CHAPEL BY EXAMPLE

IMAGE PROCESSING

Revised for the April 2018 Chapel 1.17 Release

Table of Contents

INTRODUCTION

A few weeks ago (in April 2015) we found a post on the web that, in the middle and almost as an aside, praised several lesser-known languages that were trying to tackle parallel computing. Always alert to random strong recommendations from strangers signaling something interesting, we looked at each. One of these languages was Chapel, developed by a team at Cray based on their experience with high-performance computing. We read some of the presentations on the Chapel home page and saw several things that were intriguing. A scan of the language specification reinforced the feeling, and we decided to dive deeper.

The language's goal is to cleanly express any type of parallelism in a program and map it to machine configurations ranging from multi-core to multi-system. It wants to avoid low-level constructs, such as threads or the MPI or OpenMP architectures, for all users without dictating how to achieve parallelism, as previous languages had done. To quote their literature, "Chapel was designed with concepts that support data parallelism, cooperative task parallelism, and synchronization-based concurrent programming." The language focuses on high-level abstractions so that they are independent of the detailed execution, while still offering the opportunity for low-level control. You can manually partition data into chunks and distribute it over the machines, or you can treat the data as one piece and let the language do the work for you. All this is wrapped in features borrowed from modern programming languages to make a comfortable programming environment.

It sounds good. Of course here we focus mainly on image processing on multi-core CPUs and GPUs, not on big hardware. The issues faced there are outside our experience. The white papers, tutorials, and documentation, however, hinted that there might be a good overlap between this problem domain and some of the basic features of the language – array support, domains, built-in parallel loops. To learn the language, we decided to implement several image processing programs in it and see how they developed. Part of our motivation is the love of learning, part is answering the practical question "is this something we can use?" As we developed these programs, chosen to increase in complexity from per-pixel operations to local neighborhoods to iterative algorithms, we would document the language and how it applies to this problem domain. This is the result.

The outcome isn't certain. We might find there's not a good fit. We aren't trying to sell the language, to convince you Chapel is the next great thing – and why aren't you using it already? Let the language designers and Cray (or whoever has control of the project) do that. We believe that the strong, interesting features of any language will become clear as you read about them. They'll prick your interest, get a nod of approval. If this happens often enough, then you too might decide to investigate further, and then hopefully this document will help you get started. We'll keep our opinions to ourselves until we get to the end.

This isn't a start-at-the-beginning kind of guide. We assume you know how to program and have some experience with C (as the first chapter will show). We will cover the basics of the language, from types to data structures to operators to statements, but our focus is more on the aspects that differ from our expectations than on basic programming techniques, and the code snippets will either focus on the layout of the commands or how their behavior differs. We'll generally build up the program for each chapter in pieces, adding more features and using more of the language as we go along. We design on Linux boxes and the development flow for compiling and running executables is based on that. The tool chain that you need to build the Chapel compiler – specifically gcc and make – is all you'll need for these programs. You will need the PNG library and header file; if not provided with your distribution, look at http://www.libpng.org.

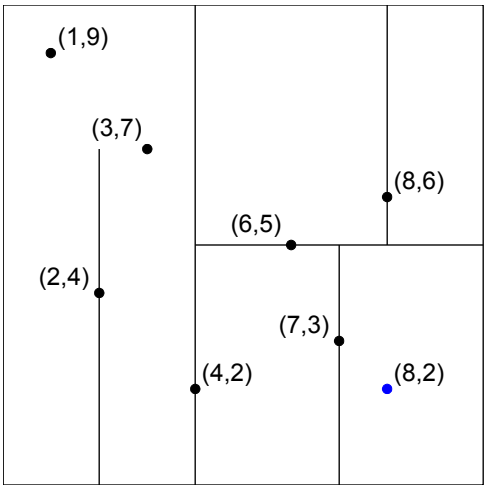
The Chapel home page is at https://chapel-lang.org. You'll find documentation there – tutorials, presentations, and the language spec – as well as a download for the compiler source. One you have that, we can start. The examples and text are up-to-date for the latest (April 2018) release.

The home page for this project is at http://www.primordand.com/chapel_by_ex.html. You can find downloads for all program files and on-line and PDF copies of this text there.

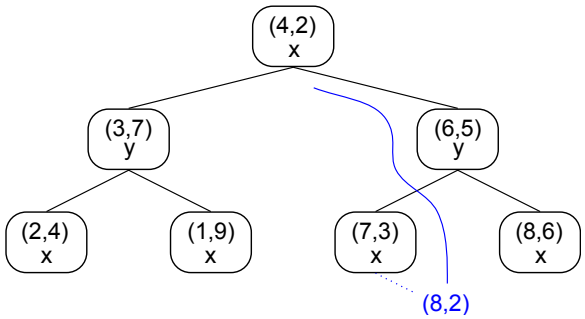
AN EXAMPLE

"Hey, we found a programming language that looks interesting so here's a long description of how to use it" might not be the most compelling motivation for spending time learning about Chapel. Here's an example that shows many of the core ideas behind the language in action.

A kd-tree is an efficient way to search a k-dimensional space. It is essentially a binary tree which cycles through the coordinates level by level, so that the root node splits the space along x, the second level by y, the third again by x (for a 2-dimensional space) or z, the fourth by y (2D) or x (3D) or the fourth coordinate, and so on. The goal of the example is to divide the space with a set of ordered points, building a balanced tree. We'll use a flat array with 1-based indexing to store it, so the left child of the parent `p` is at `2*p` and the right at `2*p + 1`. The algorithm will run in parallel as it recursively builds the tree layer by layer.



Plane division for sample kd-tree



kd-tree with point and base coordinate in node

```
param ndim = 3;
config const treedepth = 5;

var Ltree : domain(rank=1) = { 1..(2**treedepth)-1 };
var tree : [Ltree] ndim * int;
var data : [Ltree] ndim * int;

/* cbase is the coordinate 1..ndim which is the primary sort key for
   this level.  p is the node in the tree to assign in this pass. */
proc assemble_tree(ref points : [] ndim * int, ref tree : [] ndim * int,
                   cbase : int = 1, p : int = 1) {
   const cnext = if (cbase == ndim) then 1 else (cbase + 1);
   const medpos = partition_median(points, cbase);
   tree(p) = points(medpos);
   cobegin {
     assemble_tree(points[..medpos-1], tree, cnext, 2*p);
     assemble_tree(points[medpos+1..], tree, cnext, 2*p+1);
   }
}

assemble_tree(data, tree);
```

Let's look at this piece by piece.

```
param ndim = 3;
config const treedepth = 5;
```

These two lines define two constants which are type-inferred to be integers. The `param` keyword means `ndim` is a compile-time constant. A `const` is a constant set at runtime and `config` means the compiler will automatically create a command-line option `--treedepth` that allows you to change the value each time you run the program. You do not have to parse the command line to get this behavior.

```
var Ltree : domain(rank=1)
```

Arrays are a primary data structure in Chapel. They are backed by domains which define the indices. Domains can be a series of values, as we'll use here, or discrete, like a set of hash keys or sparse indices. The `var` means we can change the value of the domain; any array that uses it will automatically re-size. The colon is followed by a type declaration which says that `Ltree` is a domain. The rank of the domain is the number of dimensions, so here we're saying `Ltree` is a linear set of indices.

```
= { 1..(2**treedepth)-1 };
```

Domains are built on top of ranges of values which define the continuous set of indices. You can think of ranges as the bounds of a `for` loop, and they can have an increment between values as well as a direction. The .. notation indicates this is a range from 1 to the number of nodes in the tree (inclusive), which is one less than the power of two of its depth (ie. a tree of depth 3 has 7 nodes, 4 has 15). The ranges are placed inside braces when setting the domain. The assignment is done when `Ltree` is declared so this is an initialization of the variable but all variables have default initial values if not provided.

```
var tree : [Ltree] ndim * int;
var data : [Ltree] ndim * int;
```

This defines two arrays. `[Ltree]` indicates they are arrays over the `Ltree` domain. The elements of the array will have type `ndim * int` which represents an `ndim`-element tuple of integers. With `ndim=3` our points will look like (x, y, z). The `data` array will hold the unsorted points, `tree` the sorted. A kd-tree cannot be modified easily once built, so the points will need to be copied into `data` before calling `assemble_tree()`.

```
proc assemble_tree() { .. }
```

This is a function declaration where

```
ref points : [] ndim * int,
```

is the first argument, an array of `ndim`-dimensional point tuples. We don't have to specify the domain in the declaration and can get it later with `points.domain`. `ref` is an argument intent and says that the array is passed by reference. This means that we will be able to change the contents inside the procedure and the value in the caller will also change. Other intents include `in` for pass-by-value (where the value in the caller does not change) and `out` which copies the value assigned within the procedure to the argument in the caller.

```
ref tree : [] ndim * int,
```

is also an array of `ndim`-dimensional tuples. Because we haven't specified a size between the brackets for either array, they may be different, although Chapel has a way to require that they be the same.

```
cbase : int = 1,
```

defines an integer argument. The default intent for primitive types is `const in` which means the argument's

value cannot be changed within the procedure (but `in` alone would allow this). `cbase` is assigned a default value of 1 in case it is omitted when calling the procedure. You can also specify arguments by name when calling.

```
                                        p : int = 1) {
```

`p` is the root node of the tree and should have the value of 1 on the first call. We also provide a default so we can omit the argument and get the correct behavior. Such arguments must come at the end of the list if you want to mix them with arguments without defaults.

`assemble_tree()` has no return type, which would be placed between the closing parenthesis and opening brace.

```
    const cnext = if (cbase == ndim) then 1 else (cbase + 1);
```

This determines the coordinate used for sorting the next pass. The `if..then..else` construction is Chapel's ternary expression and we use it to cycle the coordinates through the dimensions of the space. The `const` intent means we cannot change the variable within the function after initialization. The type of `cbase`, `int`, comes from the expression.

```
    const medpos = partition_median(points, cbase);
```

Here we call another function, `partition_median()`, to find the median point of the points array. It works like the partition function used in QuickSort: the elements of the points array are re-ordered so that those smaller than the median lie to the left of it, those larger to the right, and the position of the median is returned and set to the constant `medpos`. Note we haven't provided an implementation of this function in this example. A simple but very inefficient approach would be to sort the points array using the `cbase`'th element of the tuple, and then to return the index of the array's middle.

```
  tree(p) = points(medpos);
```

This copies the median point into the tree. Chapel uses parentheses for array indexing.

```
  cobegin { .. }
```

Chapel has four statements for parallel execution. `cobegin` says to run each of the statements in its block in a separate task. There is also `begin` which runs statements asynchronously, `forall` which can split a range into many subranges and run each in a task, and `coforall` which must execute each iteration of the loop in a separate task. You can think of the first two as being oriented to do tasks (statements) in parallel, while the other two process data (indices) simultaneously. With the `cobegin` we will run the two recursions in parallel; we don't need any other code else to get this behavior.

```
    assemble_tree(points[..medpos-1], tree, cnext, 2*p);
    assemble_tree(points[medpos+1..], tree, cnext, 2*p+1);
```

These two recursions operate on the points that are smaller and bigger than the median. The smaller points go in the left child at `2*p`, and we take the first half-array of the points by slicing with the open-ended range `..medpos-1`. This slice takes the intersection of the array's range with one that has no beginning. It ends just before the median. The result is a new range that starts with the first point and goes to `medpos-1`, inclusive. On the other side we need to build the tree at the right child `2*p+1` over the subrange `medpos+1..`, where the new range will start after the median and go to the last point. Because the children don't overlap we don't have to worry about changing the array in independent tasks.

```
assemble_tree(data, tree);
```

The procedure call starts the process. As we said, we can leave out the `cbase` and `p` arguments because they have defaults. Chapel has well-defined rules for resolving a mixture of positional, keyword, and default arguments.

The core of this algorithm – partitioning and recurring on separate subranges of the input – is the same used for QuickSort. You'll find a full implementation of a kd-tree using a different approach in the RANSAC chapter.

A bullet-list of additional language features that aren't in the example includes

- Language support for defining parallel tasks which the runtime will automatically create and run for the hardware available, as well as offering control over how the mapping to hardware will be done.
- Synchronized and atomic variables.
- Generic types and data structures and polymorphic functions, where the specific versions are generated at compile time and not by dynamic routing.
- Object-oriented data structures, with classes and records that combine data and functions and inheritance.
- Modules to further partition programs into separate libraries.

OVERVIEW

The programs we'll be writing were chosen to build in complexity through the tutorial.

1. First we need to read images. We'll use the PNG library, which requires binding to some C code to handle the interface. We'll need to cover types, variable and function declarations, and two structural types, classes and records. We'll see how Chapel organizes programs into modules.

2. Next we'll do color conversion to transform our RGB input into another color space, the CIE L*A*B*, where the L plane is a good greyscale version of the image. This is a pixel-by-pixel operation and will introduce us to arrays, domains, and ranges. We'll cover the standard language – expressions and statements. Then we'll extend this to other color spaces and introduce Chapel's support for first-class functions.

3. For the third example we'll write a Gabor filter, which is a large convolution run over an image to detect oriented edges. We'll learn about subdomains and subranges to iterate over the the convolution kernel, and the syntactic support Chapel offers for defining them. We'll see reductions that combine multiple results into one.

4. The fourth example introduces us to Chapel's support for parallelism, either at a task (statement) level or by dividing a data set and working independently on each part. For the first we'll make a bank of Gabor filters to evaluate at the same time, and for the second we'll split the RGB space to verify the bounds of the LAB color correction. The language features we'll study are the statements that generate parallel tasks, and synchronization variables.

5. Our next example is k-means clustering which we'll use to quantize (reduce) the number of colors in an image. This uses an iterative refining of how pixels are assigned to clusters, which can, and will, be done in parallel. We'll talk about Chapel's support for atomic variables.

6. Leading up to our final program we'll need to implement a FAST corner detector to mark features in images we want to compare. The detector looks around a small ring to see if the center is the tip of a light or dark corner. We'll write a custom loop iterator to step around the ring, and we'll study Chapel's generic data structures and procedures in order to store a list of the corners found.

7. Finally, we'll use the corner detector to align two images, compensating for rotation, scaling, and translation. This technique uses a random sample of points to determine the alignment, running many trials in parallel to find the best correspondence. There won't be any new language features. We ourselves haven't done this before and the goal of the exercise is to see what developing new image processing code is like in Chapel.

INSTALL

Once you have downloaded the distribution from http://chapel.cray.com and untarred it, you'll find a README with instructions for compiling.

There is a simple version that does not use many third-party libraries (which are provided with the distribution). To compile it you need to source a script in your shell and then run make. We run bash, so the commands run from the top-level directory CHPL_HOME are

```
source util/quickstart/setchplenv.bash
make
make check
```

The last step runs built-in tests to make sure everything is correct.

The full version uses a different script.

```
source util/setchplenv.bash
make
make check
```

This version of the script sets several environment variables to add in extra features. (Well, technically it does not set several variables, and their default values cause the extra features to be compiled in.) You'll find a full list in CHPL_HOME/doc/rst/usingchapel/chplenv.rst (HTML versions also exist). The most important are:

variable	purpose	quickstart	full
CHPL_TASKS	threading library to use	fifo (pthread)	qthreads
CHPL_HWLOC	package to detect hardware setup	none	hwloc
CHPL_COMM	inter-processor communication package	none - local CPU only	
CHPL_GMP	GNU Multi-Precision library	no	integrated
CHPL_REGEXP	regular expression library	no	integrated
CHPL_LLVM	use LLVM compiler, can embed C directly	no	integrated

You can set these manually before compiling if you want to force the behavior. Be aware that LLVM support takes a long time to compile and you might want to pass -j # to make to do it in parallel.

Before using Chapel you will need to source the setchplenv script from CHPL_HOME once after login. You can then run the compiler anywhere. We set the following variables instead in our .bashrc

```
export CHPL_HOME=<where you unpacked the distribution>
export CHPL_HOST_PLATFORM=`"$CHPL_HOME"/util/chplenv/chpl_platformpy`
export PATH="$CHPL_HOME"/bin/"$CHPL_HOST_PLATFORM":"$CHPL_HOME"/util:$PATH
export MANPATH="$CHPL_HOME"/man:$MANPATH
```

You will also need to set the other CHPL_* variables you forced before building the compiler.

Inside the distribution you'll find these important directories and files:

doc/rst/language/chapelLanguageSpec.pdf - complete description of the language
doc/rst/technotes - descriptions of features being added to language (Work In Progress)

doc/html/index.html - browser version of the documentation, including modules
examples/primers - code examples for language features
examples/programs - complete, simple programs
examples/spec - code snippets from the spec
examples/benchmarks - various performance benchmarks
modules/standard - standard libraries , including IO, math functions, timers, and error handling
modules/internal - parts of the language that are written in Chapel
modules/packages - other libraries
third-party - support programs from outside the Chapel team used to build the compiler and runtime

There's a style definition for emacs and vim in highlight/[emacs|vim].

Compiling and Running a Chapel Program

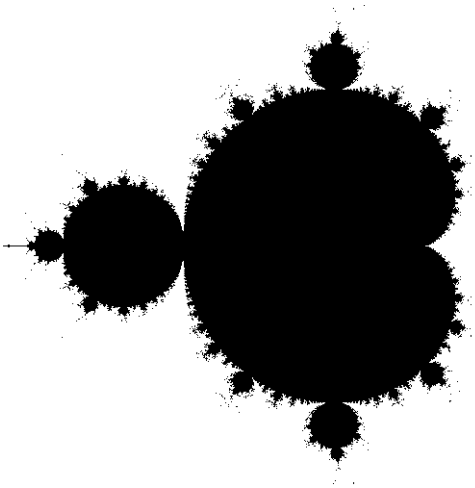
The distribution contains many example code snippets and programs. Let's pick one to see the basic compile and run cycle. Copy mandelbrot.chpl from CHPL_HOME/examples/benchmarks/shootout to a project directory. Then do

```
chpl -o mandelbrot mandelbrot.chpl
```

to compile. To generate an n x n image type

```
./mandelbrot --n=16000 > img.out
```

This creates a 16000x16000 image in PBM format with the result. The `--n` is a run-time constant that Chapel allows to be set on the command line.



Mandelbrot set

Typing `-h` or `--help` the program name prints a table with all the command-line options that are available.

Typical for a compiler, there are pages of options for the chpl command line. See the man page. We will need only a few, though, which we'll cover as we go along.