Computação Paralela e Distribuída

Heucles Jr

9395391

Agenda

- Breve Histórico
- Sintaxe
- Just-in-Time Compiler
- Tipos Numéricos
- Macros
- Paralelismo
  - Tasks
  - Workers
  - Messaging
  - Parallel loops and maps
  - Distributed arrays

Um pouco de História...

- Criada em 2012
- Principais criadores:
  - Jeff Bezanson
  - Stefan Karpinski
  - Viral B. Shah
  - Alan Edelman
- Motivação: Criar uma linguagem que atenda os seguintes requisitos
  - Seja open source
  - velocidade de C
  - dinamismo de Ruby
  - Tenha macros como Lisp
  - Com uma notação matemática óbvia, assim como Matlab.
  - Útil para a programação geral, como Python
  - Fácil para as estatísticas como R
  - E tão natural para o processamento de cadeia como Perl
  - Que seja interativa e compilada.
  - Que seja dinâmica
  - Que seja funcional
  - Que suporte tipos quando necessitar-se de funções polimórifcas
  - Uma linguagem limpa

- Atual versão estável: 0.3.9

Sintaxe

Rapidamente os desenvolvedores se sentem familiarizados com a sintaxe de Julia.

- O uso de for, while e if é muito próximo ao de Ruby ou Python.
- continue, break e return também funcionam conforme o esperado.
- Definir uma função também é simples, o nome da função é seguido pelos seus argumentos.
- Você pode especificar os tipos dos argumentos, mas isso é na verdade opcional.
- @inbounds é um tipo de macro e macros sempre começam com o caractere @.

```
function sort!(v::AbstractVector, lo::Int, hi::Int, ::InsertionSortAlg, o::Ordering)
    @inbounds for i in lo+1:hi
        j = i
        x = v[i]
        while j > lo
            if lt(o, x, v[j-1])
                v[j] = v[j-1]
                j -= 1
                continue
            end
            break
        end
        v[j] = x
    end
    return v
end
```

base/sort

Sintaxe

- n: m cria uma “range” de dados que e é inclusivo em ambos os lados.
- • [... for x in xs] cria uma matriz de xs, que é algo iterável.
- Esta notação é conhecida “list comprehension” em Python e Haskell.

```
4:8                     #> 4:8
[x for x in 4:8]        #> [4,5,6,7,8]
[4:8]                   #> [4,5,6,7,8]
[x * 2 for x in 4:8]    #> [8,10,12,14,16]
```

- O índice de um array sempre começa sempre com 1, não 0.
- O Isso significa que quando um array com tamanho n, todos os índices em 1: n são acessíveis.
- Você pode usar uma range de dados para copiar uma parte de um array.
  - O incremento em uma range pode ser colocado entre o início e o fim (i.e. start: step: stop)
  - Poder-se especificar um incremento negativo, o que cria uma range inversa.
  - Existe um índice especial(palavra reservada) - end - indicando o último índice de um array.

```
xs = [8, 6, 4, 2, 0]
xs[1:3]         #> [8,6,4]
xs[4:end]       #> [2,0]
xs[1:2:end]     #> [8,4,0]
xs[end:-2:1]    #> [0,4,8]
```

Tipos numéricos

- Julia possui vários tipos numéricos com diferentes tamanhos

Integer types

Type	Signed?	Number of bits	Smallest value	Largest value
`Int8`	✓	8	-2^7	2^7 - 1
`Uint8`		8	0	2^8 - 1
`Int16`	✓	16	-2^15	2^15 - 1
`Uint16`		16	0	2^16 - 1
`Int32`	✓	32	-2^31	2^31 - 1
`Uint32`		32	0	2^32 - 1
`Int64`	✓	64	-2^63	2^63 - 1
`Uint64`		64	0	2^64 - 1
`Int128`	✓	128	-2^127	2^127 - 1
`Uint128`		128	0	2^128 - 1
`Bool`	N/A	8	`false` (0)	`true` (1)
`Char`	N/A	32	`'\0'`	`'\Uffffffff'`

Floating-point types

Type	Precision	Number of bits
`Float16`	half	16
`Float32`	single	32
`Float64`	double	64

Just-in-Time Compiler

- Para executar o seu programa escrito em Julia, não há necessidade de compilá-lo de antemão.
- Basta que se que dar o arquivo de ponto de entrada para o JIT:

```
% cat myprogram.jl
n = 10
xs = [1:n]
println("the total between 1 and $n is $(sum(xs))")
% julia myprogram.jl
the total between 1 and 10 is 55
```

- A partir da versão 0.3 as bibliotecas padrão ficam precompiladas, o que economiza muito tempo de inicialização do programa.

```
% time julia myprogram.jl
the total between 1 and 10 is 55
        0.80 real         0.43 user         0.10 sys
```

Macros – Metaprograming in Julia

- Macros permitem que se modifique o código antes de o compilador processar o mesmo.
- No exemplo a seguir, a macro assert recebe a expressão (x> 0), em seguida, avalia a expressão para o contexto em questão. Quando o resultado avaliado é falso, ele lança um erro de declaração. Observe que a mensagem de erro contém expressão recebida (x> 0), que tem em seu resultado false; esta informação é útil para fins de depuração.

```
x = -5
@assert x > 0  #! ERROR: assertion failed: x > 0
```

- Ao invés da expressão pode-se expecificar a mensagem de retorno

```
x = -5
@assert x > 0 "x must be positive"  #! ERROR: assertion failed: x must be positive
```

Paralelismo

- Em um mundo de CPU “multicore” e computação em cluster, é indispensável para qualquer nova linguagem que tenha excelentes capacidades para computação paralela.
- Julia faz disso um de seus pontos fortes por meio de:
  - Tasks
  - Workers
  - Messaging
  - Parallel loops and maps
  - Distributed arrays

Tasks

- Julia tem um sistema nativo próprio para rodar tarefas(tasks).
- Em uma task um processo computacional que gera um valor (utilizando-se da função produce).
- Por meio da função produce a task é suspensa, enquanto outro processo, no caso um cliente desta tarefa consome seus valores conforme utilizando-se da função consume. Esta estrutura é considerada similar ao que se faz com o yield em python.
- Por exemplo temos abaixo uma função que calcula os primeiros n números da sequência de Fibonacci, mas neste caso a função em questão não retorna os números, ela os produz.
- Co-rotinas não são executadas em threads diferentes, sendo assim não podem ser executadas em diferentes CPUs. Somente uma co-rotina é executada por vez. Um scheduler interno controla uma fila de tarefas executáveis e alterna entre elas baseado em eventos, tais como aguardando dados ou dados entrando.

```
function fib_producer(n)
    a, b = (0, 1)
    for i = 1:n
        produce(b)
        a, b = (b, a+b)
    end
end
consume(tsk1) #=> 1
consume(tsk1) #=> 1
consume(tsk1) #=> 2
consume(tsk1) #=> 3
consume(tsk1) #=> 5
consume(tsk1) #=> 8
consume(tsk1) #=> 13
consume(tsk1) #=> 21
consume(tsk1) #=> 34
consume(tsk1) #=> 55
consume(tsk1) #=> nothing # Task (done) @0x0000000005696180
```

Workers

- Worker: processo criado para a realização de tarefas em paralelo.
- Julia pode ser inicializado no REPL ou em uma aplicação separada com um número de "workers" n disponíveis.

```
julia -p n #starts REPL with n workers
```

- Estes "workers" são processos diferentes, não "threads", logo eles não compartilham memória.
- Uma dica para que realmente se possa obter o melhor de uma máquina é que se crie a quantidade de processos com o mesmo número de "cores" de processamento que se possui na maquina.
- Portanto para a sintaxe acima se n = 7 tem-se 8 "workers", 1 para o REPL shell e outros 7 para a realização de tarefas paralelas. Os ids são inteiros para os "workers" podem ser obtidos a partir da função "workers()".

```
workers()
7-element Array{Int64,1}:
 2
 3
 4
 5
 6
 7
 8
```

- Se em algum momento for necessário adicionar um novo "worker" a função addprocs(n), onde n representa a quantidade que se quer adicionar.
- Dentro de seu próprio contexto cada worker pode recuperar seu próprio id utilizando a função myid().
- Um worker pode ser removido por meio da função rmprocs(id) e nprocs() retorna a quantidade de "workers" disponíveis.
- Os "workers" podem rodar todos na mesma máquina, assim como se comunicar entre em eles via portas TCP. Para que ative os "workers" em um cluster de vários computadores.

Messaging

- O modelo nativo de computação paralela em Julia baseia-se em 2 conceitos chave: "remote calls" e "remote references".
- Desta forma pode-se solicitar a um "worker" que execute uma função com argumentos através da função remotecall, e obter seu resultado de volta com fetch.
- Por exemplo, se se quiser que o worker 2 eleve o valor de 1000 ao quadrado:

```
r1 = remotecall(2, x->x^2,1000) # > RemoteRef(2,1,9)
```

- E então para obter o valor de r1 basta que se execute uma chamada para função fetch:

```
fetch(r1)
1000000
```

- A chamada para a função fetch irá bloquear o processo principal até que o "worker" 2 tenha terminado o processamento.
- Uma dica é que se use a função remotecall_fetch que é mais eficiente do que fetch(remotecall(..)).

```
remotecall_fetch(2,x->x^2,1000)
1000000
```

- Pode-se utilizar também a macro @spawnat que recebe como argumentos o id do "worker" assim como a função a e ser executada
- Ou a macro @spawn que so precisa receber a função, e escolhe o worker baseado em uma lógica interna de otimização.

```
r2 = @spawnat 4 sqrt(16)
fetch(r2)
4

r3 = @spawn sqrt(5)
RemoteRef(5,1,26)
fetch(r3)
2.23606797749979
```

- Temos também a macro @everywhere que executa uma função em todos os "workers":

```
@everywhere function fib(n) #definindo a função em todos os workers
if (n < 2) then
        return  n
else return fib(n-1) + fib(n-2)
        end
end
@everywhere println(fib(myid()))
1
        From worker 2:  1
        From worker 6:  8
        From worker 4:  3
        From worker 5:  5
        From worker 7:  13
        From worker 8:  21
        From worker 3:  2
```

Parallel loops and maps

- Caso tenha-se um laço com um número muito grande de iterações, Julia provê um ótimo recurso que permite que as iterações deste laço sejam paralelizadas, por meio da macro @parallel.
- Para calcular uma aproximação para Π usando o famoso problema da agulha de Buffon.

```
function buffon(n)
        hit = 0
        for i = 1:n
                mp = rand()
                phi = (rand() * pi) - pi / 2 # angle at which needle falls
                xright = mp + cos(phi)/2 # x location of needle
                xleft = mp - cos(phi)/2
                # does needle cross either x == 0 or x == 1?
                p = (xright >= 1 || xleft <= 0) ? 1 : 0
                hit += p
        end
        miss = n - hit
        piapprox = n / hit * 2
end
```

- Com o valor n aumentando constantemente a fim de que se possa obter um valor mais preciso para Π, o tempo necessário para o cálculo de buffon aumentara de forma linear.
- Porém com algumas sutis mudanças no código pode-se aproveitar do paralelismo oferecido pela macro @parallel que permitira a distribuição do calculo entre todos os "workers" disponíveis. É importante que se tenha em mente que para isso é necessário que já tenham sido adicionado estes "workers" ao processo principal.

Parallel loops and maps

- A macro divide o intervalo de dados, e o distribui para cada um dos “workers” disponíveis.
- Ela opcionalmente pode receber um "redutor" como seu primeiro argumento. Se for especificado um redutor, os resultados de cada processamento serão agregados usando o redutor. No exemplo a seguir, usamos a função de (+) como um redutor, o que significa que os últimos valores dos blocos paralelos em cada "worker" serão somados para calcular o valor final de "button_par":

```
function    buffon_par(n)
        hit =   @parallel   (+) for i   =   1:n
                        mp  =   rand()
                        phi =   (rand() *   pi) -   pi  /   2
                        xright  =   mp  +   cos(phi)/2
                        xleft   =   mp  -   cos(phi)/2
                            (xright >=  1   ||  xleft   <=  0)  ?   1   :   0
                end
        miss    =   n   -   hit
        piapprox    =   n   /   hit *   2
end
```

```
##Comparação de tempos:

@time buffon(100000) #elapsed time: 0.018817472 seconds (96 bytes allocated)

@time buffon_par(100000) #elapsed time: 0.024267599 seconds (447408 bytes allocated)

@time buffon(100000000) #elapsed time: 5.945724033 seconds (96 bytes allocated)

@time buffon_par(100000000) #elapsed time: 1.467986884 seconds (457036 bytes allocated)
```

- Pode-se observar um desempenho muito melhor para o maior número de iterações (um fator de aprox. 4,06 neste caso). Ao alterar para uma versão em redução paralela, foi-se capaz de se obter melhorias substanciais no tempo de cálculo, em contrapartida o custo de memória foi também consideravelmente maior.
- Em geral, é sempre válido testar se a versão paralela é realmente uma melhoria em relação a versão sequencial para cada caso **específico**!

Parallel loops and maps

- Se a tarefa computacional consiste em aplicar uma função a todos os elementos em alguma coleção, essa operação poderá ser realizada através da função pmap.
- A função pmap tem a seguinte definição: pmap (f, coll), aplica-se uma função f em cada elemento da coleção coll de forma paralela, porém é importante ressaltar que ela preserva a ordem da coleção em seu resultado.
- Suponha que se tenha que estabelecer um ranking a classificação entre grandes arrays.

```
function rank_marray()
        marr = [rand(1000,1000) for i=1:10]
        for arr in  marr
                println(rank(arr))
        end
end

function prank_marray()
        marr = [rand(1000,1000) for i=1:10]
        println(pmap(rank, marr))
end

#tempos obtidos no livro com 8 workers criados
@time rank_marray()
elapsed time: 4.351479797 seconds (166177728 bytes allocated, 1.43% gc time)

@time prank_marray()
elapsed time: 2.785466798 seconds (163955848 bytes allocated, 1.96% gc time)

#tempos obtidos em minha máquina com 7 workers criados, não consegui reproduzir o ganho mostrado na literatura.
@time rank_marray()
elapsed time: 4.991476819 seconds (165955184 bytes allocated, 0.99% gc time)

@time prank_marray()
{1000,1000,1000,1000,1000,1000,1000,1000,1000,1000}
elapsed time: 5.949807812 seconds (163959860 bytes allocated, 0.89% gc time)

#tempos obtidos em minha máquina com 8 workers criados
@time prank_marray()
{1000,1000,1000,1000,1000,1000,1000,1000,1000,1000}
elapsed time: 6.14687637 seconds (163960956 bytes allocated, 0.87% gc time)
```

Distributed Arrays

- Quando os cálculos tem que ser feitos sob uma grande coleção de dados, esta coleção pode ser distribuída de modo que cada “worker” processe em paralelo uma porção diferente da coleção.
- Desta forma, pode-se fazer uso dos recursos de memória de várias máquinas, e permitir operações com matrizes que seriam grandes demais para processar em uma única máquina.
- O tipo de dado usado para isso é chamado DArray.
- Para DArray a maioria das operações se comportam exatamente como do tipo Array convencional, de modo que o paralelismo é transparente.
- Com DArray, cada processo tem acesso local a apenas uma parte dos dados, e não há dois processos compartilham a mesma porção dos dados.
- Por exemplo, o seguinte código cria uma matriz de distribuição de números aleatórios com dimensões de 100 x 100 e está dividido em quatro "workers". A divisão de dados é determinada pelo terceiro argumento e divide o número de colunas uniformemente ao longo de 4 "workers":

```
arr = drand((100,100), workers()[1:4], [1,4])
100x100 DArray{Float64,2,Array{Float64,2}}:
0.871469   0.7997     0.00888888  0.278222   …  0.897995  0.735878  0.159563
0.747167   0.0268745  0.854976    0.427925      0.813278  0.706306  0.505296
#...
```

- DArrays também podem ser criados com a macro @parallel:

```
da=@parallel[2i for i=1:10]
#10-element DArray{Int64,1,Array{Int64,1}}
```

Distributed Arrays

- O seguinte trecho de código é muitas vezes usado para construir um Array distribuído dividido sobre o "workers" disponíveis:

```
DArray((10,10)) do I
    println(I)
    return rand(length(I[1]), length(I[2]))
end

    From worker 5:  (1:10,5 From worker 7:  (1:10,8:9)
    From worker 3:  (1:10,2:3)
    From worker 2:  (1:10,1:1)
    From worker 4:  (1:10,4:4)
    From worke  From worker 8:  (1:10,10:10
Out [144]: 10x10 DArray{Float64,2,Array{Float64,2}}:
 0.0877044  0.911382    0.541605   …  0.585515   0.899899  0.0352855
 0.704492   0.619204    0.0365494     0.188961   0.306272  0.643887
 0.853896   0.520737    0.10297       0.664151   0.178997  0.488951
 0.775867   0.956848    0.552983      0.143999   0.303394  0.348659
 0.725631   0.2602      0.289017      0.789605   0.936989  0.444051
 0.415167   0.180224    0.816734   …  0.706728   0.252826  0.208654
 0.812981   0.335355    0.24065       0.475395   0.071712  0.749807
 0.612279   0.899194    0.0100128     0.329977   0.654011  0.244893
 0.476686   0.00839265  0.288618      0.0222408  0.695946  0.577096
 0.435181   0.526262    0.021563      0.429026   0.605228  0.370958
```

Referências

- *Balbert, Ivo et al.* ***Getting Started with Julia Programming****. 1.ed. Birmingham: Packt Publishing, 2015. 356 p.*
- https://en.wikibooks.org/wiki/Introducing_Julia/Modules_and_packages
- http://julia.readthedocs.org/en/latest/manual/
- http://julialang.org/blog/2012/02/why-we-created-julia/
- https://github.com/JuliaParallel/DistributedArrays.jl
- http://www.admin-magazine.com/HPC/Articles/Julia-Distributed-Arrays
- http://bogumilkaminski.pl/les/julia express.pdf
- https://en.wikipedia.org/wiki/Buffon's_needle