Julia: A New Language for Scientific Computing

http://freemind.pluskid.org/technology/julia-a-new-language-for-scientific-computing

posted on Free Mind on August 10, 2013
generated with pandoc on December 3, 2015
category: Technology

tags: Tools, Julia, Python

Julia 是一门相对比较新的着眼于科学计算的语言，语法上看起来有点类似于 Matlab 的脚本语言，但是实际上却是从 Ruby、Python、Lisp 之类的语言里吸收了许多有趣的特性。其主页上的描述是

> *Julia is a high-level, high-performance dynamic programming language...*

当然卖点主要在于 high-level 和 high-performance 了。在主页上有一个 benchmark showcase 的表格，可以看到 Julia 的性能几乎接近 C 的性能。当然这种语言之间的 Benchmark 做出来总是会被各个语言社区的人批的，因为不同的语言实现同一个算法有很多种不同的方法，有的性能差别可以非常大。但是无论如何这些数字都还是相当 impressive 的。

如果真的是要做到极致性能的话，像我最近接触过的一些做 high performance computing 的人，他们都是把算法实现退到 C 或者 Fortran 级别然后再做非常细致的分析和优化的；但是在科学研究中虽然经常性能是重点，快速 prototyping 却是重中之重。所以像 Matlab、R、Python 这样的 high-level 语言通常会成为首选。Julia 的定位也是在这里。事实上，Julia 的语法非常像 Matlab，除了不需要在行尾加上分号以外，基本的运算、矩阵操作之类的几乎一样，更过分的是甚至连数组下标都和 Matlab 一样是从 1 开始数的，所以说 Matlab 用户大概扫一下 Manual 差不多就可以上手了。例如下面这段 benchmark 里的测试计算随机矩阵的 statistics 的 Matlab 代码：

```
function [s1, s2] = randmatstat(t)
    n=5;
    v = zeros(t,1);
    w = zeros(t,1);
    for i=1:t
        a = randn(n, n);
        b = randn(n, n);
        c = randn(n, n);
        d = randn(n, n);
        P = [a b c d];
        Q = [a b;c d];
        v(i) = trace((P.'*P)^4);
        w(i) = trace((Q.'*Q)^4);
    end
    s1 = std(v)/mean(v);
    s2 = std(w)/mean(w);
```

```
end

[s1, s2] = randmatstat(1000);
assert(round(10*s1) > 5 && round(10*s1) < 10);
timeit('rand_mat_stat', @randmatstat, 1000)
```

其对应的 Julia 代码如下：

```
function randmatstat(t)
    n = 5
    v = zeros(t)
    w = zeros(t)
    for i=1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P.'*P)^4)
        w[i] = trace((Q.'*Q)^4)
    end
    return (std(v)/mean(v), std(w)/mean(w))
end

(s1, s2) = randmatstat(1000)
@test 0.5 < s1 < 1.0 && 0.5 < s2 < 1.0
@timeit randmatstat(1000) "rand_mat_stat"
```

看起来几乎没有任何区别，除了函数的返回是用 `return` 以及有 `@test` 这样的 `@` 号开头的宏调用之外。根据其主页上的结果是 Julia 在这个例子上速度是 Matlab 的 4 倍。但是 Julia 并不是像 Octave 那样要做一个 Matlab 的开源克隆，所以仔细看的话，会发现 Julia 这个语言本身其实和 Matlab 脚本差别非常大的。

从它的定位来看，其主要竞争对手应该是 Matlab、R 和 Python。Matlab 和 R 有一个通病就是代码一般需要用极端向量化操作写出来才会很高效，这里有很多 trick 在里面，我曾经也写过一篇总结提高 Matlab 代码效率的 tricks 的博客，这样造成的后果是：如果可以进行向量化的运算和逻辑，最后会变得“面目全非”，当然这基本上是所有优化的通病；另一个更严重的问题就是有许多算法本身并不能表达成一些矩阵乘法之类的简单线性代数运算，Julia 主页上的 benchmark 列表里前面几项，性能差别成百上千倍的诸如斐波那契数列计算、快速排序之类的。

所以说 Julia 在这里的优势就是其高性能 JIT 使得复杂逻辑的代码在 Julia 里写出来跟直接用 C 写出来差不多。注意，这并不是说在 Julia 里就并不需要或者不鼓励用向量化的方法了。我们不妨来看一下下面的例子：

```
function dot_map(x,y)
    mapreduce(*, +, x, y)
end

function dot_loop(x,y)
    z = 0
    for i = 1:length(x)
        z += x[i]*y[i]
    end
    return z
end

function dot_vec(x,y)
    return sum(x.*y)
end

function dot_dot(x,y)
    return dot(x,y)
end

x = rand(10000)
y = rand(10000)

macro timeit(func)
    quote
        @printf("%10s ", $func)
        #force compile
        $func(x,y)

        @time for i = 1:200
            $func(x,y)
        end
    end
end

@timeit dot_map
@timeit dot_loop
@timeit dot_vec
@timeit dot_dot
```

在我的机器上的运行结果如下：

```
 dot_map elapsed time: 92.720649835 seconds (160160000000 bytes allocated)
dot_loop elapsed time: 0.084208763 seconds (64000000 bytes allocated)
 dot_vec elapsed time: 0.006839463 seconds (16152000 bytes allocated)
 dot_dot elapsed time: 0.000575898 seconds (3200 bytes allocated)
```

其中使用 `mapreduce` 的函数式编程表达的版本是最慢的，而且慢了 1000 倍啊……（如果我每算错的话）。原因是重复调用 `*` 函数，而编译器没有办法进行内联，所以导致很多很多次的函数调用，于是速度就慢下来了，这个问题从 C 语言时代的 `qsort` 函数就开始存在了。

在 Julia 中函数是 first-class 对象，可以传来传去的，但是本质上其实就是和 C 语言里面的函数指针一样的东西。C++ 里解决这个问题的办法是使用 functor，为什么 functor 可以内联而函数指针不行，是因为 functor 是一个有类型的静态的东西，在你使用某个 functor 的时候编译器就知道了对应的 `operator ()` 的函数体，但是函数指针不一样，指针的内容在编译期是完全未知的，必须在运行是动态解析，因此也就没有办法内联了。按理说 JIT 编译应该在这样的问题上会比较有优势一些，但是问题也还是并不是那么简单就可以解决的，比如如果你的代码在运行时把 `+` 重新定义成另一个函数了怎么办？

不过 Julia 里是有类型系统的[1]，还有根据类型 parametric 的函数，利用这个东西也可以做出像 C++ 里的 Functor 一样的东西，从而使得在类似的场合可以实现内联。Dahua Lin 专门做了一个叫做 NumericExtensions 的扩展来处理这个问题。

[1] 可以像 Haskell 那样做类型 annotation，并且也有强大的类型自动推演系统。

回到我们刚才的例子，手写 `for` 循环的实现居然比向量化的实现慢了十倍左右，不过对于后面 report 的内存使用总觉得有点诡异，因为我们的 `for` 循环实现除了结果一个数字之外并没有使用任何临时变量的样子，倒是向量化的实现 `dot_vec` 应该是有中间临时结果存在的样子。最后一个是直接调用内置的 `dot` 函数，这应该是 call 到 BLAS 的矩阵运算库里去了，BLAS 里的操作都是经过各种精心优化过的，所以速度是最快的。

简而言之，即使是在 Julia 里，如果比较明显能用向量化操作的地方，通常向量化还是有优势的，因为会使用 BLAS 里的高度优化过的矩阵运算代码，一般在矩阵维度比较大的时候优势比较能体现出来；反过来，即使是无法向量化的场合，直接在 Julia 里写复杂逻辑也不会像其他那些主流的 high-level 科学计算语言那么慢。有点略讽刺的是，有时候一些向量化的表达式会产生一些巨大的中间临时变量矩阵，反而影响性能，因此专门有一个 Devectorize 的扩展库来处理这样的事情……不过也许以后 Julia 的编译器能在临时变量的检测和消除方面做得更好一些的话这样的东西可能就不太需要了。

说起来 Python 虽然应该比 Matlab 和 R 要好一些，但是在这方面的表现似乎也不尽人意。在会被不断地调用很多很多次的那些计算用的代

码片段上，经常会被用 Cython 之类的工具来将那一段用 C 重写，或者是使用 numexpr 或者更神奇的 Numba 之类的包来做定点优化，JIT 方面 PyPy 似乎也一直在努力的样子。

不过不管怎么说 Python 在科学计算 high-level 语言方面是占据了非常重要的地位的，其中最重要的原因是它无数的各种各样的库所形成的生态圈。所以即使 Julia 拥有各种让人兴奋的新特性并且在性能上各种完败 Python，要取代 Python 也不是一朝一夕的事，或者甚至是永远也不会发生的事。实际上，Python 的生态圈的强大程度，只要看它自己从 Python 2 过渡到 Python 3 都如此地举步维艰就不难理解了。就好像是即使你有三条龙，也必须得寻找盟友才能统一七国……唔，最近好像《冰与火之歌》看太多了，突然就想起来 King Robert 对 Ned Stark 说的 “I have a son, you have a daughter”……在这件事情上两个社区似乎已经达成一致，所以最近看到越来越多的合作趋势，包括 Python 和 Julia 的互相调用日趋完善，IJulia 差不多算是一个标志性的 “Marriage” 吧，就是将 Julia 作为 IPython Notebook 的后端。

然后再回到 Julia 语言本身，它的发源地是 MIT，然后我最近发现那门口碑很好的 Parallel Computing 的课似乎就是用 Julia 来进行教学和实验的。它的有一些特性和另一个最近比较抢眼球的语言 Google Go 比较类似。一个方便的语言集成的基于 git 的扩展包管理系统，另外一个是异步、并行编程方面，Julia 里有内置的异步的 coroutine 以及多核并行调用支持。

另一个值得一提的特性是 multiple dispatch method，这个是在 Common Lisp 之类的语言里可以见到的特性。现在比较常见的 OO 语言使用的是根据 `this` 的不同可以具有多态性质，而 `this` 其实只是函数的第一个参数而已。multiple dispatch 则是更 general 的情况，不止看第一个参数，而是根据所有的参数的类型来共同决定调用 “同名的” 具体哪一个函数。这在定义诸如二元操作的时候更加 make sense 一些，例如 “一个矩阵加上一个数” 和 “一个数加上一个矩阵” 如果被考虑为 “数” 和 “矩阵” 各自的方法的话，总是有些奇怪的。

还有一个比较 handy 的特性是可以用诸如 `2x` 这样的方法来表达 `2*x`，这会使得数学表达式更简洁一些。而单从语言层面的话，最不得不提的应该就是强大的宏功能了。和 Lisp 一样，Julia 的代码本身是作为 Julia 里可操作的数据结构存在的，当然不是像 Lisp 那样的 list 结构，而是由 `Expr` 类型构成的一棵树。因此在宏里就可以做几乎任意的代码修改、转换、构造等等各种操作了。

当然了，作为一门实用的语言，除了语言本身的特性之外，周边的各种特性也是非常重要的。其中一个很重要的考察因素就是社区以及第三方库。这方面 Julia 还处于发展初期。但是在网上可以看到不少关于 Julia 的博客，大家纷纷表示对这个新事物非常 exciting，并且未来看起来非常 promising，所以如果你有想学习一下这门语言的话，动手来做一个你熟悉领域的扩展库应该是不错的选择。这里有一个目前已经有的库的列

表。当然就我自己目前有的一点点经验来看，Julia 的社区也好，包括语言本身也好，也还在不断的发展中[2]，有点麻烦的一点是现在资料也比较少，如果官方给的 manual 上觉得没有写清楚的话，能找的其他资料基本上就没有了。

[2] 并不是指语法等方面会有巨大变动，而是说在性能和一些核心标准库方面应该会不断改良吧。比如也许可能会出现将编译好的 code 在磁盘里缓存起来之类的功能。

最后就是 IDE 方面的功能。IDE 倒是并不一定要有，但是作为科学计算语言，方便的画图功能是必须要有的。目前已经有一些第三方扩展包在为 Julia 添加这方面的功能了，虽然由此引入 cairo 以及随之而来的各种依赖的包……至于调试和 profiling，（就我所知的范围内）在科学计算方面目前大家都比 matlab 差了一大截的样子。

然后偷偷吐槽一下，为什么似乎我尝试过的所有基于 JIT 的号称各种性能很好的语言启动速度都慢得不行呢？还是只是我的幻觉吗？`:D`

update: 刚好看到一个 SciPy 2013 的关于 Julia 和 Python 的 Talk 视频，可以翻墙的同学请点击。