel programs that target arbitrary parallel hardware within a single, unified language.

3) Portable Design and Implementation: Another of Chapel's successes was its goal of portability, both in the language's design and implementation. It goes without saying that a non-portable language is unlikely to be broadly adopted, particularly given the current diversity of parallel architectures. In spite of this, our programming models tend to embed more assumptions about the target architecture than ideal from a portability perspective. As a result, while programming models tend to be portable within an architectural class, they tend to be limited once too much changes, necessitating the use of hybrid programming models, as discussed previously. A recent example can be seen in the failure of MPI and OpenMP to adequately address GPU programming, necessitating the emergence of new programming models like CUDA (also non-portable, regrettably) and OpenACC [22]. In contrast, we believe that Chapel is far better suited for portability across scales, classes, and generations of machines than its predecessors, through the separation of concerns supported by its multiresolution design and the generality enabled by distinguishing parallelism from locality.

Chapel is portable not only philosophically, but also in its implementation. We have strived very intentionally to build our default implementation upon portable technologies such as C99, POSIX threads, and GASNet [23]. We have also architected our runtime libraries in a modular fashion in order to simplify the support of alternative technologies. It is precisely this design that has permitted us to create custom runtime implementations which take advantage of Cray-specific hardware features [24], [25] for optimized performance [16]. In this sense, Chapel achieves the best of both worlds: portable in design and implementation, yet able to be optimized to a specialized target platform.

4) User Feedback and Interactions: A final Chapel success to note in this report has been its continual interaction with, and responsiveness to, the user community. From the earliest days, we knew that Chapel's practical success would be tied to how useful and compelling users found it. To that end, we have made it a priority to speak publicly about Chapel, to make releases as broadly available as we can manage, and to solicit and act on user feedback to the best of our ability. The net result has been a better language design, with a continual uptick in user interest as the project has progressed. At the outset of the HPCS program, there was a great degree of skepticism in the ability of any language to improve productivity or excite the user community, primarily due to the high-profile failure of HPF during the 1990's [26]. Through good design and continual user interaction, we believe that Chapel has played a significant role in reducing this skepticism and revitalizing the HPC community's interest in novel parallel languages.

As a specific anecdote relating to user interactions, in late 2011–early 2012, we began working with computational scientists at Lawrence Livermore National Laboratory (LLNL) on their LULESH mini-application—an Unstructured Lagrangian Shock Hydrodynamics computation [27]. While the actual computation that LULESH represents is completely unstructured, the benchmark itself uses a structured cubic input in order to simplify the problem setup and verification; however, the implementation is not supposed to take advantage of this fact. The original port of LULESH to Chapel failed to abide by this principle, and so represented the data set using 3D Chapel domains and arrays, reflecting the cubic nature of the problem.

In March of 2012, one of LULESH's originators visited the Chapel group to pair-program with us and help convert the structured implementation into a more general, unstructured one. This goal was achieved in a short afternoon's work, requiring only a few dozen lines of declarations and supporting code to change. This was an eye-opener for the LLNL application expert. While he had heard Chapel's platitudes about separating computation from implementation details before, to see it firsthand on a code of personal importance dramatically improved his impressions of Chapel.

Since that original session, we have gone on to make our LULESH implementation highly configurable: Via compile-line settings, the current implementation can compute the input data set or read it from disk, compile for a shared- or distributed-memory setting, store the data set using structured 3D or unstructured 1D arrays, and represent material subsets using sparse or dense arrays. To support these variations, only a few lines of declarations are involved; the physics computations are completely independent of such details, due to Chapel's successful use of domain maps to support a true separation of concerns. LLNL led a cross-language comparison paper for LULESH in the fall of 2012 which was awarded the "best software paper" distinction for IPDPS 2013 [28].

B. Lessons Learned

While we consider Chapel under HPCS to have been a success, there are also lessons learned along the way that are worth passing along for consideration by future efforts. Among them are the following:

From a language design perspective, if there is something that the development team wants to do, it is likely simply a matter of time before an end-user wants to do it as well. In many respects, we made good choices here, such as permitting users to define array implementations and parallel iteration strategies. And it was precisely these choices that permitted us to implement so much of Chapel within Chapel itself and take advantage of its productivity benefits. But in other respects, we did not open features up to the user and have ultimately regretted it. A primary example is Chapel's lack of support for user-defined coercions and casts. This is something we plan to add to Chapel in the future, both to give users more power and to simplify our implementation by moving these features out of the compiler and into the internal Chapel modules.

While we anticipated that compute nodes would become increasingly hierarchical at the outset of HPCS, we underestimated how soon Chapel would be adversely impacted by these trends. We anticipated an eventual need to make Chapel's locale types hierarchical, but did not correctly anticipate how soon such a feature would be valuable. Once compute nodes began to contain multiple *non-uniform memory access (NUMA)* domains within a shared memory segment, Chapel's performance for simple benchmarks like STREAM Triad was significantly compromised due to our inability to map tasks and memory to specific NUMA domains at the Chapel level. This is a situation we are still affected by in today's implementation, and which we are addressing via the work described in Section VI-B3.

A few prioritization challenges are worth stating, though we have no easy answers: The first relates to the need, in a program like HPCS, to sprint toward various demonstrations and milestones, often necessitating code to be thrown together quickly in order to meet a compressed schedule. Over time, that rapidly written research prototype becomes something you end up living with for an extended period of time. While Chapel's software engineering practices have been quite good compared to many research projects, the fact remains that many aspects of the implementation are less ideal than they ought to be due to lack of time to re-architect and improve certain aspects of the implementation. At times, we have found the opportunity to catch our breath and refactor things into a better state, but other aspects of the implementation continue to betray the rapid way in which they were originally written. This is a situation that we will improve upon during the next phase of Chapel's evolution.

The second tension relates to the HPC community's insatiable need for performance, which can be difficult to meet while pursuing research objectives that could be met more quickly via less specialized solutions. The tension here is to avoid putting off performance optimization for too long while trying to demonstrate advanced capabilities that can be more easily implemented in less optimal ways. It could be argued that Chapel should have focused less on advanced capabilities and productivity in favor of improved performance in order to have a bigger impact on traditional HPC users; however, as argued previously, we believe that our design is better suited for ultimately achieving great performance and productivity than if we had compromised the productivity features simply to get good performance earlier.

The final lesson for this report relates to underestimating the effort required to support early users and collaborators. While both of these communities have been invaluable for Chapel's success so far, we noted a significant change in the rate of our progress in the Chapel implementation once members of the community began taking a more active role in the project as users or developers. The lesson here is to be sure to support early users and collaborators for the value that they bring; but to avoid fooling yourself into believing that the added effort will be negligible.

VI. Next Steps

Having exited the HPCS program, the Chapel team at Cray has turned its attention to the next phase of Chapel's development. In considering future activities, we have been thinking in terms of a five-year timeframe, where the main goal is to evolve Chapel from the prototype implementation developed under HPCS to a product-grade implementation. To this end, we plan to focus on the following areas.

A. Performance Improvements

As noted in Section IV-A, Chapel's performance is consistently cited as the major impediment to the community's faith in Chapel as a practical long-term solution. To this end, it is one of our foremost priorities looking forward. The main areas in the Chapel implementation that require improvement from a performance perspective are:

- Generating cleaner C code to better enable optimizations in the back-end C compiler. Generating cleaner loop structures to enable vectorization is particularly important.
- Generating better code for multidimensional loop nests and array access idioms.
- Optimizing compiler-generated communication to simplify scalar code paths, hide latency, and amortize overheads, similar to our work in the ZPL language [29], [30].
- Plugging compiler-introduced memory leaks.

Our intention is to take a benchmark- and proxy application-based approach to identifying and prioritizing performance fixes, as driven by customer codes of interest. We also intend

to focus on high-profile benchmark suites and demonstrations that may help attract users from outside of the traditional HPC space in order to grow the Chapel community.

B. Language Improvements

Sections IV-A and IV-B listed language features in the current design that are in need of improvements. These will be focus areas in the implementation effort going forward. In addition, we also anticipate more significant changes to the language which we consider necessary for its long-term success. This section describes some of those additions.

1) Base Language: In the base language, one of the biggest lacks in the original language design is support for error-handling, whether through exceptions or some other mechanism. As a result of this lack, when error cases are encountered in Chapel programs today, the most frequent response is to print an error message and halt the program. This approach is obviously heavy-handed and does not permit users to respond to errors in a more resilient fashion, as would be necessary for larger-scale software projects in Chapel.

A second major area, under investigation with the University of Colorado, Boulder, is support for adding *concepts* to the language to support improved type checking of classes and specification of their interfaces. This work is informed by some of the design work that took place for C++11, but extends it to better fit with Chapel's other base language and productivity features.

Other areas for improvement include adding support for namespace control—in particular by supporting declarations as being public or private, and for filtering module 'use' statements to restrict or rename the symbols that are imported into the current scope. We also plan to add the aforementioned support for user-defined casts and coercions between types in Chapel code.

2) Task and Data Parallelism: Within the task parallelism features, one of the main faults in the current design is the lack of a capability to organize tasks into logical *teams*, in order to talk about operations on, or policies for, a subset of tasks without involving every task in the program. Without some means of sorting tasks into sub-groups, the current language works well for small or reasonably homogenous uses of tasking, but tends to fall apart in larger codes that make use of independently-developed coupled components or asynchronous libraries.

Another feature that we have found ourselves wanting is Chapel support for task-private variables and reduction variables, to provide programmability benefits in loop-based idioms, as in OpenMP. While it is possible to program around these features in most cases, built-in support for them would simplify things, both for end-users and optimizing compilers.

A more modest change to the task parallel features that we have discussed and are investigating jointly with Rice University, involves adding support for expression-level tasks and/or futures in the language. Again, this represents an area where the current support does not limit the user's ability to express various parallel patterns, but at times it is less natural or efficient than it could be. Adding support for expression-level tasking would simplify certain idioms while providing the runtime environment with important semantic information that can be useful in optimizing runtime scheduling.

Finally, we are interested in ensuring that Chapel's interoperability features are sufficient to support a productive parallel interoperability story. Traditionally, most interoperability work has focused on single-threaded interoperability; we would like to ensure that it is possible to call from an MPI application out to Chapel and vice-versa as a means of ensuring that HPC users can incrementally transition their existing codes to Chapel. Our hypothesis is that Chapel's user-defined domain map capability should be sufficient to refer to MPI-based data structures as global-view Chapel arrays *in situ*, without copying or redistribution, but this has yet to be demonstrated.

3) Locality Improvements: In the area of locality features, the primary lack that we have identified, and have been working on addressing over the past few years, is support for *hierarchical locales*. This feature is motivated by the observation that compute nodes on parallel machines are becoming increasingly heterogeneous and/or hierarchical (or at least, locality-sensitive) over time. Chapel's current locale concept is great at expressing *horizontal locality* (between compute nodes), but insufficient for talking about *vertical locality* (within a compute node). To that end, we are working on an extension to the locale concept that permits locales to be nested to describe such compute nodes. More importantly, we are permitting these locale structures to be defined using Chapel, as a means of moving the role of modeling and targeting an architecture out of the compiler and runtime, and increasingly into Chapel code.

C. Portability Improvements

One key application of hierarchical locales over the next few years will be to target compute nodes containing GPUs or Intel MIC processors [31]. In addition to the work required to model these architectures using the hierarchical locale framework, we will also need to extend the Chapel compiler's back-end to generate code and idioms that can effectively use these architectures, for example by generating OpenACC or CUDA. A previous collaboration with UIUC demonstrated the possibility of targeting GPUs with Chapel [2]; over the next few years, we intend to merge these ideas with the hierarchical locales work in order to support accelerator-based node architectures as first-class citizens within Chapel.

D. Community Engagement and Development

Over the next several years, we plan to continue supporting and expanding the Chapel community through ongoing talks, tutorials, user interactions, and collaborations. As mentioned earlier, one focus area will be to seek ways of growing the Chapel community to engage non-HPC communities interested in parallelism, such as mainstream, open-source, and *big data* programmers. Another aspect of community development will involve exploring plans for moving Chapel from a Cray-dominated effort to one that utilizes a collaborative, community model of governance and development.

E. Research Directions

In addition to work items related to making the current definition of Chapel more robust, there are numerous research topics that could expand Chapel's utility and applicability, if successful. Examples include: support for predicated task creation and conveying that information to a runtime task scheduler; investigation of work-stealing/load-balancing tasking runtimes; resilience features including fault-tolerant domain maps; extending Chapel domain maps to support out-of-core computations; energy-aware language features; tool support for Chapel, including debuggers and performance analysis tools; use of Chapel for non-HPC parallel computations such as *big data* calculations; and investigations into supporting parallel domain-specific languages (DSLs), either by targeting Chapel as a back-end language, or by embedding DSLs within it directly. While most of these topics are not part of our short-term roadmap, each of them suggests a direction that could potentially increase Chapel's utility, given the appropriate funding vehicle or collaboration.

VII. Summary

To summarize, Chapel has had a good run under DARPA HPCS and has benefitted greatly from the long-term nature of the program. Through its design, development, and demonstrations to date, it has served to overcome much of the skepticism within the HPC community about the potential value and utility of new parallel programming languages. As it currently stands, Chapel is a useful prototype, yet one that requires additional work in order to be valuable to real users. Our intention is to address these shortcomings over the next several years by improving Chapel's performance, filling in features that are incomplete or missing, ensuring that Chapel works on the next generation of node architectures, and continuing to grow the Chapel user and developer communities. With these improvements, we believe that Chapel has an excellent chance of becoming the language to break the historical streak of parallel languages that are intriguing yet ultimately unsuccessful. A big part of this challenge will involve changing the tone of the conversation from "How will Cray ever manage to make Chapel successful?" to "How will we, as a community, do so?" As always, we welcome collaborations and supporting activities from like-minded teams with an interest in contributing to the success of the effort.

Acknowledgments

The Chapel team at Cray Inc. would like to thank the external and past members of the project over its history and to acknowledge the role that they have played in helping Chapel reach its current state. We'd also like to thank the Chapel user community for its ongoing support of the language and role in helping to improve its utility. Finally, we'd like to thank DARPA for creating a program as long-term and visionary as HPCS to permit a language like Chapel to be developed; the program's consistent stability played a huge role in advancing Chapel in ways that smaller, shorter-term programs would not have been able to as effectively.

References

[1] B. L. Chamberlain, "Chapel," in *A Brief Overview of Parallel Programming Models*, P. Balaji, Ed. MIT Press, 2014 (expected), (draft preprint available as *A Brief Overview of Chapel* at http://chapel.cray.com/papers/BriefOverviewChapel.pdf).

[2] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzaran, and D. Padua, "Performance portability with the Chapel language," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012, pp. 582–594.

[3] S. Sridharan, J. S. Vetter, B. L. Chamberlain, P. M. Kogge, and S. J. Deitz, "A scalable implementation of language-based software transactional memory for distributed memory systems," Future Technologies Group, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA, Tech. Rep. FTGTR-2011-02, May 2011.

[4] K. B. Wheeler, R. C. Murphy, D. Stark, and B. L. Chamberlain, "The Chapel tasking layer over Qthreads," in *Cray User Group (CUG) 2011*, Fairbanks, AK, USA, May 2011.

[5] A. Sanz, R. Asenjo, J. Lopez, R. Larrosa, A. Navarro, V. Litvinov, S.-E. Choi, and B. L. Chamberlain, "Global data re-allocation via communication aggregation in Chapel," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2012, pp. 235–242.

[6] B. Ren, G. Agrawal, B. Chamberlain, and S. Deitz, "Translating Chapel to use FREERIDE: A case study in using an HPC language for data-intensive computing," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011, pp. 1242–1249.

[7] The Chapel Team, *Productive Programming in Chapel: a Next-Generation Language for General, Locality-Aware Parallelism*, University of Bergen, Bergen, Norway, April 2013, tutorial slides, available at: http://chapel.cray.com/tutorials-archives.html.

[8] ——, *Chapel Language Specification (version 0.93)*, Cray Inc., Seattle, WA, USA, April 2013, http://chapel.cray.com/spec/spec-0.93.pdf.

[9] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in CLU," *Communications of the ACM*, vol. 20, no. 8, pp. 564–576, August 1977.

[10] B. L. Chamberlain, "The design and implementation of a region-based parallel language," Ph.D. dissertation, University of Washington, November 2001.

[11] S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder, "Global-view abstractions for user-defined reductions and scans," in *Eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. ACM Press, 2006, pp. 40–47.

[12] B. L. Chamberlain, "Myths about scalable parallel programming languages part 1: Productivity and performance," *IEEE Technical Committee on Scalable Computing (TCSC) blog*, April 2012, https://www.ieeetcsc.org/activities/blog/myths_about_scalable_parallel_programming_languages_part1.

[13] ——, "Myths about scalable parallel programming languages part 6: Performance of higher-level languages," *IEEE Technical Committee on Scalable Computing (TCSC) blog*, September 2012, https://www.ieeetcsc.org/activities/blog/Myths_About_Scalable_Parallel_Programming_Languages_Part_6:_Performance_of_Higher-Level_Languages.

[14] H. Burkhart, M. Sathe, M. Chisten, M. Rietmann, and O. Schenk, "Run, stencil, run! HPC productivity studies in the classroom," in *6th Conference on Partitioned Global Address Space Programming Models (PGAS'12)*, Santa Barbara, CA, USA, October 2012.

[15] N. Dun and K. Taura, "An empirical performance study of Chapel programming language," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, 2012, pp. 497–506.

[16] B. Chamberlain, S.-E. Choi, M. Dumler, T. Hildebrandt, D. Iten, V. Litvinov, G. Titus, C. Battaglino, R. Sobel, B. Holt, and J. Keasler, *Chapel HPC Challenge Entry: 2012*, Salt Lake City, UT, USA, November 2012, presentation slides available at http://hpcchallenge.org/presentations/sc2012/ChapelHPCC2012.pdf.

[17] B. L. Chamberlain, S. J. Deitz, and L. Snyder, "A comparative study of the NAS MG benchmark across parallel languages and architectures," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (Supercomputing '00))*. Washington, DC, USA: IEEE Computer Society, 2000.

[18] 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.

[19] B. L. Chamberlain, "Multiresolution languages for portable yet efficient parallel programming," October 2007, http://chapel.cray.com/papers/DARPA-RFI-Chapel-web.pdf.

[20] 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.

[21] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined distributions and layouts in Chapel: Philosophy and framework," in *2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, June 2010.

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

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

[24] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: a scalable HPC system based on a Dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 103:1–103:9.

[25] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini system interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 83–87.

[26] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of High Performance Fortran: an historical object lesson," in *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 7–1–7–22.

[27] "Hydrodynamics challenge problem," Lawrence Livermore National Laboratory, Livermore, CA, USA, Tech. Rep. LLNL-TR-490254, (available at https://computation.llnl.gov/casc/ShockHydro/LULESH-files/spec.pdf).

[28] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang *et al.*, "Exploring traditional and emerging parallel programming models using a proxy application," *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, May 2013.

[29] S.-E. Choi and L. Snyder, "Quantifying the effects of communication optimizations," in *Proceedings of the 1997 International Conference on Parallel Processing*, 1997, pp. 218–222.

[30] B. L. Chamberlain, S.-E. Choi, and L. Snyder, "A compiler abstraction for machine independent parallel communication generation," in *In Tenth International Workshop on Languages and Compilers for Parallel Computing (LCPC'97)*. Springer-Verlag, 1997, pp. 261–76.

[31] *Intel Xeon Phi Coprocessor System Software Developers Guide*, Intel Corporation, May 2013.