Interdisciplinary Postgraduate Program in Advanced Computer and Communication Systems
Parallel and Distributed Systems Course

George Bisbas, January 2018

Chapel, the Cascade High Productivity Language

gmpismpas@ece.auth.gr

Abstract

This report is a short presentation for Chapel the Cascade High Productivity Language, in order to fulfil the requirements of the 3rd project in Parallel and Distributed Systems Course. Chapel is a parallel programming language developed by Cray.[3] It is being developed as part of the Cray Cascade project, a participant in DARPA's High Productivity Computing Systems (HPCS) program, which had the goal of increasing supercomputer productivity by the year 2010. It is being developed as an open source project, under version 2 of the Apache license.

Figure 1: Cray Chapel Logo.

Important note

This short report intends to provide reader with a first introduction concerning Chapel's features. No code samples were given in this report in order to give user the opportunity to begin from a level he thinks is the best for him. For more, visit Chapel's pioneer in Chamberlain (2013) or Chamberlain (2015) and for more visit Dun and Taura (2012).

Introduction

Chapel is an emerging parallel programming language designed for productive scalable computing. Chapels primary goal is to make parallel programming far more productive, from multicore desktops and laptops to commodity clusters and the cloud to high-end supercomputers. Chapels design and development are being led by Cray Inc. in collaboration with academia, computing centers, and industry. Chapel is being developed in an open-source manner at GitHub under the Apache v2.0 license and also makes use of other third-party open-source packages under their own licenses. Chapel emerged from Crays entry in the DARPA-led High Productivity Computing Systems program (HPCS). It is currently being hardened from that initial prototype to more of a product-grade implementation.

A Brief History of Chapel

DARPAs HPCS program was launched in 2002 with five teams, each led by a hardware vendor: Cray Inc., Hewlett-Packard, IBM, SGI, and Sun. The program challenged the teams to develop technologies that would improve the productivity of High Performance Computing users in the areas of performance, portability, programmability, and robustness. The vendors were encouraged to reconsider all aspects of their system stack with the goal of delivering technologies that would be revolutionary and distinct from their established roadmap. Along with changes to their processor, memory, and network architectures, the vendor teams also proposed new and enhanced software technologies, including novel programming languages. In 2003, the HPCS program transitioned to phase II, and a programmatic downselect occurred, enabling the Cray, IBM, and Sun teams to pursue their proposed research plans. At the outset of this phase, the initial designs of the new programming languages began to emerge, with the Cray team pursuing the Chapel language, IBM starting work on X10, and Sun (now Oracle) developing Fortress. Crays HPCS project was termed Cascade after the prominent mountain range just east of its corporate headquarters in Seattle. The project was led by Burton Smith, Chief Scientist of Cray at the time. Though he believed that existing HPC programming models were a productivity limiter for high-end systems, Burton was initially hesitant to pursue a new programming language under HPCS, due to skepticism about whether languages designed by lone hardware vendors could be successful. He soon reconsidered this position, however, after an enumeration of well-established programming languages in both HPC and mainstream computing revealed that most of them had orig-

inally been developed by a single hardware vendor. In most cases, the key to a languages long-term success involved a transition to a broader, more community-oriented model at an appropriate point in its life cycle. In JanuaryFebruary 2003, the Cascade team announced its intention to pursue a new language at various HPCS reviews and meetings. Work on Chapel began in earnest that year under the leadership of David Callahan. The Chapel language took its name as an approximate acronym for Cascade High Productivity Language , coined by Callahan. The team generally felt lukewarm-to-negative about the name, in large part due to its possible religious implications. However, nobody came up with a preferable alternative quickly enough, and the name stuck. When asked about it, team members would occasionally quip, Well wait until weve gotten the language to a point that were happy with it and then switch to a truly great name.

Guiding Principles

The following four principles guided the design of Chapel:

1. General parallel programming
2. Locality-aware programming
3. Object-oriented programming
4. Generic programming

The first two principles were motivated by a desire to support general, performance-oriented parallel programming through high-level abstractions. The second two principles were motivated by a desire to narrow the gulf between high-performance parallel programming languages and mainstream programming and scripting languages.

General Parallel Programming

First and foremost, Chapel is designed to support general parallel programming through the use of high-level language abstractions. Chapel supports a global-view programming model that raises the level of abstraction in expressing both data and control flow as compared to parallel programming models currently in use. A global-view programming model is best defined in terms of global-view data structures and a global view of control. Global-view data structures are arrays and other data aggregates whose sizes and indices are expressed globally even though their implementations may distribute them across the locales of a parallel system. A locale is an abstraction of a unit of uniform memory access on a target architecture. That is, within a locale all threads exhibit similar access times to any specific memory address. For example, a locale in a commodity cluster could be defined to be a single core of a processor, a multicore processor, or an SMP node of multiple processors. Such a global view of data contrasts with most parallel languages which tend to require users to partition distributed data aggregates into per-processor chunks either manually or using language abstractions. As a simple example, consider creating a 0-based vector with n elements distributed between p locales. A language that supports global-view data structures, as Chapel does, allows the user to declare the array to contain n elements and to refer to the array using the indices $0...n1$. In contrast, most traditional approaches require the user to declare the array as p chunks of n/p elements each and to specify and manage inter-processor communication and synchronization explicitly (and the details can be messy if p does not divide n evenly). Moreover, the chunks are typically accessed using local indices on each processor (e.g. , $0..n/p$), requiring the user to explicitly translate between logical indices and those used by the implementation. A global view of control means that a users program commences execution with a single logical thread of control and then introduces additional parallelism through the use of certain language concepts. All parallelism in Chapel is implemented via multithreading, though these threads are created via high-level language concepts and managed by the compiler and runtime rather than through explicit fork/join-style programming. An impact of this approach is that Chapel can express parallelism that is more general than the Single Program, Multiple Data (SPMD) model that todays most common parallel programming approaches use. Chapels general support for parallelism does not preclude users from coding in an SPMD style if they wish. Supporting general parallel programming also means targeting a broad range of parallel architectures. Chapel is designed to target a wide spectrum of HPC hardware including clusters of commodity processors and SMPs; vector, multithreading, and multicore processors; custom vendor architectures; distributed-memory, shared-memory, and shared address-space architectures; and networks of any topology. Our portability goal is to have any legal Chapel program run correctly on all of these architectures, and for Chapel programs that express parallelism in an architecturally-neutral way to perform reasonably on all of them. Naturally, Chapel programmers can tune their code to more closely match a particular machines characteristics

Locality-Aware Programming

A second principle in Chapel is to allow the user to optionally and incrementally specify where data and computation should be placed on the physical machine. Such control over program locality is essential to achieve scalable performance on distributed-memory architectures. Such control contrasts with shared-memory programming models which present the user with a simple flat memory model. It also contrasts with SPMD-based programming models in which such details are explicitly specified by the programmer on a process-by-process basis via the multiple cooperating program instances.

Object-Oriented Programming

A third principle in Chapel is support for object-oriented programming. Object-oriented programming has been instrumental in raising productivity in the mainstream programming community due to its encapsulation of related data and functions within a single software component, its support for specialization and reuse, and its use as a clean mechanism for defining and implementing interfaces. Chapel supports objects in order to make these benefits available in a parallel language setting, and to provide a familiar coding paradigm for members of the mainstream programming community. Chapel supports traditional reference-based classes as well as value classes that are assigned and passed by value.

Generic Programming

Chapels fourth principle is support for generic programming and polymorphism. These features allow code to be written in a style that is generic across types, making it applicable to variables of multiple types, sizes, and precisions. The goal of these features is to support exploratory programming as in popular interpreted and scripting languages, and to support code reuse by allowing algorithms to be expressed without explicitly replicating them for each possible type. This flexibility at the source level is implemented by having the compiler create versions of the code for each required type signature rather than by relying on dynamic typing which would result in unacceptable runtime overheads for the HPC community.

Chapel Feature Overview

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

Base Language Features

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

Syntax

Chapels syntax was designed to resemble Cs in many respects, due to the fact that most currently-used languages, including C++, Java, C #, Perl, and Python, are generally C-based. Like C, Chapel statements are separated by semicolons and compound statements are defined using curly brackets. Most Chapel operators follow Cs lead, with some additional operators added; Chapels conditionals and while-loops are based on Cs; and so forth. In other areas Chapel departs from C, typically to improve upon it in terms of generality or productivity. One of Chapels main syntactic departures can be seen in its declarations which use more of a Modula-style left-to-right, keyword-based approach.

Basic Types

Chapels basic scalar types include boolean values (bool), signed and unsigned integers (int and uint), real and imaginary floating point values (real and imag), complex values (complex), and strings (string). All of Chapels numeric types use 64-bit values by default, though users can override this choice by explicitly specifying a bit width. For example, uint(8) would specify an 8-bit unsigned integer. All types in Chapel have a default value that is used to initialize variables that are otherwise uninitialized. Numeric values default to zeroes, booleans to false, and strings to empty strings. Chapel supports record and class types, each of which supports objects with member variables and methods. Records are declared using the record keyword and result in in-place memory allocation. Classes are declared using the class keyword and use heap-allocated storage. Records support value semantics while classes support reference semantics. For example, assigning between variables of record type will result in a copy of the record members by default. In contrast, assigning between variables of class type results in the two variables aliasing a single object. Records can be thought of as being similar to C++ structs while classes are similar to Java classes. Chapel also supports tuple types that permit a collection of values to be bundled in a lightweight manner. Tuples are useful for creating functions that generate multiple values, as an alternative to adopting the conventional approach of returning one value directly and the others through output arguments. Chapel also uses tuples as the indices for multi-dimensional arrays, supporting a rank-independent programming style

Range and Array Types

Another built-in type in Chapel is the range, used to represent a regular sequence of integer values. For example, the range $1..n$ represents the integers between 1 and n inclusive, while $0..$ represents all of the non-negative integers. Chapels ranges tend to be used to control loops, and also to declare and operate on arrays. Ranges support a number of operators including intersection ([]), prefix/suffix selection (#),

striding (by), and setting the alignment of a strided range (align).

Type Inference

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

For-loops and Iterators

Chapels for-loops are different than Cs, both syntactically and semantically. In Chapel, for-loops are used to iterate over data structures and to invoke iterator functions. Chapels for-loops declare iteration variables that represent the values yielded by the iterand expression . These variables are local to a single iteration of the loops body. In addition to looping over standard data types, Chapel programmers can write their own iterator functions that can be used to drive for-loops.

Other Base Language Features

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

https://chapel-lang.org/papers/BriefOverviewChapel.pdf

Task Parallel Features

All parallelism in Chapel is ultimately implemented using tasksunits of computation that can and should be executed in parallel. All Chapel programs begin with a single task that initializes the programs modules and executes the users *main()* procedure. This section provides an overview of Chapels concepts for creating tasks and synchronizing between them.

Unstructured Task Parallelism

The simplest way to create a task in Chapel is by prefixing a statement with the begin keyword. This creates a new task that will execute the statement and then terminate. Meanwhile, the original task goes on to execute the statements that follow. Tasks in Chapel are anonymous, so there is no direct way to name a task directly. The two ways in which a user can check for task completion are through the sync keyword or by synchronizing with it through shared variables.

The Sync Statement

Chapels sync keyword prefixes a statement and causes the task encountering it to wait for all tasks created within the statements dynamic scope to complete before proceeding. As an example, the use of the sync statement in the following code will wait for all the tasks generated by a recursive binary tree traversal to complete before the original task continues. The *sync* statement is a big hammer. For finer-grain interactions between tasks, programmers should use special variable types that support data-centric synchronization. Chapels synchronization and atomic variable typesdescribed in the following sections.

Synchronization Variables

A Chapel synchronization variable is like a normal variable, except that in addition to storing its value, it also stores a full/empty state that is used to guard reads and writes. As mentioned in Section 9.1.2, this concept was adopted from the similar Tera MTA and Cray XMT features. By default, a read of a synchronization variable blocks until the variable is full, reads the value, and leaves the variable in the empty state. Similarly, a write blocks until the variable is empty, writes the new value, and then leaves it full.

In addition to the default read/write semantics, synchronization variables support a number of methods that permit other modes of reading/writing their values. For example the *readFF()* method provides a way to read a synchronization variable, blocking until it is full, but leaving it full rather than empty. Similarly, *readXX()* permits the task to peek at a synchronization variables value regardless of the full/empty state. In addition to providing a controlled way of sharing data, synchronization variables also play an important role in defining Chapels memory consistency model. Typical Chapel variables are implemented using a relaxed memory consistency model for the sake of performance which makes them an unreliable choice for coordinating between tasks. By contrast, loads and stores cannot be reordered past synchronization variable accesses, which also serve as memory fences. This permits synchronization variables to be used as a means of coordinating data sharing for larger, more relaxed data structures.

Chapel supports a variation of synchronization variables that are called single-assignment variables. They are almost identical except that once their full/empty state is set to full, it can never be emptied. As a result, default reads of single-assignment variables use the *readFF()* semantics. Single-assignment variables (and synchronization variables, for that matter) can be used to express future-oriented parallelism in Chapel by storing the result of a begin statement into them.

Atomic Variables

Chapel also supports data-centric coordination between tasks using atomic variables. These are variables that support a set of common atomic operations which are guaran-

teed to complete without another task seeing an intermediate or incomplete result. Chapels atomic variables are modelled after those being incorporated into the ISO C1x standard, and benefit from the design work done there.

Structured Parallelism

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

Data Parallelism

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

Getting started

A Chapel version of the standard hello, world computation is as follows:

```
writeln("hello, world");
```

This complete Chapel program contains a single line of code that makes a call to the standard `writeln` function. In general, Chapel programs define code using one or more named modules, each of which supports top-level initialization code that is invoked the first time the module is used. Programs also define a single entry point via a function named main. To facilitate exploratory programming, Chapel allows programmers to define modules using files rather than an explicit module declaration and to omit the program entry point when the program only has a single user module. Chapel code is stored in files with the extension **.chpl**. Assuming the *hello world* program is stored in a file called hello.chpl, it would define a single user module, *hello*, whose name is taken from the filename. Since the file defines a module, the top-level code in the file defines the modules initialization code. And since the program is composed of the single hello module, the main function is omitted. Thus, when the program is executed, the single *hello* module will be initialized by executing its top-level code thus invoking the call to the `writeln` function.

To compile and run the *hello world* program, execute the following commands at the system prompt:

```
> chpl hello.chpl
> a.out
```

The following output will be printed to the console:

```
hello, world
```

References

Chamberlain, B. L. (2013). A Brief Overview of Chapel. *Cray*, 1(January):0–20.

Chamberlain, B. L. (2015). Chapel. *Programming models for parallel computing*, 2015:129–159.

Dun, N. and Taura, K. (2012). An empirical performance study of chapel programming language. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pages 497–506.