Basic Comparison of Python, Julia, R, Matlab and IDL

February 20, 2018: An updated version of this analysis can be found HERE.

History:

- **January 9, 2017 (Original ducument)**
- **February 28, 2017 (Added Problem 5 and Problem 6)**
- **June 22, 2017 (At the recommendation of Simon Byrne, edited the Julia script for Problem 6)**
- **July 26, 2017 (Simon Danish proposed different optimization options for solving Problem 3 with Julia)**

Comparing programming languages such as Python, Julia, R, etc. is not an easy task. Many researchers and practinioners have attempted to determine how fast a particular language performs against others when solving a specific problem (or a set of problems). Raschka presents Matlab, Numpy, R and Julia while they performed matrix calculations (Raschka, 2014). Hirsch does a benchmarking analysis of Matlab, Numpy, Numba CUDA, Julia and IDL (Hirsch, 2016). From his experiments, he states which language has the best speed in doing matrix multiplication and iteration. Rogozhnikov uses the calculation of the log-likelihood of normal distribution to compare Numpy, Cython, Parakeet, Fortran, C++, etc. He draws conclusions on which ones of them are faster to solve the problem (Rogozhnikov, 2015). Puget determines how several languages scire in carrying out the LU factorization (Puget, 2016).

All these analyses are important to assess how fast a language performs. However, focusing only on the speed may not give us a good picture on the capability of each language. It turns out if we compare how fast languages execute a given computation over the years, we might reach different conclusions as some of them evolve over time (to be more efficiency in solving a set of problems). To determine the usefulness of a language, we want to take into consideration its accessibility (open source or commercial), its readability, its support base, how it can interface with other languages, its strengths/weaknesses, the availabilty of a vast collection of libraries.

As we deal with legacy scientific applications (written in Fortran or C for instance), our primary intent is not to find a new language that can be used to rewrite existing codes. We rather want to identify and leverage "new" languages to facilitate and speed up pre/post-processing, initialization and visualization procedures. As far as possible, we may want to interface our legacy codes to "new" languages. We also intend to use new language to prototype some applications before they are written in languages like Fortran and C.

In this work, we are intested in how each package handles loops and vectorization, reads a large collection of netCDF files and does multiprocessing. The goal is not to highlight which software is faster than the other but to provide basic information on the strengths and weaknesses of individual packages when dealing with specific applications.

All the experiments presented here were done on Intel Xeon Haswell processor node. Each node has 28 cores (2.6 GHz each) and 128 Gb of available memory. We consider the following versions of the languages:

Language	Version	Open Source?
Python	2.7.1	Yes
Julia	0.6.0	Yes
R	3.2.2	Yes
IDL	8.5	No
Matlab	R2016a	No
GNU Compilers	6.1.0	Yes
Intel Compilers	17.0.0.098	No
Scala	2.12.1	Yes

Remark: We assume that Python refers to Numpy too.

We consider here six problems.

Problem 1

Consider an arbitrary nxnx3 matrix A. We want to perform the following operations on A:

A(i,j,1) = A(i,j,2)

A(i,j,3) = A(i,j,1)

A(i,j,2) = A(i,j,3)

For instance, in Python the code looks like:

```
for i in range(n):
    for j in range(n):
        A[i,j,0] = A[i,j,1]
        A[i,j,2] = A[i,j,0]
        A[i,j,1] = A[i,j,2]
```

The above code segment uses loops. We are also interested on how the same operations are done using vectorization:

```
A[:,:,0] = A[:,:,1]
A[:,:,2] = A[:,:,0]
A[:,:,1] = A[:,:,2]
```

The problem allows us to see how each language handles loops and vectorization. We record the elapsed time needed to do the array assignments. The results are summarized on the tables below.

Language	Array/Matrix Storage	Option	n=5000	n=7000	n=9000
Python	Row-major		19.12	37.49	61.97
Python + Numba			0.25	0.22	0.30
Julia	Column-major		0.10	0.22	0.34
R	Column-major		233.78	451.77	744,93
IDL	Column-major		7.75	15.21	14.77
Matlab	Column-major		2.20	4.11	6.80
Fortran	Column-major	gfortran	0.23	0.33	0.76
		gfortran -O3	0.068	0.136	0.22
		ifort	0.07	0.18	0.29
		ifort -O3	0.068	0.136	0.22
C	Row-major	gcc	0.17	0.34	0.55
		gcc -Ofast	0.09	0.18	0.37
		icc	0.09	0.18	0.30
		icc -Ofast	0.09	0.18	0.42
Scala	Row-major		0.22	0.44	0.707

Table 1.1: Elapsed times obtained by copying a matrix using loops.

Language	Option	n=5000	n=7000	n=9000
Python		0.50	0.97	1.61
Python + Numba		0.36	0.54	0.77
Julia		0.46	0.73	1.09

R		3.55	6.95	11.46
IDL		0.56	1.60	2.93
Matlab		0.28	0.53	0.90
Fortran	gfortran	0.14	0.28	0.50
	gfortran -O3	0.096	0.18	0.30
	ifort	0.16	0.30	0.50
	ifort -O3	0.12	0.23	0.38

Table 1.2: Elapsed times obtained by copying a matrix using vectorization.

Apart from Julia, vectorization is the fastest method for accessing arrays/matrices.

Problem 2

We multiply two randomly generated *nxn* matrices A and B:

C=AxB

This problem shows the importance of taking advantage of built-in libraries available in each language.

The elapsed times presented here only measure the times spent on the multiplication (as the size of the matrix varies).

Language	Option	n=1500	n=1750	n=2000
Python	intrinsic	0.49	0.80	0.95

Python + Numba (loops)		3.6	6.28	13.4
Julia	intrinsic	0.54	0.63	0.73
R	intrinsic	12.09	19.18	28.63
IDL	intrinsic	0.22	0.28	0.36
Matlab	intrinsic	0.77	1.02	0.99
Fortran	gfortran (loop)	24.01	42.57	83.66
	gfortran -O3 (loop)	3.32	5.31	12.13
	gfortran (matmul)	1.58	2.52	4.34
	gfortran -O3 (matmul)	1.28	2.05	3.68
	ifort (loop)	1.55	2.01	4.48
	ifort -O3 (loop)	0.51	0.81	1.24
	ifort (matmul)	1.56	2.47	4.48
	ifort -O3 (matmul)	0.52	0.82	1.25
	ifort (DGEMM)	0.19	0.23	0.33
C	gcc (loop)	13.33	21.18	31.77
	gcc -Ofast (loop)	1.34	2.35	4.30
	icc (loop)	1.25	2.19	3.99
	icc -Ofast (loop)	1.23	1.72	2.62
Scala	Simple loop	10.76	18.22	32.30
	with la4j	4.056	6.354	9.592
	with JAMA	3.51	6.211	9.377

Table 2.1: Elapsed times (in seconds) obtained by multiplying two randomly generated matrices.

The above table suggests that built-in functions are more appropriate to perform matrix multiplication. *DGEMM* is far more efficient. It is important to note that *DGEMM* is more suitable for large size matrices. If for instance n=100, the function *matmul* out performs *DGEMM*. An interesting discussion on the performance of DGEMM and matmul using the Intel Fortran compiler can be read at:

How to calculate a multiplication of two matrices efficiently?

Problem 3

We find the numerical solution of the 2D Laplace equation:

$$U_{xx} + U_{yy} = 0$$

We use the Jacobi iterative solver. We are interested in fourth-order compact finite difference scheme (Gupta, 1984):

$$U_{i,j} = (4(U_{i-1,j} + U_{i,j-1} + U_{i+1,j} + U_{i,j+1}) + U_{i-1,j-1} + U_{i+1,j-1} + U_{i+1,j+1} + U_{i-1,j+1})/20$$

The Jacobi iterative solver stops when the difference of two consecutive approximations falls below 10^{-6}.

Language	Option	n=100	n=150	n=200
Python		144.54	715.96	2196.97
Python + Numba		1.23	5.37	16.34
Julia		1.049	5.253	18.00
	optimized_time_step	1.102	5.617	18.91
	optimized_time_step_simd	0.840	3.994	13.075
R		935.93	4560.91	-
IDL		95.48	498.23	1521.97
Matlab		5.06	12.50	23.40
Fortran	gfortran	1.21	5.56	15.64
	gfortran -O3	0.668	3.072	8.897

	ifort	0.38	2.15	6.10
	ifort -O3	0.536	2.46	7.15
C	gcc	0.51	2.47	7.85
	gcc -Ofast	0.21	1.04	3.18
	gcc -fPIC -Ofast -O3 -xc -shared	1.139	5.7001	18.318
	icc	0.45	2.23	6.78
	icc -Ofast	0.32	1.60	4.87
Scala		0.69	2.81	7.63

Table 3.1: Elapsed times (in seconds) obtained by numerically solving the Poisson equation using a Jacobi iterative solver with loops.

Language	Option	n=100	n=150	n=200
Python		2.52	11.66	47.13
Python + Numba		3.55	13.05	35.59
Julia		2.11	8.99	27.96
R		21.16	112.27	389.16
IDL		2.38	12.13	39.67
Matlab		3.57	8.01	16.17
Fortran	gfortran	0.872	4.032	11.53
	gfortran -O3	0.356	1.82	5.11
	ifort	0.288	1.568	4.568
	ifort -O3	0.288	1.560	4.284

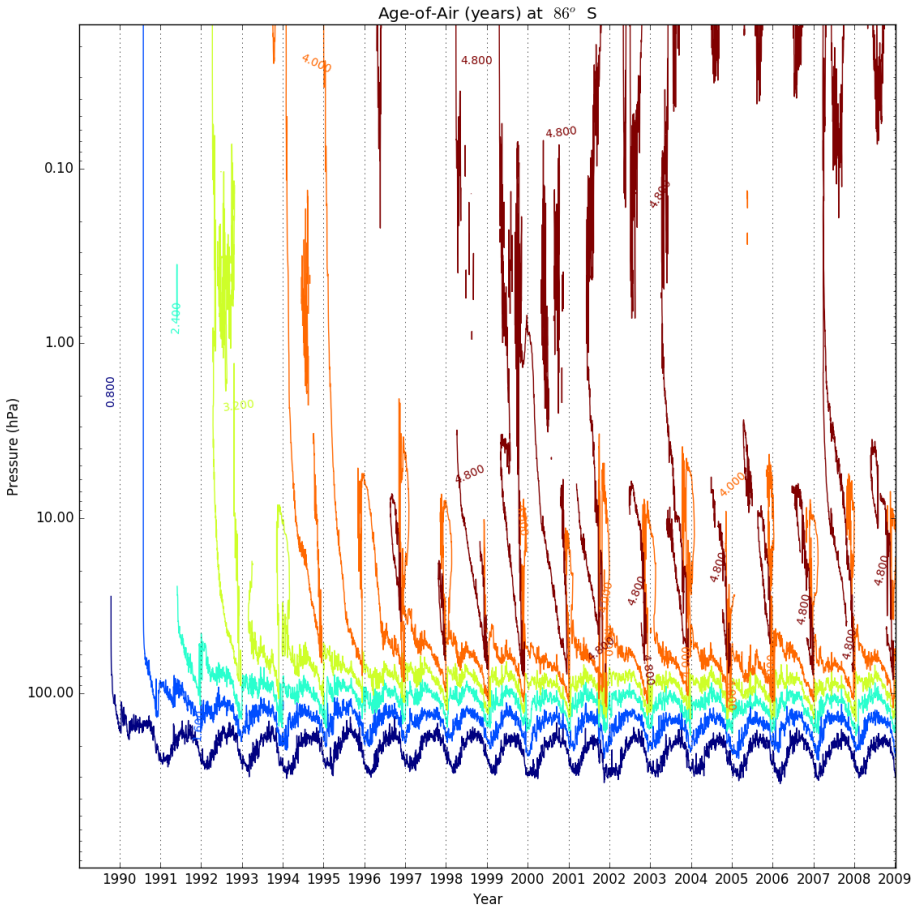
Table 3.2: Elapsed times (in seconds) obtained by numerically solving the Poisson equation using a Jacobi iterative solver with vectorization.

Problem 4

We have a set of daily NetCDF files (7305) covering a period of 20 years (1990-2009). The files for a given month are in a sub-directory labeled YYYYMM (for instance 199001, 199008, 199011). We want to write a script that opens each file, reads a three-dimensional variable (longitude/latitude/level), manipulates it and does a contour plot after all the files are read. A pseudo code for the script reads:

```
Loop over the years
    Loop over the months
        Obtain the list of NetCDF files
        Loop over the files
            Read the variable (longitude/latitude/level)
            Compute the zonal mean average (new array of latitude/level)
            Extract the column array at latitude 86 degree South
            Append the column array to a "master" array (or matrix)
create a contour plot using the "master" array
(the x-axis should be the days (1 to 7035)to be converted into years)
(the y-axis should be the vertical pressure levels in log scale)
```

A sample plot obtained with Python is shown in the figure below:



This is the kind of problems that a typical user we support faces: a collection of thousands of files that needs to be manipulated to extract the desired information. Having tools that allow us to quickly read data from files (in formats such as NetCDF, HDF4, HDF5, grib) is critical for the work we do.

We report in Table 4.1 the elapsed times it took to solve Problem 4 with the various languages.

Language	Elapsed time (s)
Python	1399

Julia	1317 (no plot)
IDL	1689
R	2220
Matlab	1678

Table 4.1: Elapsed time (in seconds) obtained by manipulating 7305 NetCDF files on a single processor. We were not able to produce the plot with Julia because we could not build the plotting tool.

All the above runs were conducted on a node that has 28 cores. Basically, only one core was used. We want to take advantage of all the available cores by spreading the reading of the files and making sure that the data of interest are gathered in the proper order. We use the multi-processing capabilities of the various languages to slightly modify the scripts. For each month, the daily files are read in by different threads (cores).The results are shown in Table 4.2. We were able to fully complete the task with Python, R and Julia only. The Julia script is fragile and we could run with 8 threads. We obtained unexpected error messages Matlab and could not resolve the issues (we will continue to look into it). We did not try to do the task in IDL because we could not find a simple IDL multi-processing documentation that could help us.

Language	**Elapsed time (s)**
Python	273
Julia	520 (no plot)
IDL	
R	420
Matlab	

Table 4.2: Elapsed time (in seconds) obtained by manipulating 7305 NetCDF files using multiple threading.

We observe that the use of multiple threads significantly reduces the processing time without requiring more resources (all the calculations were done within a node). The multi-thread processing scripts were written by making minor modifications of the serial ones. In fact, the multi-thread scripts ended up being more modular (use of functions) and more readable.

Problem 5

We implement the Belief Propagation calculations that can be seen as a repeated sequence of matrix multiplications, followed by normalization. The Matlab, C and Julia codes are shown in the Justin Domke's weblog (Domke 2012). We report the computing times for various values of the number of iterations (N) when the matrix dimension is 5000x5000.

Language	Option	N=250	N=500	N=1000
Python		5.01	9.5	18.8
Julia		4.4510	8.1424	15.137
R		42.302	82.963	164.33
IDL		15.472	30.085	59.318
Matlab		2.3930	4.5720	8.4749
Fortran	gfortran	5.1843	9.5965	18.457
	gfortran -O3	5.2243	9.7126	18.705
	ifort	4.6922	9.0045	20.881
	ifort -O3	4.6962	9.0245	17.709
C	gcc	3.8300	7.5700	15.160
	gcc -Ofast	3.5400	7.1000	14.240
	icc	1.7800	3.4400	6.9000
	icc -Ofast	1.7300	3.4600	6.9300
Scala				

Table 5.1: Elapsed times (in seconds) obtained by doing the Belief Propagation computations.

Problem 6

We perform calculations for the implementation of a Metropolis-Hastings algorithm using a two dimeensional distribution (Domke 2012). Results are shown when the number of iterations (N) varies.

Language	Option	N=5000	N=10000	N=15000
Python		0.02	0.05	0.08
Julia		0.000228	0.000441	0.000676
R		0.088	0.169	0.243
IDL		0.00960	0.01874	0.0393
Matlab		0.02619	0.06244	0.0870
Fortran	gfortran	0.00000	0.00000	0.00000
	gfortran -O3	0.00000	0.00000	0.00000
	ifort	0.00000	0.004	0.004
	ifort -O3	0.004	0.00000	0.008
C	gcc	0.00000	0.00000	0.00000
	gcc -Ofast	0.00000	0.00000	0.00000
	icc	0.00000	0.00000	0.00000
	icc -Ofast	0.00000	0.00000	0.00000
Scala		0.011	0.013	0.017

Table 6.1: Elapsed times (in seconds) obtained by doing the Metropolis algorithm computations.

Remarks:

1. All the experiments were done on a Linux cluster (with thousands of nodes) shared by hundreds of users.
2. We did not attempt to optimize any of the scripts we wrote. It is possible that developers of each languages may come with faster approaches to solve each of the problems presented here.
3. We also did the tests with Python 3.5 and we obtained the same results as in Python 2.7.
4. Using IDL and Matlab was difficult because at several occasions, there was not enough available licence.
5. When we install an open-source software, our preference is to do it from source because we have more control over the installation process (we can freely select any configuration we need). In addition, we want to be able to create a self-contained module (for instance Python together with Numpy, SciPy, Matplotlib, NetCDF4, etc.) and make it available to users. We are not sure that we can achieve it with Julia that seems to assume that each user is expected to add/build on his/her own packages on top of Julia.

References

1. Justin Domke, Julia, Matlab and C, September 17, 2012.
2. Michael Hirsch, Speed of Matlab vs. Python Numpy Numba CUDA vs Julia vs IDL, June 2016.
3. Murli M. Gupta, A fourth Order poisson solver, Journal of Computational Physics, 55(1):166-172, 1984.

4. Jean Francois Puget, A Speed Comparison Of C, Julia, Python, Numba, and Cython on LU Factorization, January 2016.
5. Alex Rogozhnikov, Log-likelihood benchmark, September 2015.
6. Sebastian Raschka, Numeric matrix manipulation - The cheat sheet for MATLAB, Python Nympy, R and Julia, June 2014.
7. Yousef Saad, Iterative Methods for Sparse Linear Systems (2 ed.), SIAM, ISBN 0898715342, 200366

Source Files

All the source files for the problems presented here are in the attached file: *sourceFiles.tar.gz*

If you have a comment/suggestion/question, contact Jules Kouatchou (Jules.Kouatchou@nasa.gov)