From C and Python to Chapel as my main Programming Language

Nelson Luís Dias
nelsonluisdias@gmail.com
Department of Environmental Engineering, UFPR – Universidade Federal do Paraná
Curitiba, PR, Brazil

ABSTRACT

I describe a trajectory of using a number of programming languages in research, from my early days with Fortran, Pascal and Modula-2, to mainly C and Python during approximately 30 years, and then to Chapel in the last 2 years. During all of this time the desktop PC has been adequate to my processing needs; therefore, this is essentially a "Chapel at the desktop" experience report. Chapel is a very elegant language, providing the power and speed of C and Fortran, while allowing a high degree of abstraction and expressiveness that rivals Python's. I have used it in the last two years for: calculating statistics over massive turbulence datasets, implementing models for lake evaporation in hydrology, and testing some relatively simple numerical solutions of partial differential equations. Its easy portability from C, Fortran and Python allowed fast translation and re-use of my existing libraries. On the few (but not impossible to live with) shortcomings, passing functions as procedure parameters is somewhat unwieldy; re-indexing arrays in procedures is verbose; compilation could be faster; and executables are very large.

KEYWORDS

user experience, Chapel vs C and Python, desktop computing

ACM Reference Format:
Nelson Luís Dias. 2022. From C and Python to Chapel as my main Programming Language. In *Proceedings of The 9th Annual Chapel Implementers and Users Workshop (CHIUW 2022).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 SPEAKING MANY (PROGRAMMING) LANGUAGES

I may have been fortunate to learn to program in some dialect of Fortran 66 using punched cards on a PDP-11, because it gave me a perspective of how great the evolution has been, and how much better things are today. After learning Fortran, I have done actual research (using editors and the command line) in many languages: VAX Fortran, Modula-2, Turbo Pascal, and for almost 30 years during and after my PhD, C and Python, with an occasional relapse to Fortran 9X. I discovered Chapel because I have been interested in doing my own Fluid Mechanics simulations that will require parallel processing, but what I have found is that it is able to be much more than a niche language exclusively for HPC. Instead, it is very well suited to be a general-purpose language for research, and a very good substitute for all of Fortran, C and Python. During this talk, I will try to explain why, with examples.

CHIUW 2022, June 10 2022, Online Conference

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2 THE ROLES OF LANGUAGES IN MY RESEARCH

Programming languages seem to attract judgements and win adherents for what may well be subjective reasons: a persons's sense of aesthetics, a preference for minimalistic (C) or near-baroque (C++?) ways to write algorithms, a disposition of writing all code from the ground up or using available libraries (Python's strength), all play a role in their language of choice.

It is not my intention to add fuel to the debate. As much as I have my preferences and inclinations with programming languages — which will become clear as I progress— I also need to get something done with them. My research mostly involves calculating statistics over large datasets of turbulence data measured in the atmosphere, but also working with smaller datasets of mean hourly or daily atmospheric data, as well as occasionally solving some ordinary or partial differential equation numerically. This dictates or at least narrows down the choices to a language that:

(1) is easy to learn and use;
(2) has enough constructs to work efficiently with large chunks of data: array slicing is very desirable;
(3) runs fast.

Alas, until a couple of years ago, I did not know of any language that simultaneously fulfilled all of the criteria above. Let us concentrate on my preferred contenders: Fortran, C, and Python. Some of my own limitations narrow the list: I do not know or use C++ (nor anyone with whom I have ever collaborated), so I will not comment on it, other than mention that it obviously has many more features than C, and that it does have a `std::slice` function. Neither do I know or use R or its Python "conterparts" like Pandas; I do not use Matlab either.

Python, the youngest of the three, is easy to learn and use. It also comes, or is easy to extend, with libraries that greatly simplifty programming ("batteries included"), such as Numpy and SciPy. In particular, Numpy has very power arrays that can be sliced. Python therefore fullfils (1) and (2) above. However, being interpreted, it can be very slow. There are all sorts of add-ons like Cython and Numba that can improve performance, but then you are no longer in the context of the language itself, and start to lose the simplicity of requisite (1).

C, the second youngest, is a small and simple language, although some concepts like mixing pointers and arrays can be a little hard.

Let us say that it fulfills (1) at least in part. C arrays however cannot compete with Numpy nor with the arrays provided by modern Fortran, and cannot be sliced — although C99 improved the capability of passing multi-dimensional arrays to functions. So C fails (2). But it runs very fast, and (3) is OK.

Fortran, the oldest, has continued to evolve since it was born in the 1950s. This produced a huge language, that still needs to compile legacy code, as more and more features were introduced along the way. For its size and redundant features, I will fail Fortran in (1). But it has very good arrays that can be sliced and passed easily as arguments: (2) is OK, and it runs at least as fast as C, so (3) is also OK with Fortran.

At this point, in order to get a reasonable mix of (1)–(3), one is forced to "live with" at least two, and perhaps all three, of the above languages. For example, I have worked with Python in projects that did not require heavy processing (or for which pre-compiled code in Numpy, Scipy, etc., was good enough), and with Fortran or C when heavy numerical processing coded by myself or my students was involved.

3 CHAPEL'S STRENGTHS

With Chapel the situation is qualitatively very different. It is *not* a small language — for example, like Python, it allows for object-oriented programming and some ideas of functional programming, but one can go quite far with a reasonable subset of strictly procedural programming that readily gives all of (1)–(3). This means that Chapel's claim that it is as fun to program as Python and as efficient as C or Fortran is, in my experience, fully realized. Indeed, this talk's main point is little more than corroborating that claim.

Chapel has the potential to fill the niche of a convenient and easy-to-use interpreted language such as Python while at the same time delivering the performance of C and Fortran — even in applications that do not run in supercomputers. Then, the researcher/engineer/etc. can concentrate on one syntax, one compiler, and use the free space in his/her mind to problem solving in his/her area.

In Chapel, it is as easy to re-use code as it is in Python. "Separate compilation", which is mildly complicated in C and Fortran, is transparent, as long as the compiler knows where to look for the modules that the user is importing. Here, declaring a path to the directory where my libraries (in source code) reside is sufficient, as in

```
export CHPL_MODULE_PATH=/home/nldias/Dropbox/\
nldchapel/modules
```

Apparently, however, this comes at the price of re-compiling all source code each time that a program is compiled and linked into an executable. So far, I have translated about a dozen of my libraries that were born as C or Python code to Chapel modules, and have been using them without any problem. In a few of my libraries, for matrix multiplication and for spectral analysis, for example, I made quick changes to the code from `for`s in C or Python to `forall`s in Chapel, to speed execution up. Then, with relatively small changes in the code, some substantial speed improvements can be gained, that often approach (obviously) multiplying execution speed by the number of cores available in the CPU. Here is the list of my current Chapel modules, without any pretense of them being "professional" or "high-quality" software:

`angles.chpl` Very basic operations with angles (degrees to radians, etc.)
`atmgas.chpl` Concentrations, densities, and other properties of atmospheric gases.
`dgrow.chpl` Grow a domain dynamically to accomodate an out-of-range index.
`evap.chpl` Evaporation formulas and methods in hydrology.
`matrix.chpl` Vector and matrix multiplication, tridiagonal matrix algorithm (with a sparse domain), Gauss-Jordan inversion of a matrix, solution of a system of linear equations with Gauss elimination.
`nspectrum.chpl` Spectral analysis.
`nstat.chpl` Basic statistics, linear regression, Lowess [1] low-pass filtering.
`nstrings.chpl` Simple string operations.
`ssr.chpl` Search, sort and replace procedures for arrays.
`sunearth.chpl` Astronomical formulas for the trajectory of the Earth around the Sun.
`turbstat.chpl` Processing of turbulence data.
`water.chpl` Thermodynamic properties of water.

Specifically, Chapel of course is much faster than Python. For my most demanding applications (for instance calculating turbulence statistics over massive amounts of data, measured 20 times per second during many months) Python is too slow and what had been gained in simplicity in programming was being lost staring at the terminal while Python struggled with its slow loops which were sometimes impossible to avoid using Numpy's constructs. The number of lines is not too different in (my) C, Fortran and Chapel implementations, but this is not a problem. Actually, some of the terse statements possible in Numpy are much harder to understand than `for` (or `do`!) loops, and sharing "clever" Numpy code with my colleagues has sometimes not been very popular. I have not compared the speed of Chapel with that of C or Fortran, but assumed that they produce equally efficient code. In summary, the advantage here is that Chapel is easier to maintain and definitely easier to program than either C or Fortran, but still as fast as on a single core. When it is possible to make simple parallelizations, it is of course much faster than single-processor C or Fortran code.

Like modern Fortran, and unlike C or Python or Julia, Chapel is agnostic about the first index of an array. It does not force all your arrays to begin at 0 or at 1, and you are free to choose. This is a better feature for a programming language because, as I see it, the appropriate starting index may change with the algorithm, problem, etc., and that, in spite of unending heated arguments, there is no overall best choice between 0 and 1 (or something else), not to mention that at the end of the day it is also a matter of personal taste. It does bring complications to arrays as formal arguments: by default, in Chapel an array enters a procedure with its predefined starting indices, and "aligning" the indices (as in matrix multiplication, for example) has to be done manually by calling a `reindex` method or by writing an "index-neutral" algorithm that will be longer and require the programmer to think a little harder. In Fortran, you can force the formal array variable to start at a specified index inside a subroutine, something that in Chapel would be written (if available) somewhat like

```
proc vmax(ref a: [1..?n] real) { ... }
```

which would immediately re-index `a` to start at 1, while making `n` locally equal to `a.size`.

I have also used, sparingly, the ability to write procedures with formal arguments of generic types. This is obviously useful when exactly (or almost exactly) the same code needs to be used to calculate (say) the median of an integer and of a real array. This is quite automatic in Python, but as far as I know impossible in C — and used to be impossible in Fortran as well. In Chapel, you say

```
proc median(ref ax: [] ?at): at { ... }
```

and you are in business.

Domains are a very distinctive feature of Chapel, that I had never seen in any other language before. In Chapel, an array's index set is defined over a domain, and domains are very flexible: they can be rectangular (as in C and Fortran), but they can also be *sparse* (as when dealing with a tridiagonal matrix) and finally they can be *associative* (as in a Python dictionary). Associative domains can also function as sets, over which you can perform the usual unions, intersections, symmetric differences, etc.. Domains can also be used to automatically resize arrays: changing a domain variable changes all arrays defined over it. I have used all kinds above in my research, and the result is code that is clearer and usually shorter than the equivalent C or Fortran counterpart and, importantly, fast to write.

The upshot is a fair increase in my productivity as a programmer (which I am reporting here on subjective grounds) because (a) I have the ability to re-use my own code and maintain my libraries with ease and (b) there are enough syntactical elements in Chapel to implement some ideas easily and fast. In one example that I used recently, for data organized sequentially in time in a one-dimensional array `A`, the indices for the dates can be stored in an auxiliary associative array `idate`, so that `A[idate["2010-02-28"]]` retrives a particular day without fussing too much about years, months and days.

Parallelization is, of course, a central feature of Chapel, and is incredibly easy to implement. A lot can be achieved by changing from `for` to `forall`, being careful about race conditions (but the compiler is often clever enough to forbid suspicious accesses by default!). As an example, table 1 shows the speedup that can be achieved with that change for the solution of Laplace's equation with successive over-relaxation (SOR) in two dimensions on a computer with 12 logical cores. The speedup factor, although not the theoretical maximum of 12, is still a respectable 6–8 (and bear in mind that there are only 6 *physical* cores). It is possible to run Chapel in multiple locales, but only if you have access to a cluster, which I currently do not. I have made a few experiments simulating a multi-locale machine on my desktop computer, but cannot report much on using locales and (domain) distributions.

The content of this section is similar to my words of praise about Chapel to my colleagues and students. I haven't convinced most of my peers but understandably have had more effect on the students. Those that were already doing their job with Python were left alone: doing research is hard enough without having to learn a new language along the way. But newcomers are gently coerced to use Python, and there is an ongoing project with a colleague (the exception) where a PhD student of hers is developing a turbulence model in Chapel.

As a last praise, Chapel is *elegant*. It achieves a good balance between being easy to write (for example, you need to declare a variable, but the type can be inferred implicitly by a simultaneous assignment as in '`var a = 1;`' that declares `a` to be a integer) and safe for the programmer (again in my experience, I make far fewer errors than in C or in Python), and almost always the syntax is a delight to use when writing an algorithm.

4 WHERE ARE THE BATTERIES?

Application libraries are not so readily available in Chapel as they are in Python, and this might prevent potential users to jump on Chapel's boat. Still, there are already very useful "Package Modules" in Chapel's distribution (see https://chapel-lang.org/docs/modules/packages.html). Sometimes they require adding additional libraries to the operating system. For example, I often use FFTW, which requires `libfftw3-bin` and friends to be installed in the debian-based Linux OS (Linux Mint 20.3) that I use. Then, compiling with the additional `-lfftw3` flag does the job.

Alternatively, there are many repositories of useful Fortran and C scientific libraries. If you have the header file and the source file of a library, say `foo.h` and `foo.c`, then Chapel allows you to specify external functions in your `prog.chpl` Chapel source file and to link it with the C library via

```
chpl foo.h foo.c prog.chpl
```

Details can be found in https://chapel-lang.org/docs/technotes/extern.html. Since most Python scientific libraries are wrappers to C and Fortran code that is often open-source and easy to find, with some additional effort it is possible to take advantage of existing scientific routines. I consider that shedding light on this feature and giving detailed examples (for instance, how to install and use CMinpack (http://devernay.free.fr/hacks/cminpack/) can attract a larger audience to using Chapel on the desktop.

5 A FEW THINGS THAT COULD BE BETTER

This is just my opinion! No language can ever be "perfect", as this is a matter of personal taste. A killer feature in one's opinion may well be a disaster in someone else's. Here are a few things that I think could be slightly better.

Compilation could be faster, and the generated executables could be smaller. Faster compilation is definitely the most important thing, if only because people will compare Chapel's compilation times to the instantaneous result in Python or in the GCC family of compilers. I know that this is a goal of the project, so it is probably on its way in forthcoming versions of the compiler.

As I mentioned above, there could be an option for arrays as formal arguments to be reindexed automatically in their declaration.

Procedures as arguments to other procedures are not naturally integrated into the language. "First class" functions are available, but do not work in all cases of practical interest. There are two solutions that always work: wrapping the function *type* in a record and declaring the desired function with `this`, followed by declaring function *variables*, or declaring the function variable in a lambda expression. The corresponding formal argument has to be declared as a generic argument, without any type: the long existing solution in C and Fortran of using a function declaration in the argument list is not possible.

Table 1: Grid size N_n, number of iterations to convergence n_c, estimated $\overline{u}$, MAD and runtime t_r for the serial and parallel versions of the solution of Laplace's equation with SOR.

	serial				parallel			
N_n	n_c	$\overline{u}$	MAD	t_r (s)	n_c	$\overline{u}$	MAD	t_r (s)
128	431	0.7500	2.5625×10^{-7}	0.0647	428	0.7500	2.5278×10^{-7}	0.0107
256	1934	0.7499	1.5653×10^{-6}	0.6914	1931	0.7499	1.5609×10^{-6}	0.1152
512	6947	0.7499	6.6456×10^{-6}	9.9662	6943	0.7499	6.6525×10^{-6}	1.3221
1024	23955	0.7499	2.7032×10^{-5}	137.6410	23952	0.7499	2.7028×10^{-5}	16.7878
2048	80310	0.7498	1.0864×10^{-4}	1867.7000	80306	0.7498	1.0865×10^{-4}	232.3660

This ends, however, my very short list of things that could be better in the language. None of them is an impediment to using Chapel productively. They may require a few extra lines of code, but his has a very little impact in the overall scheme of things. Programming in Chapel is very safe and making errors is far less frequent than in C or Python. Productivity is high, the power of the language's constructs is considerable, and parallelization is almost effortless. I hope that Chapel continues to thrive and that it grows in acceptance. It is now my language of choice.

REFERENCES

[1] W. S. Cleveland. 1981. LOWESS: A program for smoothing scatterplots by robust locally weighted regression. *American Statistician* 35, 1 (1981), 54.