BOSTON | NY | LONDON | BANGALORE

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

BENCHMARKING JULIA AGAINST PYTHON

INTRODUCTION

This document provides several benchmarks that were conducted by Julia Computing.

BENCHMARK AGAINST PYTHON 1: CIRCUITSCAPE

Circuitscape is a tool that borrows algorithms from electronic circuit theory to measure connectivity in heterogeneous landscapes. Its most common applications include modelling movement and gene flow of plants and animals, as well as identifying areas important for connectivity conservation. Circuit theory complements commonly-used connectivity models because of its connections to random walk theory and its ability to simultaneously evaluate contributions of multiple dispersal pathways. Landscapes are represented as conductive surfaces, with low resistances assigned to landscape features types that are most permeable to movement or best promote gene flow, and high resistances assigned to movement barriers."

"Circuitscape v4.0 is a Python package implemented primarily using NumPy, SciPy and PyAMG. However, this version faced significant limitations in terms of speed and scalability. Circuitscape v5.0 has been reimplemented in the Julia language for speed, efficiency and scalability. According to our benchmarks it is between 4x - 8x faster than v4.0 (benchmark chart attached)

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

SIZE	PYTHON	JULIA	JULIA- CHOLMOD
1m	537.810477	106.395484	89.60318678
6m	4711.366189	1217.9016	543.0623558
12m	9486.912186	2337.54783	1124.275759
24m	20902.11244	4831.22026	
48m		10149.4611	

Hardware Used:
- Name: Intel(R) Xeon(R) Silver 4114 CPU
- Clock Speed: 2.20GHz
- Number of cores: 20
- RAM: 384 GB

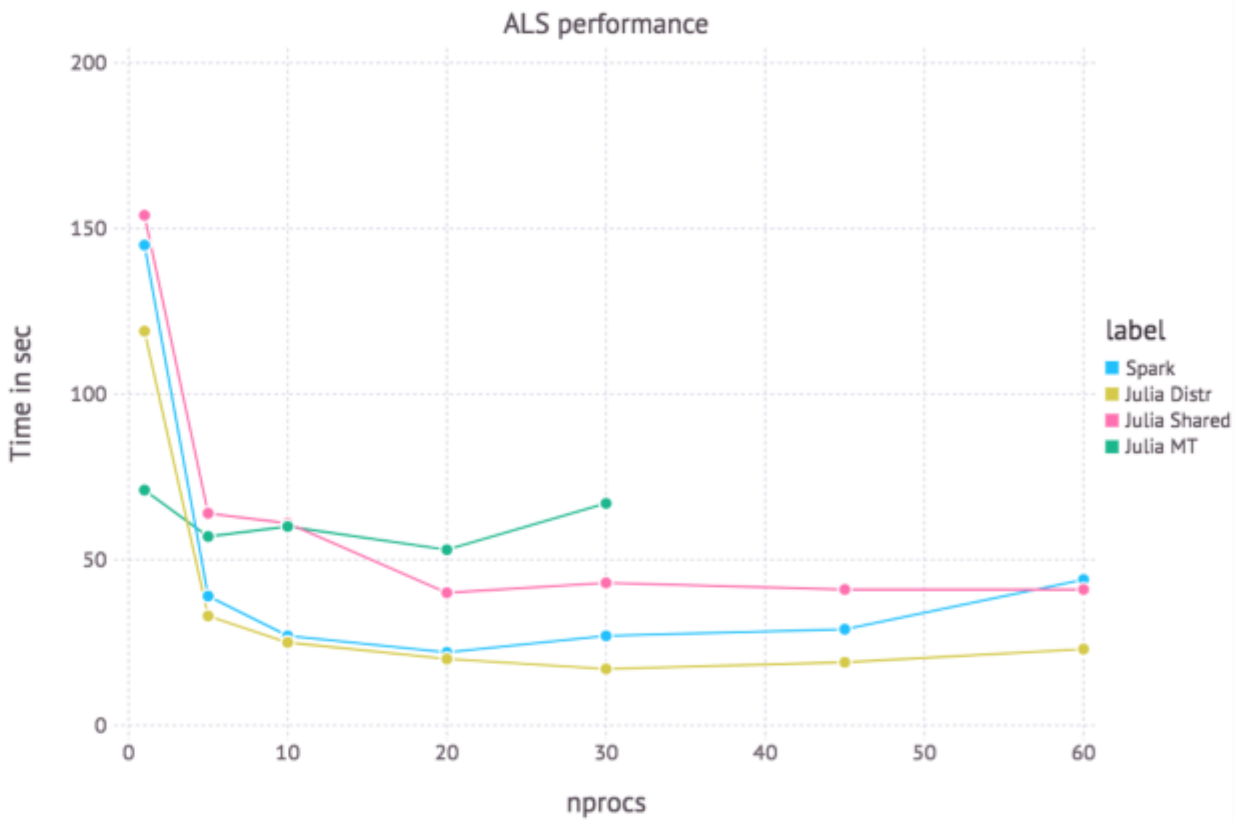
BENCHMARK AGAINST PYTHON 2: RECOMMENDER SYSTEM

The package RecSys.jl is a package for recommender systems in Julia, it can currently work with explicit ratings data. For preparing the input create an object of ALSWRtype. This takes two input parameters, firstly input file location, and second optional input is the variable par which specifies the type of parallelism. The parallelism is about how the data is shared/distributed across the processing units. When par=ParShemm the data is present at one location and is shared across the processing units, when par=ParChunk the data is distributed across the processing units as chunks. For this report only sequential timings were captured, i.e., with nprocs=1.

Parallelism is made possible in Julia mainly 2 ways, a). Multiprocessing and b). Multithreading. The multithreading development is ongoing. However, the multiprocessing based parallel processing in Julia is mature and mainly based around Tasks which are concurrent function calls. The implementation details are not covered here, the following graph summarises the performance of parallel ALS implementation in Julia and Spark:

BOSTON | NY | LONDON | BANGALORE

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com



BENCHMARK AGAINST PYTHON 3: CALCULATING PI DIGITS

Using the Borwein's algorithm with quadratic convergence to the approximation of pi, (using a discriminant of the Ramanujan-Sato Series), we try to approximate pi to roughly 10 million digits. This algorithm uses simple calculations and iteratively approximates the value of pi.

This is to see how the languages compare in terms of calculations that may not be too heavy, but large in number. This mimics how calculations would happen behind the scenes when a machine learning model is fit.

The programs are run single threaded.

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

BOSTON | NY | LONDON | BANGALORE

PYTHON

```
import time, timeit
def pi_calc(n = 10**7):
        a0 = 2**0.5
        b0 = 0
        p0 = 2 + 2**0.5
        for i in xrange(n):
        a1 = (a0 + a0**-0.5)/2.0
        b1 = ((1 + b0)*(a0**0.5))/(a0 + b0)
        p1 = ((1 + a1)*p0*b1)/(1 + b1)
        a0 = a1
        b0 = b1
        p0 = p1
print(timeit.timeit(pi_calc, number = 1))
```

JULIA

```
function pi_calc(n)
        a0 = 2^0.5
        b0 = 0
        p0 = 2 + 2^0.5
        for i in 1:n
        a1 = (a0 + a0^-0.5)/2.0
        b1 = ((1 + b0)*(a0^0.5))/(a0 + b0)
        p1 = ((1 + a1)*p0*b1)/(1 + b1)
        a0 = a1
        b0 = b1
        p0 = p1
        end
end

@time pi_calc(10^7)
```

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

RESULTS:

LANGUAGE	TIME (SECONDS)
PYTHON	4.27
JULIA	0.82

BENCHMARK AGAINST PYTHON 4: MANDELBROT SET

The Mandelbrot Set is a mathematical object known as a fractal which converges upon itself indefinitely. It is often used to benchmark programming languages for their performance as it involves non-trivial operations with the complex number space and high precision calculations. The Mandelbrot Set makes the following mapping:

z = z^2 + c

We will make use of the numpy library in python (C bindings) to demonstrate a typical scenario when non trivial workflow is taken into consideration. Numba, which provides JIT compilation can provide further speedup, but cannot be used in all kinds of cases.

PYTHON

```
import numpy as np

def mandelbrot(c,maxiter):
        z = c
        for n in xrange(maxiter):
        if abs(z) > 2:
        return n
        z = z*z + c
        return 0
```

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com


```
def mandelbrot_set(xmin = -0.74877,
        xmax = -0.74872,
        ymin = 0.06505,
        ymax = 0.06510,
        width = 1000,
        height = 1000,
        maxiter = 2048):
        r1 = np.linspace(xmin, xmax, width)
        r2 = np.linspace(ymin, ymax, height)
        n3 = np.empty((width,height))

        start = time.time()
        for i in range(width):
        for j in range(height):
        n3[i,j] = mandelbrot(r1[i] + 1j*r2[j],maxiter)
        print(time.time() - start)

        return (r1,r2,n3)

print(timeit.timeit(mandelbrot_set, number = 1))
```

JULIA

```
function mandelbrot(c,maxiter)
        z = c
        for n in 1:maxiter
                if abs(z) > 2
                                return n
                end
                z = z*z + c
        end
        return 0
end
```

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

```
function mandelbrot_set(xmin = -0.74877,
        xmax = -0.74872,
        ymin = 0.06505,
        ymax = 0.06510,
        width = 1000,
        height = 1000,
        maxiter = 2048)
        r1 = linspace(xmin, xmax, width)
        r2 = linspace(ymin, ymax, height)
        n3 = zeros(Float32, width,height)
        for i in 1:width
                for j in 1:height
                                n3[i,j] = mandelbrot(r1[i] + r2[j]im, maxiter)
                end
        end
        return (r1,r2,n3)
end

@time mandelbrot_set()
```

RESULTS:

LANGUAGE	TIME (SECONDS)
PYTHON	212
JULIA	3.72

Julia computing
BOSTON | NY | LONDON | BANGALORE

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

BENCHMARK AGAINST PYTHON 5: COIN TOSS

The coin toss problem is one of the classics of testing the performance of a highly parallelizable problem set. Here we simulate tossing a coin 1 billion times to highlight how multithreading behaves in Python and how it behaves in Julia. Special focus must be presented as both the single threaded as well as the multi threaded versions of the code are presented along with run times. Also, note the changes between the two, vis a vis the ability to multi thread on the fly.

Both Python and Julia were run with 4 workers to maintain parity.

PYTHON - SINGLE THREADED

```
def coin_toss(n = 10**9):
        res = [0]*n
        for i in range(n):
        res[i] = randint(0,1)

        return res
print(timeit.timeit(coin_toss, number = 1))
```

JULIA - SINGLE THREADED

```
function coin_toss(n::Int64, i::Int64 = 1)
        a = Array{Int8}(n)
        for i in 1:n
                a[i] = Int8(rand(0:1))
        end
        a
end
```

Julia computing
BOSTON | NY | LONDON | BANGALORE

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

RESULTS:

LANGUAGE	TIME (SECONDS)
PYTHON	1804
JULIA	28.8

PYTHON - MULTITHREADED

```
from random import randint
import time

def toss(start):
        global res
        global part
        print(start)
        for i in range(start, part + start):
                res[i] = randint(0,1)

def coin_toss(part, nthreads, n = 10**9):
        pool = ThreadPool(nthreads)
        results = pool.map(toss, range(0, n, part))
        return results

if __name__ == '__main__':
        n = 10**9
        nthreads = 4
        res = [0]*n
        part = int(n/nthreads)
        start = time.time()
        coin_toss(part = part, nthreads = nthreads)
        print(time.time() - start)
```

BOSTON | NY | LONDON | BANGALORE

JULIA - MULTITHREADED

```
@everywhere function coin_toss(n::Int64, i::Int64 = 1)
	a = SharedArray{Int8}(n)
	@parallel for i in 1:n
			a[i] = Int8(rand(0:1))
	end
	a
end

@time coin_toss(10^9)
```

RESULTS:

LANGUAGE	TIME (SECONDS)
PYTHON	920
JULIA	2.44

BENCHMARK AGAINST PYTHON 6: MATRIX MULTIPLICATION

The objective is to compare Python's and Julia's ability to parallelize a simple procedure like matrix multiplication. We will be using the straightforward ijk algorithm to perform matrix multiplication. The time and code shows how fast and easy it is to parallelize procedures in Julia. Essentially the procedure performs $C=A*B$
The ijk algorithm is an iterative one, each entry in C is calculated as Cik=j=1naij*bjk

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

MULTIPLICATION ON SINGLE CORE

JULIA

```
function ijk(A::Array{Float64}, B::Array{Float64}, C::Array{Float64})
    @inbounds for i=1:size(A)[1]
      for k=1:size(B)[2]
        for j=1:size(A)[2]
          C[i,k]+= A[i,j]*B[j,k]
        end
      end
    end
    C
  end


function perform_ijk(n::Int64)
  A = randn((n,n))
  B = randn((n,n))
  C = zeros((n,n))
  tic()
  ijk(A,B,C)
  toc()
end
```

Julia computing
BOSTON | NY | LONDON | BANGALORE

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

PYTHON

```
import numpy as np

from time import time as t

def ijk(A,B,C):

    for i in range(A.shape[0]):

        for k in range(B.shape[1]):

            for j in range(A.shape[1]):

                C[i,k] += A[i,j]*B[j,k]

    return C

if __name__ == "__main__":

    A = np.random.normal(size=(1000,1000))

    B = np.random.normal(size=(1000,1000))

    C = np.zeros((1000,1000))

    start = t()

    C = ijk(A,B,C)

    print("elapsed time: {0} seconds".format(t()-start))
```

RESULTS

LANGUAGE	TIME (SECONDS)
PYTHON	800
JULIA	1.38

Julia computing
BOSTON | NY | LONDON | BANGALORE

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

MULTIPLICATION ON MULTIPLE (4) CORES

JULIA

```
addprocs(4) #adding 4 processes

#the following defines function on all processes

@everywhere function matmul_multicore(n,w,A,B,C)

  range = 1+(w-2) * div(n,4) : (w-1) * div(n,4)

  @inbounds for i=range

    for k=1:size(B)[2]

      for j=1:size(A)[2]

        C[i,k]+= A[i,j]*B[j,k]

      end

    end

  end

end


function perform_ijk_multicore(n::Int64)

  A = SharedArray{Float64}(randn(n,n));

  B = SharedArray{Float64}(randn(n,n));

  C = SharedArray{Float64}((n,n));

  tic()
```

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

```
    @sync begin
    for w in workers()
        @async remotecall_wait(matmul_multicore, w, n, w, A, B, C)
    end
end
toc()
end
```

PYTHON

```
import multiprocessingimport numpy as np
from time import time as t
def lineMult(start):
    global A, B, C, part
    n = len(A)
    for i in range(start, start+part):
        for k in range(n):
            for j in range(n):
                C[i,j] += A[i,k] * B[k,j]

def ikjMatrixProduct(A, B, threadNumber):
    n = len(A)
    pool = multiprocessing.Pool(threadNumber)
```

Julia Computing, Inc.
45 Prospect St., Cambridge, MA 02139
Email: info@juliacomputing.com
Web: www.juliacomputing.com

```
    pool.map(lineMult, range(0,n, part))
    return C

if __name__ == "__main__":
    A = np.random.normal(size=(1000,1000))
    B = np.random.normal(size=(1000,1000))
    C = np.zeros((1000,1000))
    n, m, p = len(A), len(A[0]), len(B[0])

    threadNumber = 4
    part = int(len(A) / threadNumber)
    start = t()
    C = ikjMatrixProduct(A, B, threadNumber)
    print("elapsed time:  {0} seconds".format(t()-start)
```

RESULTS

LANGUAGE	TIME (SECONDS)
PYTHON	436
JULIA	2.27