LOOSE ENDS

There are a few parts of Chapel that we haven't touched on. Here's a quick summary until we're able to provide a more detailed description:

- Locales are separate processing units with local memory. The runtime identifies the hardware configuration and provides it through an global array `Locales` that describes each unit, such as the number of cores it has, the size of its memory, how many tasks it can run, and a unique identifier. The `on` statement executes a code block on a specific locale.

- Domain maps control the binding of index ranges to locales. They can be created as standalone variables with the `dmap` type and bound to a domain by placing `dmapped <map>` after the domain declaration.

  ```
  var dommap : dmap(Block(rank=2)) = new dmap(new Block((1..4,1..4));
  var dom : domain(rank=2) dmapped dommap;
  ```

  creates a 4x4 grid over which the domain indices will be divided in block fashion. Alternatively you can specify the map directly with the domain

  ```
  var dom : domain(rank=2) dmapped Block((1..4, 1..4));
  ```

- Chapel provides four built-in domain maps called distributions; they are found in CHPL_HOME/modules/dists.

  1. Block - each range is broken into several pieces according to the layout of the locales and all combinations of the pieces are run on different locales

  2. Cyclic - range indices are taken modulo the number of locales along that range (locales may be arranged over multi-dimensional domains)

  3. BlockCyclic - each range is broken into a block with a specified size and the blocks are assigned modulo the number of locales along that range

  4. Replicated - each locale receives a copy of the domain and works independently, without communicating changes to the data to other locales

  Chapel also has a well-defined interface called the DSI for creating new domain maps. It is explained in CHPL_HOME/doc/rst/technotes/dsi.rst.

- Associative domains and arrays use discrete values of any primitive type or class as indices, serving as a hash table or dictionary. The set of values can be changed with the `add()` or `remove()` methods, or using the `+=` and `-=` operators; values are also added if used as the index to an array during assignment. Declare an associative domain by specifying the type. An initial set of values can be given between braces.

  ```
  var counting : domain(string) = { "one", "two", "three", "four" };
  var testarr : [counting] int;
  testarr("four") = 4;
  testarr("five") = 5;     /* or counting.add("five");  counting += "five" */
  ```

- Sparse subdomains are a list of values belonging to another domain. Indices are added like an

associative domain. An array backed by a sparse subdomain has a default value `arrayname.IRV` for any index not provided in the list. You can only set values for indices that have been specified; they will not be automatically added to the sparse index list. Declare it by putting sparse before the subdomain type.

```
dom : domain(rank(2)) = { 1..ncol, 1..nrow }
var sparsedom : sparse subdomain(dom);
var sparseimg : [sparsedom] uint(8);
sparsedom += (1, 1);
sparseimg(1, 1) = 5;
sparseimg.IRV = 128;     /* all pixels but (1,1) have this value now */
```

- Set operations such as union, intersection, and membership tests are being added atop associative domains.

- Classes and records support an object hierarchy, where you can declare a superclass with the subclass:

  ```
  class subclass : superclass { .. }
  ```

  Subclasses inherit members and methods from their parents.

- IO is done to a file through a channel. These channels implement a generic `Reader` or `Writer` interface. The methods `readThis()`, `writeThis()`, and `readWriteThis()` can be overloaded to customize IO operations for a class or record. The IO standard module defines the operations supported.

- You can embed C code directly in a Chapel source file with an `extern` block.

  ```
  extern {
    rgbimage *rgb = NULL;
    if (alloc_rgbimage(&rgb, 10, 10) < 0) {
      /* handle error */
    } else {
      /* work on image in C */
    }
    free_rgbimage(&rgb);
  }
  ```

  This requires enabling LLVM support in the compiler, as it is used to parse the C.

- CHPL_HOME/modules/standard and CHPL_HOME/modules/packages have many more libraries, including file system support (FileSystem, Path), a libcurl interface for file transfers (Curl), system calls (Sys, Time, Spawn), math and number functions (Math, BitOps, FFTW (Fast Fourier Transform), LAPACK, BLAS, GMP (GNU Multiprecision Library), Norm, UtilMath, LinearAlgebra), an interface to Hadoop systems (HDFS, HDFSIterator), tracking of network communication (CommDiagnostics, startInitCommDiags), regular expressions (Regexp), cryptography (Crypto, Random), and iterators that can steal work from others if they have nothing to do (AdvancedIters).

- CHPL_HOME/doc/rst/technotes describe language features that are still in progress, including partial compilation of source files into libraries, a REPL, more locale architectures including NUMA, LLVM support, and writing formatted output to strings.

As we have said, the doc, doc/rst/technotes, modules/standards, modules/packages, and examples/primers directories in CHPL_HOME contain a diverse collection of design and usage notes.

SUM-UP

This brings us to the end of our study of Chapel. It's time to look back and answer the question posed at the start: Is this a language we can use?

We've seen Chapel in action in pixel-level computations (color conversion and FAST corner detector), at the matrix level (Gabor filter convolution), and at a more general level (k-means clustering and RANSAC feature matching). We've used most, but not all, of the language's features, which we can summarize in a list:

Data Structures

- arrays (data), domains (indices), ranges (iteration)
- classes and records with embedded variables and procedures
- standard iterator interface any class or record can provide (`these()` method)
- tuples and enumerations

Parallel Programming

- data-level parallelism breaking up iterations (`forall`, `coforall`)
- task-level parallelism based on statements (`begin`, `cobegin`)
- synchronization across tasks (`atomic`, `sync`, and `single` variables)

Generic Programming

- classes and records programmable by type, fixed-value parameters
- function overloading and polymorphism, `where` clauses to specify type requirements
- queried arguments for types and details
- variadic argument lists

Language Fundamentals

- argument intents that control writeability and linkage to the caller
- standard control statements with keyword version for single statement and braced version for blocks
- primitive types `int`, `uint`, `real`, `imag`, `complex`, `bool`, where the bit size can be specified if needed
- programs grouped by modules, with an inherent module per file and access control (public/private)
- extended set of operators, including exponentiation and variable swap
- program constants that can be changed at runtime on the command line, or parameters set at compile time

Extensibility

- custom iterators
- custom domain maps

But what has it been like programming with the language?

Experience

Writing programs in Chapel has been pleasant. The language fits this problem domain well. Domains and slices are natural ways to work with images and clearly signal the intent behind the code. Having domain changes re-allocate arrays is a strong encouragement to properly set them up. Parallelism is easy to add to programs, but not entirely worry-free. You still need to keep in mind what data is crossing task boundaries and if there might be races (the default intents prevent this but can be circumvented), and it's not clear how a piece of code will act on actual hardware – the runtime is a bit mysterious. Generic classes and queried arguments provide the flexibility for re-usable code. Argument intents make the role of each argument clear, and supporting multiple output arguments is nice. Interfacing to C is straightforward although a little verbose when needing to use the C types. The automatic creation and parsing of command-line constants means never having to program a parser again. Chapel is an expressive language and that is apparent while coding.

We have also found developing in it to be somewhat frustrating. The flexibility of the language and its default behavior is partly to blame, as is the compiler. Part of Chapel's expressiveness depends on its defaults, automatically generated constructors, accessors, and iterators, and overloaded operators that uniformly handle different data types. They reduce clutter in the code and are mostly sensible and intuitive, but we found ourselves constantly butting up against them while compiling and debugging. The overall impression is that the language is a bit squirrelly as we were constantly reminded of how much was happening behind the scenes, and thus how unsure we were of exactly what was going on. The flexibility can be confusing. For example, if out of habit you type array references with brackets instead of parentheses the program will compile, but the behavior might not be what you expect (at least for us, “strange results” cleared up after switching the brackets). Or, if you make a typo in the name of a constructor you will end up silently using the default, which is not what you want; this happened while developing the circumference class. The behavior of a program can also be confusing. We're left with the impression that passing arrays around in structures, particularly records, is not reliable. The data seems to corrupt eventually, and in the worst case the program fails. This happened a few times developing the RANSAC program, and its final shape, with only primitives stored in the mapinfo and tryinfo records and the map1to2 and map2to1 arrays being re-generated with the best try, rather than caching them as the program proceeded, is a result. The compiler did not help. Error messages are sometimes indirect; if you forget to tell a `forall` loop that a variable outside the loop is being used, the warning message complains about a bad lvalue. The compilation stops on the first error, and since the current version is a bit slow the 'fix typo - compile - fix typo - compile' cycle is tedious. Chapel also only seems to look at code that is being used. If you test compile a stand-alone library it can come out clean, with errors only appearing when you start calling the library functions from the main program. The compiler can generate error messages rivaling Clojure stack traces if it can't resolve an overloaded function, “helpfully” listing a hundred valid type signatures.

Perhaps this is part of the learning process and we find ourselves in something like an uncanny valley of confidence where we're starting to assume we know the language while pushing on it and exposing the gaps in our understanding. We don't yet have a clear idea of what is happening, and there's little feedback about missed opportunities for optimization or warnings about inefficient code (say, flagging array accesses that may be out of bounds when the compiler can't prove they are within the array's domain). It's a bit like learning the CUDA model for memory and execution without the profiling tools to show how you can improve your performance: there's a lot of stumbling about and trial-and-error, and any conclusions drawn about best coding practices may

be due as much to luck and circumstance as to valid reasons.

Performance

You might have noticed that for a High-Performance Computing language we haven't talked much about how fast Chapel programs run, other than to compare different implementations against each other. There's a few reasons for this. Most importantly the team has been focused on the language definition and only recently started working on optimizing the code the compiler produces. Their presentations from the end of 2014 acknowledge that performance is poor but is getting better. We shouldn't be surprised if our runs are slow. Another reason is that we can't compare the programs we've developed here directly with our internal versions, either because we've changed the approach or more often the data type. As an example we tend to work with integer images, and scale the greyscale plane and the Gabor kernel filter before running the convolution; our implementation here runs in floating point. There are also things we can't explain, such as where to best use `forall` (top level? everywhere? spot placement?). Finally, there is little feedback about how to optimize programs. It's a bit of a black box. Our impression, though, from these programs and others in the language shootout benchmarks is that Chapel is slow compared to C, and some constructs, namely reductions, are borderline unusable.

Maybe a story is in order. The k-means clustering was our first thought at writing a parallel program in Chapel because we had already gone through the exercise in C. Our serial version had run too slowly, so the first change we made was to split the image into equal parts, one per thread, converting the next-pass clusters into per-thread sub-totals and combining them at the end. In other words, it was the approach taken for kmeans_v1. The code edits took an hour and gave us about a 3.5X speed-up with four cores/threads. We then ported the algorithm over to CUDA, where the biggest changes needed were to fit into the GPU's memory model. The running time improved by a factor of 6. The Chapel kmeans programs, in comparison, were 15 times slower than the serial version, or 40% slower if compiled with `--fast`. Again, the differences between the C and Chapel versions affect the behavior of the clustering and the results are not directly comparable, but we have tried to have the same amount of work (same image so pixel count is the same, limiting the number of passes to be the same) so the rough result stands: the Chapel program is slower.

[For a better comparison, we've re-worked kmeans_v2.chpl to use only integers. This version converts each color plane to 8-bit, ie. `clrimage` -> `rgbimage`, and changes the types in the cluster record and procedure arguments from `real` to `int`. There are still differences in the algorithm that cause the Chapel version to take two or even three times as many iterations to converge, but the per-iteration time of the integer version is only 20-40% slower than the threaded C version. The per-iteration time of the float version is three or four times slower than the integer, which gives us the 40% slowdown to the serial C version.]

As Chapel evolves it is generating faster code. The Gabor kernels are running 2-4 X faster in 1.17 (April 2018) compared to 1.11 (April 2015); this is with --fast compilation. The kmeans programs are 20-30% faster, and both RANSAC versions 5 X. Without --fast the timing number are more stable. The Gabor kernel timing ranges from 20% slower to 40% faster, the kmeans even up to 20% faster, and RANSAC has a 2 X improvement.

Work In Progress

Chapel is still in active development. There are several language features that have not been implemented or will change. Some, such as strings and data hiding in classes/records, seem fundamental and pose the biggest risk for code re-work in the future. Others like the order of atomic operations over a distributed network hint at design issues that are outside our experience. (One reason for not trying locales and domain maps is that we haven't used that kind of hardware.) Many features are partially implemented and not yet well documented, or have bits and pieces of text describing them scattered about between the modules, docs, and examples directories. With each new release new find a few programs stop compiling. Fixes are small and make sense, but emphasize that the language and libraries are not yet stable. And of course there are bugs.

Any of these mean that there might be conclusions in these examples that are not, or should not, be valid, another example of trial-and-error learning. Two that come to mind are the decision not to use enumerations as constants or the recommendation not to use arrays-of-arrays as they cannot be used as arguments. Both apparently should work. This is not a reason we should reject using Chapel, only something to be kept in mind. The language will continue to evolve, and we must be prepared to move with it.

One piece we would like to see is CUDA support in the runtime, for both practical reasons and curiosity. Practical because it's the parallel environment most accessible to us and which has shown significant benefits, but we also wonder about how well the language maps to the GPU and handles the constraints of the hardware, especially the memory model.

Overall Conclusion

Early on, about a quarter of the way through this project, we made a placeholder note in the outline here that an example of a conclusion might be "Performance not there yet, will follow. Good for prototyping." That was a prescient remark, for it is our conclusion. Chapel is a comfortable language for programming for us and we expect it will get better with each half-yearly release. Its lineage is clear, with roots in the world of C and Unix, and this is the environment we use. Should we need to port a prototype back into that world, say the kd-tree class or the RANSAC algorithm, it seems straightforward enough to do. But first we need to determine if RANSAC can be tweaked and its accuracy with scaled images improved. Onward to the next project ...

Feedback

And that brings us to the end of the text. If you have comments or feedback, they would be very much appreciated. We can best be reached by e-mail at chapel_by_ex@primordand.com. Thank you for your time and attention.