MAC 5742 - Computação Paralela e Distribuída
Introdução a Julia Lang e suas Características de Paralelismo

Heucles Del Bianco Pelegia Jr.
heucles@gmail.com
9395391

Contents

1 Histórico

O desenvolvimento de Julia começou em 2009 e uma versão de código aberto foi divulgado em fevereiro de 2012. Seus criadores Jeff Bezanson, Stefan Karpinski, Viral Shah e Alan Edelman alegam que o desenvolvimento de Julia veio da necessidade de uma linguagem que agrupasse o melhor uma série de linguagens excelentes para alguns aspectos da computação científica, mas não tão boas para outros aspectos, tais ambições resumem-se em:

- Seja open source
- Tenha a velocidade de C
- O 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
- Seja 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
- Seja uma linguagem limpa

Os criadores brincam em sua página sobre como são gananciosos, e que é essa a maior motivação para criação de Julia, sua versão atual estável sendo a 0.3.9 e com a versão 0.4.0 ainda em desenvolvimento e recebendo atualizações diárias.

2 Sobre Julia

Julia pode ser facilmente instalada nas principais plataformas de sistemas operacionais (principais distribuições de Linux, MACOS e Windows), e a documentação é bem clara com relação a sua instalação.

2.1 REPL

Uma das mais importantes funcionalidade em Julia é o fato de inclui um shell de sessão interativa, chamado de REPL (Read Evaluate Print Loop) de Julia, que pode ser usado para experimentar e testar o código rapidamente. O seguinte fragmento representa uma sessão de amostra no REPL.:

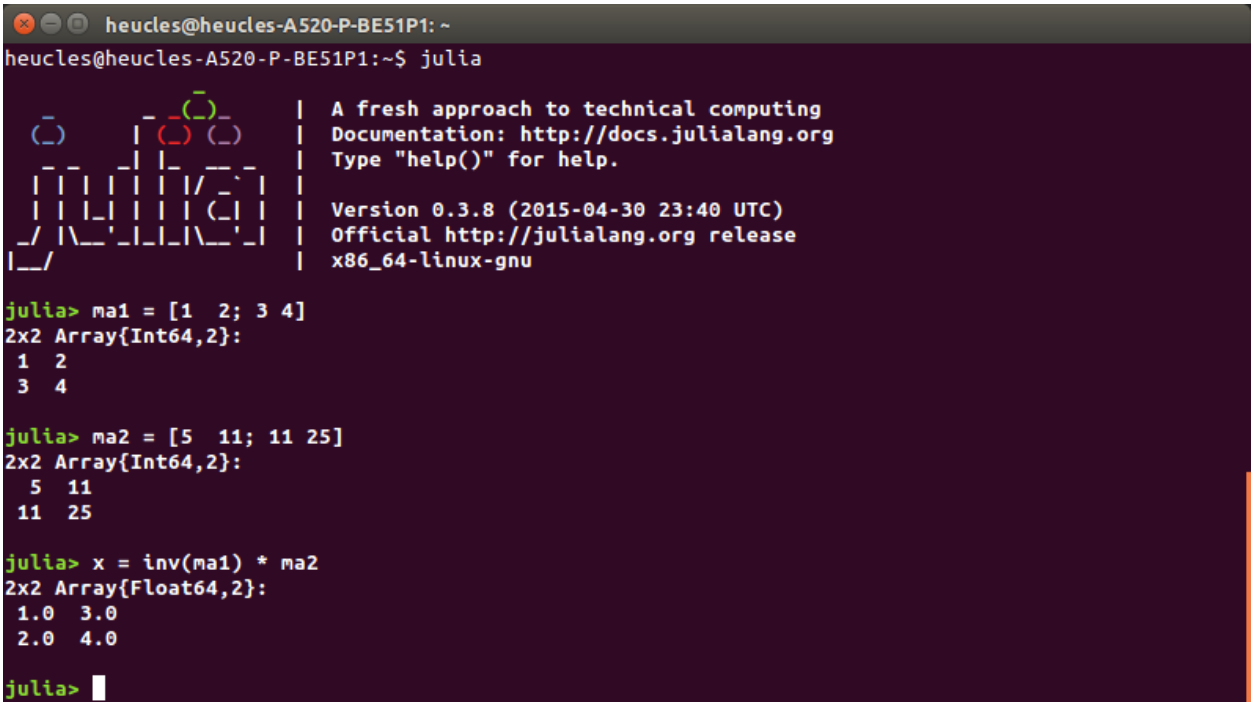

Figura 1: Exemplo de código executado no REPL.

Alguns comandos importantes para utilizar no REPL:

1. `^D` (exits julia);
2. `^C` (interrupts computations);
3. `?` (enters help mode)
4. `putting ;` after the expression will disable showing of its value.

Além de as seguintes expressões que são fundamentais no lido com a linguagem:

```
help(help) # get help on function help
apropos("help") # search documentation for help
@less(max(1,2)) # show the definition of max function when invoked with arguments 1 and 2
whos() # list of global variables and their types
cd("D:/") # change working directory to D:/ (on Windows)
pwd() # get current working directory
include("file.jl") # execute source file
require("file.jl") # execute source file if it was not executed before
exit(1) # exit with code 1 (exit code 0 by default)
clipboard([1:10]) # copy data to system clipboard
workspace() # clear worskspace - create new Main module (only to be used interactively)
```

2.2 IDEs

Existem várias IDEs que suportam Julia ou foram feitas para trabalhar diretamente com ela, entre elas estão:

- Julia Studio
- IJulia
- Juno
- Sublime - através do pacote Sublime-IJulia

Para o desenvolvimento deste trabalho foi escolhido o Sublime, que utilizando seu serviço de pacotes torna muito fácil a instalação do pacote para Julia, fica abaixo um exemplo de como fica a IDE:

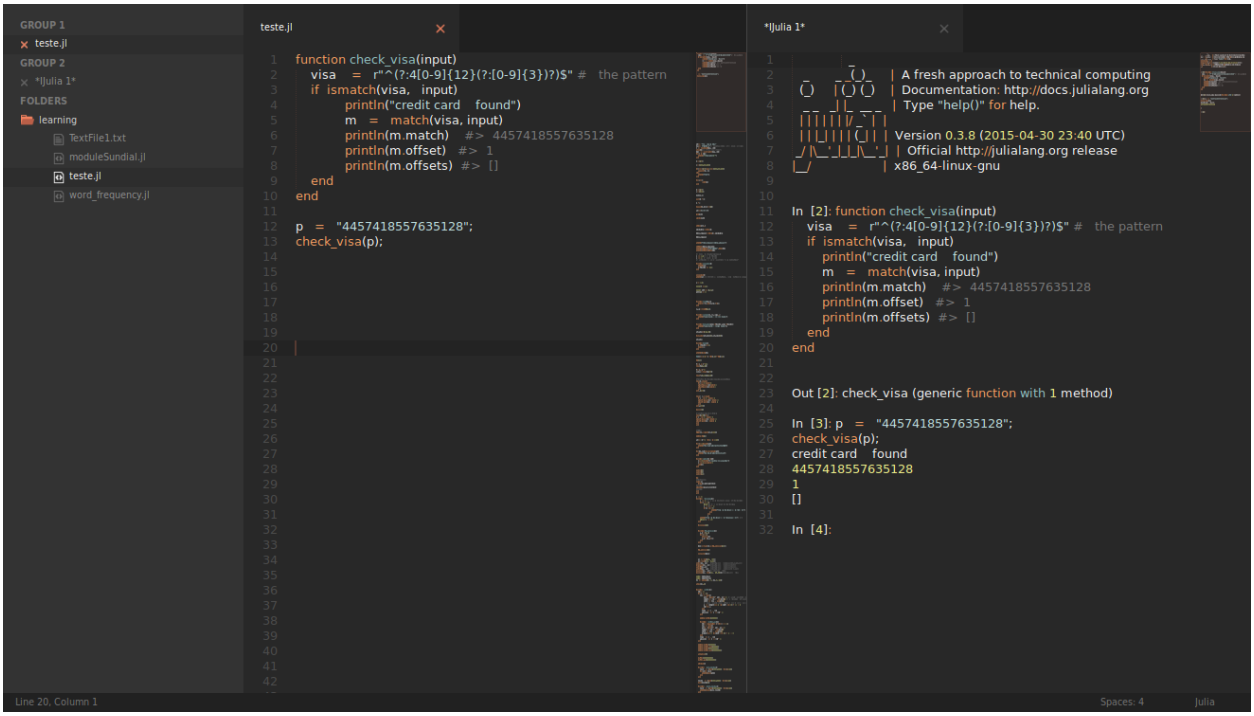

Figura 2: Exemplo de código executado no Sublime-IJulia.

2.3 Just-in-Time Compiler

Julia trabalha com um compilador JIT(Just-in-Time) LLVM (Low-level-Virtual-Machine) que é usado para a geração de código de máquina. A primeira vez que uma função é executada em Julia, ela é analisada e os tipos são inferidos. Em seguida, o código LLVM é gerado pelo compilador JIT (just-in-time), e em seguida, é otimizada e compilada para o código baixo nível nativo. A segunda vez que a mesma função for chamada em Julia, o código nativo gerado já é chamado. Esta é a razão pela qual, a segunda vez que se chamar uma função com argumentos de um tipo específico, que leva muito menos tempo para executar do que a primeira vez (manter isso em mente ao fazer benchmarks de código Julia). Este código gerado pode ser verificado. Supondo que tenha sida definida uma função `f(x) = 2x + 5` em uma sessão do REPL, Julia responderá com a mensagem "f (generic function with 1 method)" o que significa que o código é dinâmico uma vez que não tem como expecificar o tipo de x ou o tipo de retorno de f. Todas as funções, variáveis e argumentos são por padrão genéricas, a menos que se especifique um tipo. Abaixo pode-se observar a inspeção LLVM do código para algumas das possíveis variações de f:

```
julia> f(x) = 2x + 5
f (generic function with 1 method)

# Codigo onde caso o argumento x seja do tipo Int64:
julia> code_llvm(f, (Int64,))

define i64 @julia_f_20143(i64) {
top:
  %1 = shl i64 %0, 1, !dbg !1274
  %2 = add i64 %1, 5, !dbg !1274
  ret i64 %2, !dbg !1274
}

# Codigo nativo caso o argumento x seja do tipo Int64:
julia> code_native(f, (Int64,))
    .text
Filename: none
Source line: 1
    push  RBP
    mov RBP, RSP
Source line: 1
    lea RAX, QWORD PTR [RDI + RDI + 5]
    pop RBP
    ret

# Codigo nativo caso o argumento x seja do tipo Float64
julia> code_native(f, (Float64,))
    .text
Filename: none
Source line: 1
    push  RBP
    mov RBP, RSP
Source line: 1
    addsd XMM0, XMM0
    movabs RAX, 140113497473008
    addsd XMM0, QWORD PTR [RAX]
    pop RBP
    ret
```

Código Julia é rápido porque gera versões especializadas de funções para cada tipo de dados. Julia implementa gerenciamento automático de memória. A exclusão automática de objetos que não são mais necessários é feito usando um GC ("Garbage Colector"). O GC é executado ao mesmo tempo que o programa. Na Versão 0.3, o GC é um simples coletor de lixo "mark-and-sweep", a versão 0.4 trará uma uma versão "mark-and-sweep" incremental. O GC pode ser inicializado manualmente pelo comando `gc()`, ou desabilitado utilizando `gc_disable()`.

3 Tipos

Sistema de tipos de Julia é dinâmico, mas ganha algumas das vantagens dos sistemas de tipo fortemente tipados, tornando possível para indicar que certos valores são de tipos específicos. Isto pode ser de grande ajuda na geração de código eficiente, mas ainda mais significativamente, ele permite que o recurso de "Multiple Dispatch" sobre os tipos de argumentos da função sejam profundamente integrados com a língua.

Todos os tipos básicos são imutáveis. Especificar o tipo é opcional (e não é necessário a menos que se queira restringir o código ou melhorar a performance). Se o tipo não for especificado Julia vai escolher um padrão. Há diferenças em máquinas 32-bit e 64 bits. A diferença mais importante é para inteiros que são Int32 e Int64 respectivamente. Isso significa a "assertion" 1 :: Int32 (1 é do tipo Int32) irá falhar em uma máquina de 64 bits. Não há nenhuma conversão automática de tipo (especialmente importante em chamadas de função). Esta conversão tem que ser explícita:

```
int64(1.3) # rounds float to integer
int64( a ) # character to integer
int64("a") # error no conversion possible
int64(2.0^300) # error - loss of precision
float64(1) # integer to float
bool(-1) # converts to boolean true
bool(0) # converts to boolean false
char(89.7) # cast float to integer to char
string(true) # cast bool to string (works with other types)
```

Tipos literais escalares básicos (x :: Type é uma x expressão literal com tipo Type de assertion):

```
1::Int64 # 64-bit integer, no overflow warnings, fails on 32 bit Julia, use Int32 assertion instead
1.0::Float64 # 64-bit float, defines NaN, -Inf, Inf
true::Bool # boolean, allows "true" and "false"
  c  ::Char # character, allows Unicode
"s"::String # strings, allows Unicode, see also Strings
```

Conversão genérica pode ser feita usando convert (Type, x):

```
convert(Int64, 1.0) # convert float to integer
```

Promoção automática de tipos argumentos para tipos comuns à eles (se houver) pode ser feita utilizando promote:

```
promote(true, c, 1.0) # tuple (see Tuples) of floats, true promoted to 1.0
```

Muitas operações (aritméticas, de atribuição) são definidas de uma forma que executa a promoção de tipo automático. Pode-se verificar então o tipo de argumento:

```
One can verify type of argument:
typeof("abc") # ASCIIString returned which is a String subtype
isa(1, Float64) # false, integer is not float
isa(1.0, Float64) # true
```

É possível executar cálculos usando aritmética de precisão arbitrária ou números racionais:

```
BigInt(10)^1000 # big integer
BigFloat(10)^1000 # big float, see documentation how to change default precision
123//456 # rational numbers using // operator
```

Exemplo de tipo composto definido pelo usuário:

```
type Point
x::Int64
y::Float64
meta
end
```

```
p = Point(0, 0.0, "Origin")
p.x # access field
p.meta = 2 # change field value
p.x = 1.5 # error, wrong data type
p.z = 1 # error - no such field
names(p) # get names of instance fields
names(Point) # get names of type field
```

4 Construindo um programa

A forma mais simples de criar uma variável é associando um valor a ela:

```
x = 1.0 # x is Float64
x = 1 # now x is Int32 on 32 bit machine and Int64 on 64 bit machine
y::Float64 = 1.0 # y must be Float64, not possible in global scope
# performs assertion on y type when it exists
```

Expressões podem ser compostas usando `;` ou blocos `begin end`:

```
x = (a = 1; 2 * a) # after: x = 2; a = 1
y = begin
    b = 3
    3 * b
end # after: y = 9; b = 3
```

Alguns exemplos padrão de programação:

```
if false # if clause requires Bool test
    z = 1
elseif 1==2
    z = 2
else
    a = 3
end # after this a = 3 and z is undefined
1==2 ? "A" : "B" # standard ternary operator
i = 1
while true
    i += 1
    if i > 10
        break
    end
end
for x in 1:10 # x in collection
    if 3 < x < 6
continue # skip one iteration
end
println(x)
end # x is introduced in loop outer scope
```

E o usuário pode ainda definir suas próprias funções:

```
f(x, y = 10) = x + y # new function f with y defaulting to 10
# last result returned
f(3, 2) # simple call, 5 returned
f(3) # 13 returned
function g(x::Int, y::Int) # type restriction
return y, x # explicit return of a tuple
end
apply(g, 3, 4) # call with apply
apply(g, 3, 4.0) # error - wrong argument
g(x::Int, y::Bool) = x * y # add multiple dispatch
g(2, true) # second definition is invoked
methods(g) # list all methods defined for g
(x -> x^2)(3) # anonymous function with a call
() -> 0 # anonymous function with no arguments
h(x...) = sum(x)/length(x) - mean(x) # vararg function; x is a tuple
h(1, 2, 3) # result is 0
x = (2, 3) # tuple
f(x) # error
```

```
f(x...) # OK - tuple unpacking
s(x; a = 1, b = 1) = x * a / b # function with keyword arguments a and b
s(3, b = 2) # call with keyword argument
t(; x::Int64 = 2) = x # single keyword argument
The Julia Express 8
t() # 2 returned
t(; x::Bool = true) = x # no multiple dispatch for keyword arguments; function overwritten
t() # true; old function was overwritten
q(f::Function, x) = 2 * f(x) # simple function wrapper
q(x -> 2 * x, 10) # 40 returned
q(10) do x # creation of anonymous function by do construct, useful in IO
2 * x
end
m = reshape(1:12, 3, 4)
map(x -> x ^ 2, m) # 3x4 array returned with transformed data
filter(x -> bits(x)[end] == 0, 1:12) # a fancy way to choose even integers from the range
```

Como convenção funções que tem seu nome iniciado por ! modificam o valor dos argumentos que lhes foram passados por referencia. Como por exemplo a função resize.

Exemplos de criação de funções padrão:

```
y = 10
f1(x=y) = x; f1() # 10
f2(x=y,y=1) = x; f2() # 10
f3(y=1,x=y) = x; f3() # 1
f4(;x=y) = x; f4() # 10
f5(;x=y,y=1) = x; f5() # error - y not defined yet :(
f6(;y=1,x=y) = x; f6() # 1
```

5 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.

Exemplo da macro `assert`:

```
x = -5
@assert x > 0 # will return 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
```

Mais alguns exemplos de macros úteis:

- Assertions:

```
@assert 1 == 2 "ERROR" # 2 macro arguments; error raised
using Base.Test # load Base.Test module
@test 1 == 2 # similar to assert; error
@test_approx_eq 1 1.1 # error
@test_approx_eq_eps 1 1.1 0.2 # no error
```

- Function vectorization:

```
t(x::Float64, y::Float64 = 1.0) = x * y
t(1.0, 2.0) # OK
t([1.0 2.0]) # error
@vectorize_1arg Float64 t # vectorize first argument
t([1.0 2.0]) # OK
t([1.0 2.0], 2.0) # error
@vectorize_2arg Float64 t # vectorize two arguments
t([1.0 2.0], 2.0) # OK
t(2.0, [1.0 2.0]) # OK
t([1.0 2.0], [1.0 2.0]) # OK
```

- Benchmarking:

```
@time [x for x in 1:10^6]. # print time and memory
@timed [x for x in 1:10^6]. # return value, time and memory
@elapsed [x for x in 1:10^6] # return time
@allocated [x for x in 1:10^6] # return memory
tic() # start timer
The Julia Express 13
[x for x in 1:10^6].
toc() # stop timer and print time
toq() # stop timer and return time
```

6 Outros aspectos de Julia, um pouco fora de contexto, mas que valem ser mencionados

- **Outros tipos:** Julia possui suporte a diversos outros tipos mais complexos do que os mensionados no item 3, estes, assim como os tipos de composição, são passados como referência para funções, sendo assim tem que se manter em mente os conceitos de ”shallow-copy” e ”deep-copy”. Muitos destes tipos

são relacionados a coleções de dados, cada uma rica em uma série de "features" diferentes para vários diversos cenários, segue abaixo uma lista destes tipos:

  - Tuples
  - Arrays
  - Matrix
  - Composite types
  - Dictionaries
  - Data frames

- **Pacotes:** Muitas das excelentes funcionalidades vem da grande diversidade de pacotes disponíveis. Todos podem criar e publicar pacotes para Julia. Estes pacotes precisam ser instalados e referenciados para que possam ser utilizados, Julia possui um repositório para estes pacotes e o processo de instalação é bem simples.

- **Módulos:** Módulos encapsulam código, tem uma funcionalidade similar a namespaces em outras linguagens, módulos podem ser recarregados, isso é muito útil para redefinir funções e tipos, pacotes terão ao menos um módulo, o qual deverá ser declarado após a instalação de um pacote, para que as funções e tipos do pacote possam ser utilizados.

- **Multiple Dispatch:** Trata-se de uma característica de algumas linguagens de programação na qual uma função ou método podem ser dinamicamente sobrecarregados e/ou sobrescritos em tempo de execução, o mais interessante é que se pode fazer isso com funções de módulos nativos base do Julia, ou mesmo funções de módulos importados de pacotes externos. Alguns exemplos de "Multiple Dispatch":

```
f(n, m) = "base case"
f(n::Number, m::Number) = "n and m are both numbers"
f(n::Number, m) = "n is a number"
f(n, m::Number) = "m is a number"
f(n::Integer, m::Integer) = "n and m are both integers"
```

- **Integração com outras linguagens:**

  - **Utilizando outras linguagens dentro de Julia:** Muitos pacotes estão disponíveis, por exemplo, para utilizar outras línguagens dentro de Julia.
    Por exemplo:
    * JavaCall.jl, para chamar Java de Julia
    * Mathematica.jl, para chamar Mathematica
    * Rust e node-julia permitem que se chame JavaScript/node.js
    * Polyglot.jl serve para PHP, Perl and outras linguagens
  - **Utilizando Julia em outras linguagens:** A **API Julia C** permite a utilização de Julia em C, como a maioria das linguagens pode chamar C, esta API pode ser extendida para que linguagens possam ser integradas com C, como por exemplo C++, C# ou Python.

7 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 de Julia, proporcionando um ambiente baseado em passagem de mensagens entre vários processos que podem ser executados na mesma máquina ou em máquinas remotas. Nesse sentido, ele implementa o modelo do ator (como Erlang, Elixir, e Dart fazem), mas poderá se observar que o código com que se lida esta em um nível muito mais elevado do que receber e enviar mensagens entre processos, ou os "workers" (processadores). O desenvolvedor precisa apenas gerenciar explicitamente o processo a partir do qual todos os outros "workers" são iniciados. As operações de envio e recebimento de mensagens são encapsuladas por operações de um nível mais alto que se parecem muito com chamadas de função.

7.1 Tasks

Julia tem um sistema nativo próprio para rodar tarefas(**"tasks"**), as mesmas em geral conhecidas como co-rotinas. Com as mesmas um processo computacional que gera um valor (utilizando-se da função `produce`) pode ser suspenso em uma task, enquano outro processo, no caso um cliente desta tarefa pode consumir seus valores conforme necessário(utilizando-se da função `consume`). Esta estrutura é considerada similar ao que se faz com o `yield` em Python.

Como exemplo tem-se 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**.

```
function fib_producer(n)
   a, b = (0, 1)
   for i = 1:n
      produce(b)
      a, b = (b, a+b)
   end
end

tsk1 = Task( () -> fib_producer(10) )

consume(tsk1) # will return -> 1
consume(tsk1) # will return -> 1
consume(tsk1) # will return -> 2
consume(tsk1) # will return -> 3
consume(tsk1) # will return -> 5
consume(tsk1) # will return -> 8
consume(tsk1) # will return -> 13
consume(tsk1) # will return -> 21
consume(tsk1) # will return -> 34
consume(tsk1) # will return -> 55
consume(tsk1) # will return -> nothing # Task (done) @0x0000000005696180
```

As funções `produce` e `consume` utilizam uma função mais primitiva chamada `yieldto`.

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.

Tarefas devem ser vistas como uma forma cooperativa de "multitasking" em uma única "thread". Alternar entre as threads não consome espaço na pilha de execução("stack"), em geral elas tem um "overhead" muito baixo, sendo assim pode-se utilizar muitas delas sem maiores preocupações.

Os mesmos valores podem também ser facilmente consumidos dentro de um laço for, onde a variável **n** torna-se a cada iteração o valor produzido pela “Task”.

```
for n in tsk1
   println(n)
end
```

O trecho acima produz o seguinte resultado: 1 1 2 5 8 13 21 34 55.

O construtor de uma ”Task” precisa ser uma função com 0 argumentos, por isso é escrito como uma função anonima:

```
tsk1 = Task( () -> fib_producer(10) )
```

porém uma alternativa para isso é o uso da macro `@task`:

```
tsk1 = @task fib_producer(10)
```

7.2 Criando Processos - 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 sintace acima se n = 7 tem-se 8 ”workers”, 1 para o REPL shell e outros 7 para a realização de tarefas paralelas. Os ids para os ”workers” podem ser obtidos a partir da função ”workers()”.

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

Se em algum momento for necessário adicionar um novo ”worker” a função `addprocs(n)`, onde **n** representa a quantidade que se quer adicionar. Sua implementação default adiciona os processos na mesma máquina mas uma sobrecarga do método aceita argumentos para iniciar processos em máquinas remotas via SSH.

```
workers() #receives a n element array with the workers
7-element Array{Int64,1}: 2, 3, 4, 5, 6, 7, 8
```

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, julia deverá ser inicializado da seguinte forma:

No caso machines é um arquivo que contem o nome dos arquivos os quais se quer utilizar:

```
node01
node01
node02
node02
node03
```

7.3 Troca de Mensagens - Low-level communications

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, que irá recuperar o valor da referência:

```
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(..))`.

```
fetch(r1)
1000000
```

Pode-se utilizar também a macro `@spawnat` que recebe como argumentos o id do "worker" assim como a função a ser executada.

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

Isso foi ainda mais facilitado pela criação da macro `@spawn` que so precisa receber a função, e escolhe o "worker" baseado em uma lógica interna de otimização.

```
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 funo 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
```

7.4 Laços Paralelos e "maps" - 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, este é a macro `@parallel`.

Para calcular uma aproximação para π usando o famoso problema da agulha de Buffon. Se cair uma agulha em um chão com tiras paralelas de madeira, qual é a probabilidade de que a agulha vai cruzar uma linha entre duas tiras? Sem que se entre no mérito matemático do problema, uma função `buffon (n)` pode ser deduzida a partir do modelo que retornam uma aproximação para π quando jogando a agulha **n** vezes (assumindo que o comprimento da agulha **l** e da largura entre as tiras são ambos iguais a 1).

```
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 adicionados estes "workers" ao processo principal**. Ela divide o intervalo de dados, e o distribui para cada processo. 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 6,3 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!

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 matrizes. Pode-se fazer isso sequencialmente da seguinte forma:

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

Para paralelizar a execução basta remover o laço a chamada da função `println(rank(arr))` e adicionar a função `pmap` com o primeiro parametro sendo a função que se quer chamar e o segundo a coleção de dados à qual se queira aplicar a função.

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

Comparação de tempos:

```
#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 mquina com 7 workers criados
@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 mquina 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)
```

Como se pode observar, não foi possível reproduzir o ganho mostrado na literatura, o que reforça a tese de que se deve testar sempre o paralelismo em cada caso pois sua performance pode variar dependendo da arquitetura de processadores, tipo da máquina e realidade do problema apresentado em si.

7.5 Distributed Arrays

Quando os cálculos tem que ser feitos sob uma grande coleção de dados, esta coleção pode ser distribuida de modo que cada processo processe em paralelo sobre uma porção diferente da matriz. 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`.

Vale salientar que a maioria das funções para criação dos `DArrays` que são as mesmas dos `Arrays` porem prefixadas pela letra **d**, segue algumas algumas abaixo:

```
dzeros(100,100,10)
dones(100,100,10)
drand(100,100,10)
drandn(100,100,10)
dfill(x,100,100,10)
```

Algumas das principais funções para `DArrays`:

- `distribute(a::Array)` converte um `Array` local para um `DArray`.
- `localpart(a::DArray)` Obtém a porção local (no "Worker") do `Array`.
- `localindexes(a::DArray)` traz uma tupla com os indexes pertinentes ao processo local.
- `convert(Array, a::DArray)` traz todos os dados do `DArray` para o processo local.

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 dois processos não 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
 0.923559 0.681605 0.839828 0.768984 0.73684 0.863426 0.764039

```

O construtor `DArray` primitivo tem a seguinte assinatura:
`DArray(init, dims[, procs, dist])`
Onde o argumento **init** é uma função que aceita uma tupla de intervalos de "indexes"(**i.e (5:8, 18:21)**). Esta função deve alocar uma porção local do `DArray` e inicializá-o para os "indexes" especificados. O argumento **dims** é o tamanho total do `DArray`. O argumento **procs** é opcional e especifica um vetor de IDs de quais "workers" se deverá utilizar para a distribuição do `DArray`. O argumento **dist** é um vetor inteiro especificando quantas porções do `DArray` deverão ser divididos em para cada "worker".

`DArrays` também pode ser criado com a macro `@parallel` como se segue:

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

O seguinte trecho de código é muitas vezes usado para construir um `Array` distribuído dividido sobre os "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
```

8 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/files/julia_express.pdf
- https://en.wikipedia.org/wiki/Buffon's_needle