Basic Comparison of Python, Julia, Matlab, IDL and Java (2019 Edition)

Python, Julia, Java, Scala, IDL, Matlab, R, C, Fortran

Authors:

- Jules Kouatchou (jules.kouatchou@nasa.gov)
- Alexander Medema (alexander.medema@gmail.com)

NOTICE: This project is now Open-Source. All the source files are available github.com.

We plan to test the updated version of Julia in the future and add results with Python \Numba.

See the 2018 edition for previous source code.

Introduction

We use simple test cases to compare various high level programming languages. We implement the test cases from an angle of a novice programmer who is not familiar with the optimization techniques available in the languages. The goal is to highlight the strengths and weaknesses of each language but not to claim that one language is better than the others. Timing results are presented in seconds to four digits of precision, and any value less than 0.0001 is considered to be 0.

The tests presented here are run on an Intel Xeon Haswell processor node. Each node has 28 cores (2.6 GHz each) and 128 GB of available memory. The Python, Java, and Scala tests are also run on a Mac computer with an Intel i7-7700HQ (4 cores, 2.8 GHz each) with 16 GB of available memory to compare with the Xeon node. We consider the following versions of the languages:

Language	Version	Free?
Python	3.7	Yes
Julia	0.6.2	Yes
Java	10.0.2	Yes
Scala	2.13.0	Yes
IDL	8.5	No
R	3.6.1	Yes
Matlab	R2017b	No
GNU Compilers	9.1	Yes
Intel Compilers	18.0.5.274	No

The GNU and Intel compilers are used for C and Fortran. These languages are included to serve as a baseline, which is why their tests also come with optimized (-O3, -Ofast) versions.

The test cases are listed in four categories:

- Loops and Vectorization
- String Manipulations
- Numerical Calculations
- Input/Output

Each test is "simple" enough to be quickly written in any of the languages and is meant to address issues such as:

- Access of non-contiguous memory locations
- Use of recursive functions,
- Utilization of loops or vectorization,

- Opening of a large number of files,
- Manipulation of strings of arbitrary lengths,
- Multiplication of matrices,
- Use of iterative solvers
- etc.

The source files are contained in the directories:

```
C\    Fortran\  IDL\  Java\  Julia\  Matlab\  Python\  R\  Scala\
```

There is also a directory

```
Data\
```

that contains a Python script that generates the NetCDF4 files needed for the test case on reading a large collection of files. It also has sample text files for the "Count Unique Words in a File" test case.

Remark:

In the results presented below, we used an older version of Julia because we had difficulties installing the latest version of Julia (1.1.1) on the Xeon Haswell nodes. In addition, the Python experiments did not include Numba because the Haswell nodes we had access to, use an older version of the OS, preventing Numba to be properly installed.

Loops and Vectorization

- Copying Multidimensional Arrays

Given an arbitrary n x n x 3 matrix A, we perform the operations:

```
A(i, j, 1) = A(i, j, 2)
A(i, j, 3) = A(i, j, 1)
A(i, j, 2) = A(i, j, 3)
```

using loops and vectorization. This test case is meant to measure the speed of languages' access to non-contiguous memory locations, and to see how each language handles loops and vectorization.

Table CPA-1.0: Elapsed times to copy the matrix elements using loops on the Xeon node.

Language	Option	n=5000	n=7000	n=9000
Python		16.2164	31.7867	52.5485
Julia		0.0722	0.1445	0.2359
Java		0.1810	0.3230	0.5390
Scala		0.2750	0.4810	0.7320
IDL		6.4661	11.9068	19.4499
R		22.9510	44.9760	74.3480
Matlab		0.2849	0.5203	0.8461
Fortran	gfortran	0.1760	0.3480	0.5720
	gfortran -O3	0.0680	0.1720	0.2240
	ifort	0.0680	0.1360	0.2240
	ifort -O3	0.0680	0.1360	0.2800
C	gcc	0.1700	0.3400	0.5600
	gcc -Ofast	0.0900	0.1800	0.3100
	icc	0.1000	0.1800	0.3000
	icc -Ofast	0.1000	0.1800	0.3000

Table CPA-1.1: Elapsed times to copy the matrix elements using loops on the i7 Mac.

Language	n=5000	n=7000	n=9000
Python	18.6675	36.4046	60.2338
Python (Numba)	0.3398	0.3060	0.3693
Java	0.1260	0.2420	0.4190
Scala	0.2040	0.3450	0.5150

Table CPA-2.0: Elapsed times to copy the matrix elements using vectorization on the Xeon node.

Language	Option	n=5000	n=7000	n=9000
Python		0.4956	0.9739	1.6078
Julia		0.3173	0.5575	0.9191
IDL		0.3900	0.7641	1.2643
R		3.5290	6.9350	11.4400
Matlab		0.2862	0.5591	0.9188
Fortran	gfortran	0.0960	0.2520	0.3240
	gfortran -O3	0.0960	0.2440	0.3120
	ifort	0.1400	0.2280	0.3840
	ifort -O3	0.1200	0.2360	0.4560

Table CPA-2.1: Elapsed times to copy the matrix elements using vectorization on the i7 Mac.

Language	n=5000	n=7000	n=9000
Python	0.5602	1.0832	1.8077
Python (Numba)	0.8507	1.3650	2.0739

String Manipulations

- Look and Say Sequence

The look and say sequence reads a single integer. In each subsequent entry, the number of appearances of each integer in the previous entry is concatenated to the front of that integer. For example, an entry of

1223

would be followed by

112213,

or "one 1, two 2's, one 3." Here, we start with the number

1223334444

and determine the look and say sequence of order n (as n varies). This test case highlights how languages manipulate strings of arbitrary length.

Table LKS-1.0: Elapsed times to find the look and say sequence of order *n* on the Xeon node.

Language	Option	n=40	n=45	n=48

Python		2.0890	44.4155	251.1905
Java		0.0694	0.0899	0.1211
Scala		0.0470	0.1270	0.2170
IDL		20.2926	304.5049	1612.4277
Matlab		423.2241	6292.7255	exceeded time limit
Fortran	gfortran	0.0080	0.0120	0.0120
	gfortran -O3	0.0080	0.0120	0.0120
	ifort	0.0040	0.0160	0.0120
	ifort -O3	0.0080	0.0040	0.0080
C	gcc	0.0600	0.1900	0.4300
	gcc -Ofast	0.0400	0.1800	0.4000
	icc	0.0600	0.1900	0.4100
	icc -Ofast	0.0500	0.1900	0.4100

Table LKS-1.1: Elapsed times to find the look and say sequence of order n on the i7 Mac.

Language	n=40	n=45	n=48
Python	1.7331	22.3870	126.0252
Java	0.0665	0.0912	0.1543
Scala	0.0490	0.0970	0.2040

- Unique Words in a File

We open an arbitrary file and count the number of unique words in it with the assumption that words such as:

```
ab   Ab   aB    a&*(-b:    17;A#~!b
```

are the same (so that case, special characters, and numbers are ignored). For our tests, we use the four files:

```
world192.txt, plrabn12.txt, bible.txt, and book1.txt
```

taken from The Canterbury Corpus.

Table UQW-1.0: Elapsed times to count the unique words in the file on the Xeon node.

Language	world192.txt (19626 words)	plrabn12.txt (9408 words)	bible.txt (12605 words)	book1.txt (12427 words)
Python (dictionary method)	0.5002	0.1090	0.8869	0.1850
Python (set method)	0.3814	0.0873	0.7548	0.1458
Julia	0.2190	0.0354	0.3239	0.0615
Java	0.5624	0.2299	1.0135	0.2901
Scala	0.4600	0.2150	0.6930	0.2190
R	104.5820	8.6440	33.8210	17.6720
Matlab	3.0270	0.9657	6.0348	1.0390

Table UQW 1.1: Elapsed times to count the unique words in the file on the i7 Mac.

Language	world192.txt (19626 words)	plrabn12.txt (9408 words)	bible.txt (12605 words)	book1.txt (12427 words)

Python (dictionary method)	0.3541	0.0866	0.7346	0.1448
Python (set method)	0.3685	0.0820	0.7197	0.1417
Java	0.5129	0.2530	0.9183	0.3220
Scala	0.5810	0.1540	0.6650	0.2330

Numerical Computations

- Fibonacci Sequence

The Fibonacci Sequence is a sequence of numbers where each successive number is the sum of the two that precede it:

Fn = Fn-1 + Fn-2.

Its first entries are

F0 = 0, F1 = F2 = 1.

Fibonacci numbers find applications in the fields of economics, computer science, biology, combinatorics, etc. We measure the elapsed time when calculating an n^{th} Fibonacci number. The calculation times are taken for both iterative and recursive calculation methods.

Table FBC-1.0: Elapsed times to find the Fibonacci number using iteration on the Xeon node.

Language	Option	n=25	n=35	n=45
Python		0	0	0
Julia		0	0	0
Java		0	0	0
Scala		0	0	0
IDL		0	0	0
R		0.0330	0.0320	0.0320
Matlab		0.0026	0.0034	0.0038
Fortran	gfortran	0	0	0
	gfortran -O3	0	0	0
	ifort	0	0	0
	ifort -O3	0	0	0
C	gcc	0	0	0
	gcc -Ofast	0	0	0
	icc	0	0	0
	icc -Ofast	0	0	0

Table FBC-1.1: Elapsed times to find the Fibonacci number using iteration on the i7 Mac.

Language	n=25	n=35	n=45
Python	0	0	0
Python (Numba)	0.1100	0.1095	0.1099
Java	0	0	0
Scala	0	0	0

Table FBC-2.0: Elapsed times to find the Fibonacci number using recursion on the Xeon node.

Language	Option	n=25	n=35	n=45
Python		0.0593	7.0291	847.9716
Julia		0.0003	0.0308	3.787
Java		0.0011	0.0410	4.8192
Scala		0.0010	0.0560	5.1400
IDL		0.0238	2.5692	304.2198
R		0.0090	0.0100	0.0100
Matlab		0.0142	1.2631	149.9634
Fortran	gfortran	0	0.0840	10.4327
	gfortran -O3	0	0	0
	ifort	0	0	0
	ifort -O3	0	0	0
C	gcc	0	0.0400	5.0600
	gcc -Ofast	0	0.0200	2.2000
	icc	0	0.0300	3.1400
	icc -Ofast	0	0.0200	3.2800

Table FBC-2.1: Elapsed times to find the Fibonacci number using recursion on the i7 Mac.

Language	n=25	n=35	n=45
Python	0.0519	6.4022	800.0381
Python (Numba)	0.4172	43.7604	5951.6544
Java	0.0030	0.0442	5.0130
Scala	0.0010	0.0470	5.7720

- Matrix Multiplication

Two randomly generated *n* x *n* matrices *A* and *B* are multiplied. The time to perform the multiplication is measured. This problem shows the importance of taking advantage of built-in libraries available in each language.

Table MXM-1.0: Elapsed times to multiply the matrices on the Xeon node.

Language	Option	n=1500	n=1750	n=2000
Python	intrinsic	0.1560	0.2430	0.3457
Julia	intrinsic	0.1497	0.2398	0.3507
Java	loop	13.8610	17.8600	32.3370
Scala	loop	9.8380	19.1450	32.1310
R	intrinsic	0.1600	0.2460	0.3620
Matlab	intrinsic	1.3672	1.3951	0.4917
IDL	intrinsic	0.1894	0.2309	0.3258
Fortran	gfortran (loop)	17.4371	31.4660	62.1079
	gfortran -O3 (loop)	3.3282	5.3003	12.1648
	gfortran (matmul)	0.3840	0.6160	0.9241
	gfortran -O3 (matmul)	0.3880	0.6160	0.9161
	ifort (loop)	1.1401	1.8161	2.9282
	ifort -O3 (loop)	1.1481	1.8081	2.9802
	ifort (matmul)	1.1441	1.8121	2.9242
	ifort -O3 (matmul)	0.5160	0.8281	1.2441
	ifort (DGEMM)	0.2160	0.2360	0.3320
C	gcc (loop)	13.2000	20.9800	31.4400
	gcc -Ofast (loop)	1.4500	2.3600	4.0400
	icc (loop)	1.2300	2.1500	4.0500
	icc -Ofast (loop)	1.1500	1.7500	2.5900

Table MXM-1.1: Elapsed times to multiply the matrices on the i7 Mac.

Language	Option	n=1500	n=1750	n=2000
Python	intrinsic	0.0906	0.1104	0.1611
	Numba (loop)	9.2595	20.2012	35.3174
Java	loop	32.5080	47.7680	82.2810
Scala	loop	23.0540	38.9110	60.3180

- Belief Propagation Algorithm

Belief propagation is an algorithm used for inference, often in the fields of artificial intelligence, speech recognition, computer vision, image processing, medical diagnostics, parity check codes, and others. We measure the elapsed time when performing n iterations of the algorithm with a 5000x5000-element matrix. The Matlab, C and Julia code is shown in Justin Domke's weblog (Domke 2012), which states that the algorithm is "a repeated sequence of matrix multiplications, followed by normalization."

Table BFP-1.0: Elapsed time to run the belief propagation algorithm on the Xeon node.

Language	Option	n=250	n=500	n=1000
Python		3.7076	7.0824	13.8950
Julia		4.0280	7.8220	15.1210
Java		63.9240	123.3840	246.5820
Scala		53.5170	106.4950	212.3550
IDL		16.9609	33.2086	65.7071
R		23.4150	45.4160	89.7680
Matlab		1.9760	3.8087	7.4036
Fortran	gfortran	21.0013	41.0106	87.6815
	gfortran -O3	4.4923	8.2565	17.5731
	ifort	4.7363	9.1086	17.8651

	ifort -O3	4.7363	9.1086	21.1973
C	gcc	2.6400	5.2900	10.5800
	gcc -Ofast	2.4200	4.8500	9.7100
	icc	2.1600	4.3200	8.6500
	icc -Ofast	2.1800	4.3400	8.7100

Table BFP-1.1: Elapsed time to run the belief propagation algorithm on the i7 Mac.

Language	n=250	n=500	n=1000
Python	2.4121	4.5422	8.7730
Java	55.3400	107.7890	214.7900
Scala	47.9560	95.3040	189.8340

- Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm is an algorithm used to take random samples from a probability distribution. This implementation uses a two-dimensional distribution (Domke 2012), and measures the elapsed time to iterate n times.

Table MTH-1.0: Elapsed times to run the Metropolis-Hastings algorithm on the Xeon node.

Language	Option	n=5000	n=10000	n=15000
Python		0.0404	0.0805	0.1195
Julia		0.0002	0.0004	0.0006
Java		0.0040	0.0050	0.0060
Scala		0.0080	0.0090	0.0100

IDL		0.0134	0.0105	0.0157
R		0.0760	0.1500	0.2230
Matlab		0.0183	0.0211	0.0263
Fortran	gfortran	0	0	0
	gfortran -O3	0	0	0
	ifort	0.0040	0	0
	ifort -O3	0.0040	0.0040	0
C	gcc	0	0	0
	gcc -Ofast	0	0	0
	icc	0	0	0
	icc -Ofast	0	0	0

Table MTH-1.1: Elapsed times to run the Metropolis-Hastings algorithm on the i7 Mac.

Language	n=5000	n=10000	n=15000
Python	0.0346	0.0638	0.0989
Java	0.0060	0.0040	0.0060
Scala	0.0090	0.0100	0.0130

- Fast Fourier Transform

We create an *n* x *n* matrix *M* that contains random complex values. We the compute the Fast Fourier Transform (FFT) of M and the absolute value of the result. The FFT algorithm is used for signal processing and image processing in a wide variety of scientific and engineering fields.

Table FFT-1.0: Elapsed times to compute the FFT on the Xeon node.

Language	Option	n=10000	n=15000	n=20000
Python	intrinsic	8.0797	19.6357	34.7400
Julia	intrinsic	3.979	11.490	20.751
IDL	intrinsic	16.6699	38.9857	70.8142
R	intrinsic	58.2550	150.1260	261.5460
Matlab	intrinsic	2.6243	6.0010	10.66232

Table FFT-1.1: Elapsed times to compute the FFT on the i7 Mac.

Language	Option	n=10000	n=15000	n=20000
Python	intrinsic	7.9538	21.5355	55.9375

- ## Iterative Solver

We use the Jacobi iterative solver to numerically approximate a solution of the two-dimensional Laplace equation that was discretized with a fourth order compact scheme (Gupta, 1984). We record the elapsed time as the number of grid points varies.

Table ITS-1.0: Elapsed times to compute the approximate solution using iteration on the Xeon node.

Language	Option	n=100	n=150	n=200
Python		158.2056	786.3425	2437.8560
Julia		1.0308	5.1870	16.1651
Java		0.4130	1.8950	5.2220
Scala		0.540	2.1030	5.7380

IDL		73.2353	364.1329	1127.1094
R		157.1490	774.7080	2414.1030
Matlab		2.8163	5.0543	8.6276
Fortran	gfortran	0.8240	3.7320	10.7290
	gfortran -O3	0.6680	3.0720	8.8930
	ifort	0.5400	2.4720	7.1560
	ifort -O3	0.5400	2.4680	7.1560
C	gcc	0.5000	2.4200	7.7200
	gcc -Ofast	0.2200	1.0500	3.1900
	icc	0.4600	2.2300	6.7800
	icc -Ofast	0.3300	1.6000	4.8700

Table ITS-1.1: Elapsed times to compute the approximate solution using iteration on the i7 Mac.

Language	n=100	n=150	n=200
Python	174.7663	865.1203	2666.3496
Python (Numba)	1.3226	5.0324	15.1793
Java	0.4600	1.7690	4.7530
Scala	0.5970	2.0950	5.2830

Table ITS-2.0: Elapsed times to compute the approximate solution using vectorization on the Xeon node.

Language	Option	n=100	n=150	n=200
Python		2.6272	14.6505	40.2124
Julia		2.4583	13.1918	41.0302
IDL		1.71192	8.6841	28.0683

R		25.2150	121.9870	340.4990
Matlab		3.3291	7.6486	15.9766
Fortran	gfortran	0.8680	4.2040	11.5410
	gfortran -O3	0.3600	1.8040	5.0880
	ifort	0.2800	1.5360	4.4560
	ifort -O3	0.2800	1.5600	4.4160

Table ITS-2.1: Elapsed times to compute the approximate solution using vectorization on the i7 Mac.

Language	n=100	n=150	n=200
Python	1.7051	7.4572	22.0945
Python (Numba)	2.4451	8.5094	21.7833

- Square Root of a Matrix

Given an *n* x *n* matrix *A*, we are looking for the matrix *B* such that:

B * B = A

B is the square root. In our calculations, we consider *A* with 6s on the diagonal and 1s elsewhere.

Table SQM-1.0: Elapsed times to calculate the square root of the matrix on the Xeon node.

Language	n=1000	n=2000	n=4000
Python	1.0101	5.2376	44.4574
Julia	0.4207	2.5080	19.0140
R	0.5650	3.0660	19.2660
Matlab	0.3571	1.6552	2.6250

Table SQM-1.1: Elapsed times to calculate the square root of the matrix on the i7 Mac.

Language	n=1000	n=2000	n=4000
Python	0.5653	3.3963	25.9180

- Gauss-Legendre Quadrature

Gauss-Legendre quadrature is a numerical method for approximating definite integrals. It uses a weighted sum of n values of the integrand function. The result is exact if the integrand function is a polynomial of degree 0 to 2n - 1. Here we consider an exponential function over the interval [-3, 3] and record the time to perform the integral when n varies.

Table GLQ-1.0: Elapsed times to find the approximate value of the integral on the Xeon node.

Language	Option	n=50	n=75	n=100
Python		0.0079	0.0095	0.0098

Julia		0.0002	0.0004	0.0007
IDL		0.0043	0.0009	0.0014
R		0.0260	0.0240	0.0250
Matlab		0.7476	0.0731	0.4982
Fortran	gfortran	0	0.0040	0.0080
	gfortran -O3	0	0.0120	0.0120
	ifort	0.0080	0.0080	0.0080
	ifort -O3	0.0080	0.0040	0.0080

Table GLQ-1.1: Elapsed times to find the approximate value of the integral on the i7 Mac.

Language	n=50	n=75	n=100
Python	0.0140	0.0035	0.0077

- Trigonometric Functions

We iteratively calculate trigonometric functions on an *n*-element list of values, and then compute inverse trigonometric functions on the same list. The time to complete the full operation is measured as n varies.

Table TRG-1.0: Elapsed times to evaluate the trigonometric functions on the Xeon node.

Language	Options	n=80000	n=90000	n=100000
Python		14.6891	16.5084	23.6273
Julia		55.3920	62.9490	69.2560
IDL		37.4413	41.9695	35.2387

R		91.5250	102.8720	113.8600
Matlab		5.2794	5.8649	6.3699
Scala		357.3730	401.8960	446.7080
Java		689.6560	774.9110	865.057
Fortran	gfortran	53.4833	60.0317	66.6921
	gfortran -O3	49.9271	56.0235	62.1678
	ifort	18.6411	20.9573	23.2654
	ifort -O3	18.6451	20.9573	23.2694
C	gcc	107.4400	120.7300	134.0900
	gcc -Ofast	93.0400	104.5700	116.0600
	icc	76.2600	85.7900	95.3100
	icc -Ofast	48.8400	54.9600	61.0600

Table TRG-1.1: Elapsed times to evaluate the trigonometric functions on the i7 Mac.

Language	n=80000	n=90000	n=100000
Python	3.5399	6.1984	6.9207

- Munchausen Numbers

A Munchausen number is a natural number that is equal to the sum of its digits raised their own power. In base 10, there are four such numbers: 0, 1, 3435 and 438579088. We determine how much time it takes to find them.

Table MCH-1.0: Elapsed times to find the Munchausen numbers on the Xeon node.

Language	Option	Elapsed time

Python		1130.6220
Julia		102.7760
Java		4.9008
Scala		72.9170
R		exceeded time limit
IDL		exceeded time limit
Matlab		373.9109
Fortran	gfortran	39.7545
	gfortran -O3	21.3933
	ifort	29.6458
	ifort -O3	29.52184
C	gcc	157.3500
	gcc -Ofast	126.7900
	icc	228.2300
	icc -Ofast	228.1900

Table MCH-1.1: Elapsed times to find the Munchausen numbers on the i7 Mac.

Language	Elapsed time
Python	1013.5649
Java	4.7434
Scala	64.1800

Input/Output

- Reading a Large Collection of Files

We have a set of daily NetCDF files (7305) covering a period of 20 years. The files for a given year are in a sub-directory labeled YYYY (for instance Y1990, Y1991, Y1992, etc.). We want to write a script that opens each file, reads a three-dimensional variable (longitude/latitude/level), and manipulates it. Pseudocode for the script reads:

```
Loop over the years
    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)
```

The goal is to be able to do a generate the three-dimensional arrays (year/level/value) and carry out a contour plot. This is the type of problem that a typical user we support faces: a collection of thousands of files that need to be manipulated to extract the desired information. Having tools that can quickly read data from files (in formats such as NetCDF, HDF4, HDF5, grib) is critical for the work we do.

Table RCF-1.0: Elapsed times to process the NetCDF files on the Xeon node.

Language	Elapsed time
Python	660.8084
Julia	787.4500
IDL	711.2615
R	1220.222
Matlab	848.5086

Table RCF-1.1: Elapsed times to process the NetCDF files on the i7 Mac.

Language	Elapsed time
Python	89.1922

Table RCF-2.0: Elapsed times to process the NetCDF files with Python using multiple cores on the Xeon node.

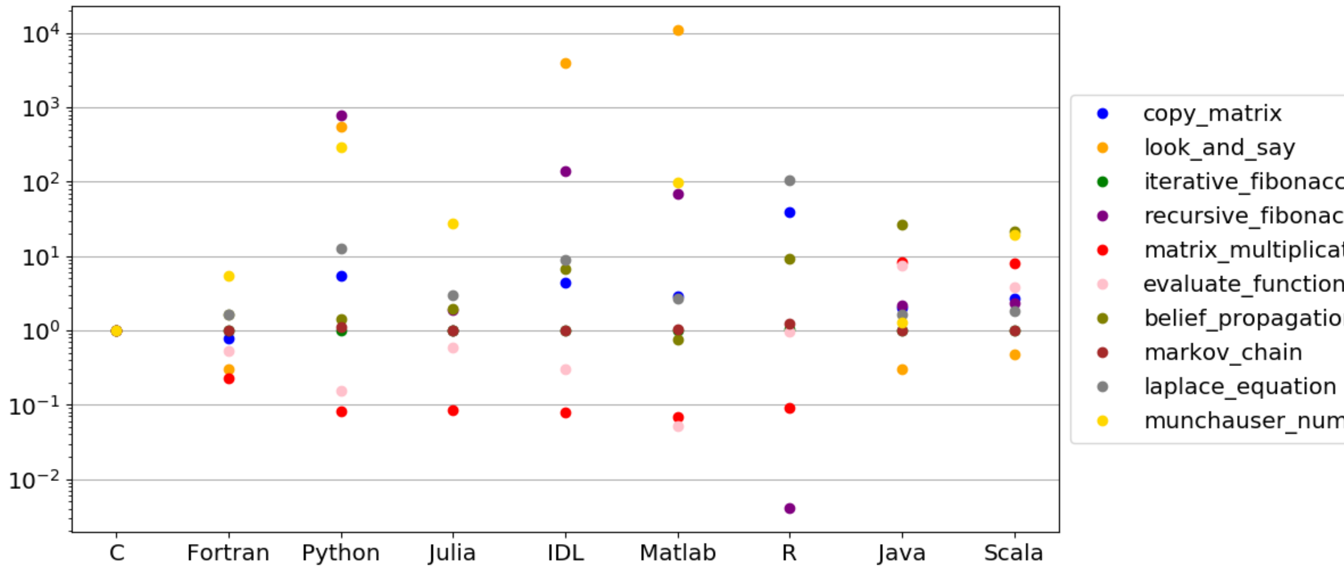
Cores	Elapsed time
1	570.9791
2	317.6108
4	225.4647
8	147.4527
16	84.0102
24	59.7646
28	51.2191

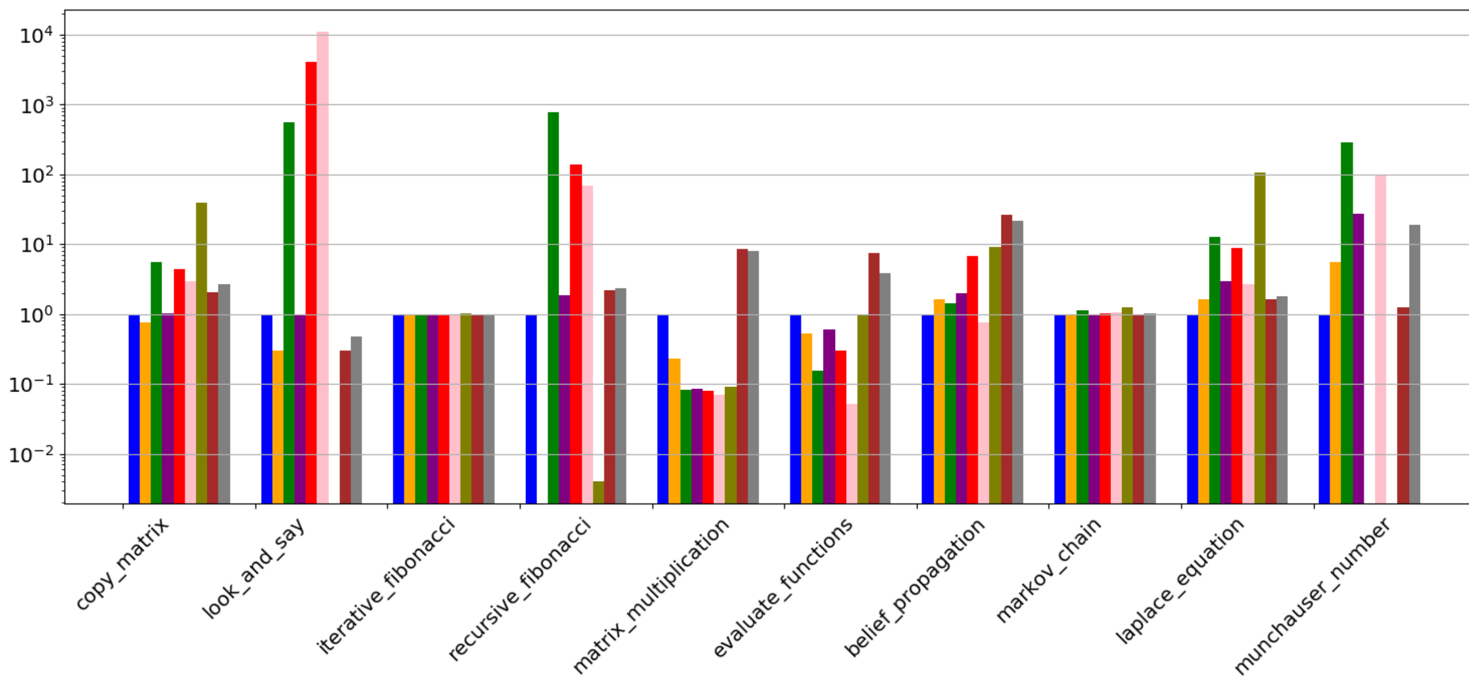
Table RCF-2.1: Elapsed times to process the NetCDF files with Python using multiple cores on the i7 Mac.

Cores	Elapsed time
1	84.1032
2	63.5322
4	56.6156

Summary with a Plot

In the plots below, we summarize the above timing results by using as reference the timing numbers (last column only, i.e., largest problem size) obtained with GCC.


10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
C
Fortran
Python
Julia
IDL
Matlab
R
Java
Scala
copy_matrix
look_and_say


10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
copy_matrix
look_and_say
iterative_fibonacci
recursive_fibonacci
matrix_multiplication
evaluate_functions
belief_propagation
markov_chain
laplace_equation
munchauser_number

Findings

General:

- No single language outperforms the others in all tests.
- It is important to reduce the memory footprint by creating variables only when necessary and by "emptying" variables that are no longer used.
- Using intrinsic functions results in improved performance compared to inline code for the same task.
- Julia and R offer simple benchmarking tools. We wrote a simple Python tool that allows us to run Python test cases as many times as we wish.

Loops and Vectorization:

- Python (and Numpy), IDL, and R consistently run more quickly when vectorized compared to when using loops.
- When using Numba, Python is faster with loops as long as Numpy arrays are used.
- With Julia, loops run more quickly than vectorized code.
- Matlab does not appear to change significantly in performance when using loops versus vectorization in a case that involves no calculations. When calculations are performed, vectorized Matlab code is faster than iterative code.

String Manipulations:

- Java and Scala appear to have notable performance relative to the other languages when manipulating large strings.

Numerical Calculations:

- R appears to have notable performance relative to the other languages when using recursion.
- Languages' performance in numerical calculation relative to the others depends on the specific task.
- Matlab's intrinsic FFT function seems to run the most quickly.

Input/Output:

- While some of the languages run the test more quickly than others, running the test on a local Mac instead of the processor node results in the largest performance gain. The processor node uses hard drives, whereas the Mac has a solid-state disk. This indicates that hardware has a larger impact on I/O performance than the language used.

Acknowledgements

This work partially funded by Michigan Space Grant Consortium, NASA grant #NNX15AJ20H.

References

1. Justin Domke, Julia, Matlab and C, September 17, 2012.
2. Murli M. Gupta, A fourth Order poisson solver, Journal of Computational Physics, 55(1):166-172, 1984.

Source Files

We are currently working on making the files open-source.