ERAD | RS 2022

Computação de Alto Desempenho em Julia

Minicurso 4

Roberto M. Velho, Rafael B. Klausner, Matheus S. Serpa, Adriano M. A. Côrtes
roberto.velho@gmail.com, rafa.bench@gmail.com, msserpa@inf.ufrgs.br, adriano@nacad.ufrj.br

APRESENTAÇÃO

Roberto M. Velho

Formação:

- Graduação em Matemática Aplicada (UFRJ 2009)
- Mestrado em Matemática Aplicada (UFRJ 2011)
- Doutorado em Applied Mathematics and Computer Science (KAUST, Saudi Arabia 2017)
- Pós-doutorado (IMPA 2019) (UFRGS 2020)

APRESENTAÇÃO

Rafael B. Klausner
linkedin.com/in/rafael-benchimol-klausner/

Atividades:

- Analista @ PSR Consultoria Ltda

Formação:

- Bacharelado em Matemática Aplicada (UFRJ 2021)

APRESENTAÇÃO

Matheus S. Serpa
linkedin.com/in/matheusserpa/

Atividades:
- Lead Data Scientist @DigiFarmz Smart Agriculture **[We're hiring]**
- Data Science Instructor @TargetTrust

Formação:
- Bacharelado em Ciência da Computação (UNIPAMPA 2015)
- Mestrado em Computação (UFRGS 2018)

  Período sanduíche na Université de Neuchâtel - Suíça
- Doutorado em andamento em Computação (UFRGS)

APRESENTAÇÃO

Adriano M. A. Côrtes <adriano.cortes@ufrj.br>

Atividades:

- Professor Adjunto do Departamento de Matemática Aplicada (DMA) do IM/UFRJ
- Pesquisador do Núcleo de Atendimento a Computação de Alto-Desempenho (NACAD) da COPPE/UFRJ

Formação:

- Bacharel em Ciência da Computação (UFRJ 2004)
- Mestre em Matemática Aplicada (UFRJ 2006)
- Doutor em Mecânica Computacional (UFRJ 2013)

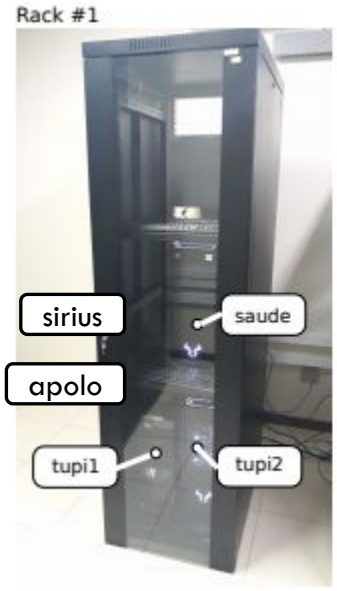
PARQUE COMPUTACIONAL DE ALTO DESEMPENHO (PCAD)

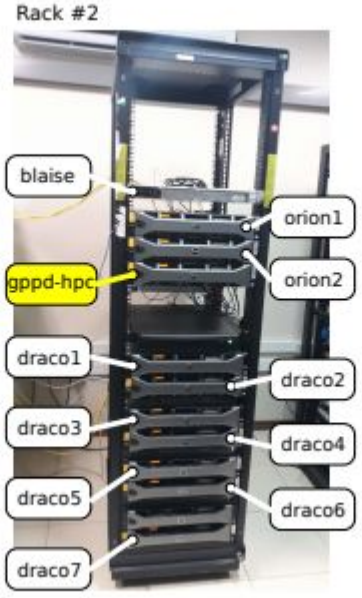
Infraestrutura computacional

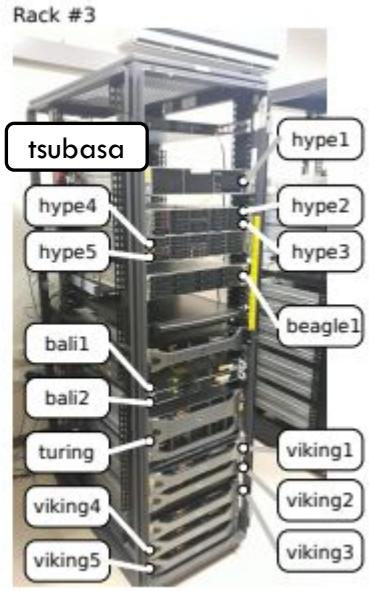
Possui aproximadamente 40 nós, 700+ núcleos de CPU e 73000+ de GPU

Site: http:/

cei1
cei2
cei3
cei4
cei5







knl1
knl2
knl3
knl4

TESTANDO LOGIN NO PCAD

Download da chave

```
wget https://abre.ai/pcad-erad22 && chmod 700 pcad-erad22
```

Login remoto

```
ssh -i pcad-erad22 workshop@gppd-hpc.inf.ufrgs.br
```

Copie os exercícios

```
cp -r ~/workshop/ ~/seu-nome-sobrenome/
cd ~/seu-nome-sobrenome/
```

ERAD | RS 2022

PROGRAMAÇÃO PARALELA

Minicurso 4

Computação de Alto Desempenho em Julia

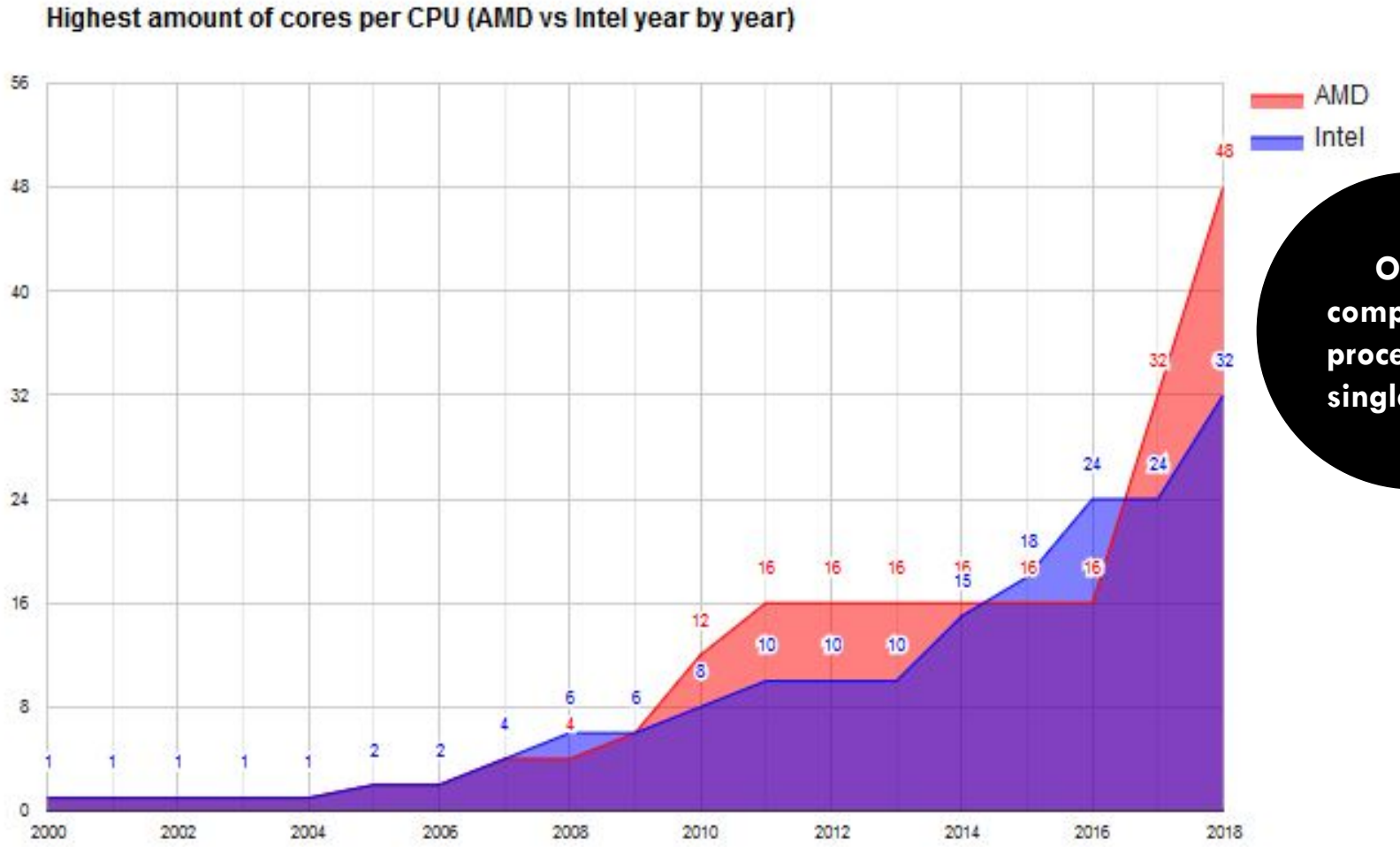
POR QUE ESTUDAR PROGRAMAÇÃO PARALELA?

Os programas já não são rápidos o suficiente?

As máquinas já não são rápidas o suficiente?

REQUISITOS SEMPRE MUDANDO

EVOLUÇÃO DA INTEL E AMD



Onde comprar um processador single-core?

POR QUE PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reduzir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

POR QUE PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reduzir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

Outros motivos são:

- Utilizar recursos computacionais subaproveitados.
- Ultrapassar limitações de memória quando a memória disponível num único computador é insuficiente para a resolução do problema.
- Ultrapassar os limites físicos que atualmente começam a restringir a possibilidade de construção de computadores sequenciais cada vez mais rápidos.

COMO FAZER ATIVIDADES EM PARALELO? NO MUNDO REAL

Você pode enviar cartas/mensagens aos amigos e pedir ajuda

Você pode criar uma lista de tarefas (pool)

Quando um amigo ficar atoa, pegue uma nova tarefa da lista

Você pode ajudar na tarefa ou então ficar apenas gerenciando

Mais amigos, mais tempo para eles chegarem/ se acomodarem

Tarefas curtas ou longas?

Muitos amigos olhando e riscando a lista?

Precisa gerenciar algo mais?

OPÇÕES PARA CIENTISTAS DA COMPUTAÇÃO

1. Crie uma **nova linguagem** para programas paralelos
2. Crie um **hardware** para extrair paralelismo
3. Deixe o **compilador** fazer o trabalho sujo
   - Paralelização automática
   - Ou **crie anotações no código sequencial**
4. Use os recursos do **sistema operacional**
   - Com memória compartilhada – threads
   - Com memória distribuída – SPMD
5. Use a **estrutura dos dados** para definir o paralelismo
6. Crie uma **abstração de alto nível** – Objetos, funções aplicáveis, etc.

PRINCIPAIS MODELOS DE PROGRAMAÇÃO PARALELA EM JULIA

Programação em Memória Compartilhada

- Programação usando processos ou threads.
- Decomposição do domínio ou funcional com granularidade fina, média ou grossa.
- Comunicação através de **memória compartilhada**.
- Sincronização através de mecanismos de exclusão mútua.

Programação em Memória Distribuída

- Programação usando processos distribuídos
- Decomposição do domínio com granularidade grossa.
- Comunicação e sincronização por **troca de mensagens**.
- Pode ser usado pacotes conhecidos como MPI

Programação em GPU

- O compilador de GPU dá a habilidade de rodar Julia nativamente nas GPUs

FATORES DE LIMITAÇÃO DO DESEMPENHO

Código Sequencial: existem partes do código que são inerentemente sequenciais (e.g. iniciar/terminar a computação).

Concorrência/Paralelismo: o número de tarefas pode ser escasso e/ou de difícil definição.

Comunicação: existe sempre um custo associado à troca de informação e enquanto as tarefas processam essa informação não contribuem para a computação.

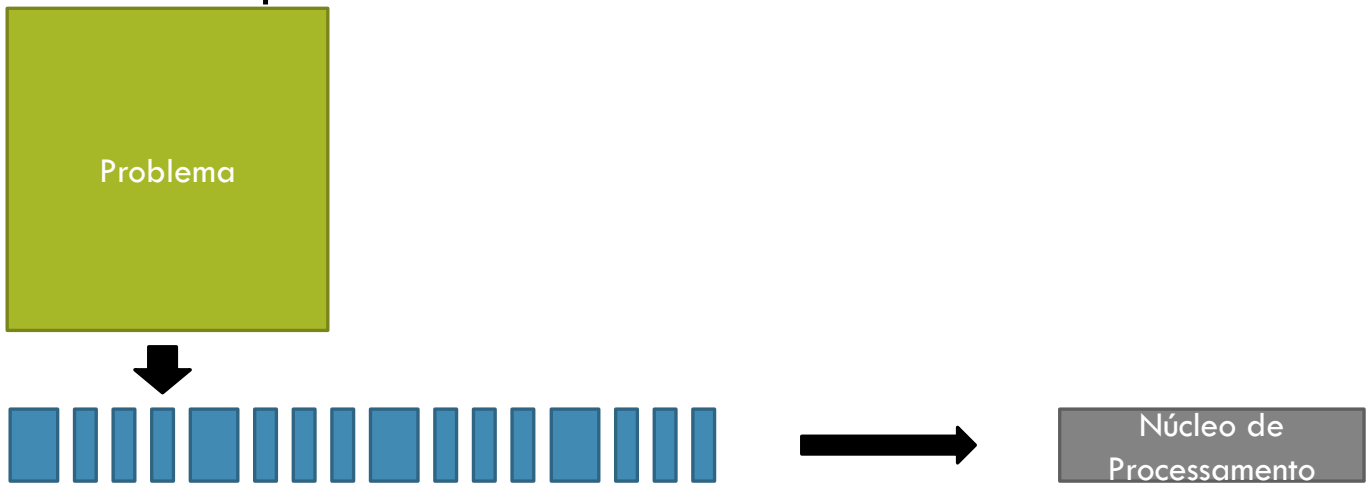
Sincronização: a partilha de dados entre as várias tarefas pode levar a problemas de contenção no acesso à memória e enquanto as tarefas ficam à espera de sincronizar não contribuem para a computação.

Granularidade: o número e o tamanho das tarefas é importante porque o tempo que demoram a ser executadas tem de compensar os custos da execução em paralelo (e.g. custos de criação, comunicação e sincronização).

Balanceamento de Carga: ter os processadores maioritariamente ocupados durante toda a execução é decisivo para o desempenho global do sistema.

PROGRAMAÇÃO SEQUENCIAL

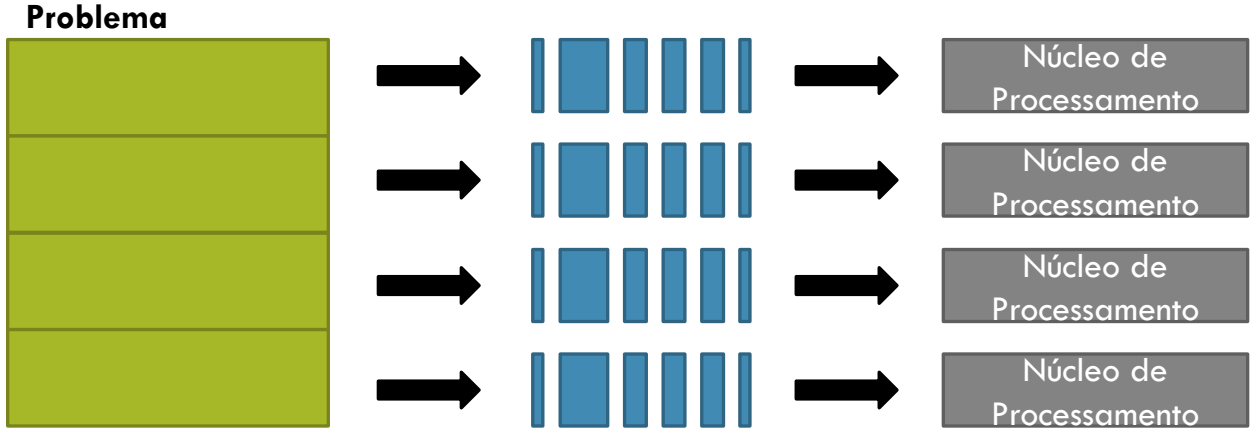
Considerado **programação sequencial** quando este é visto como uma série de instruções sequenciais que devem ser executadas num único processador.



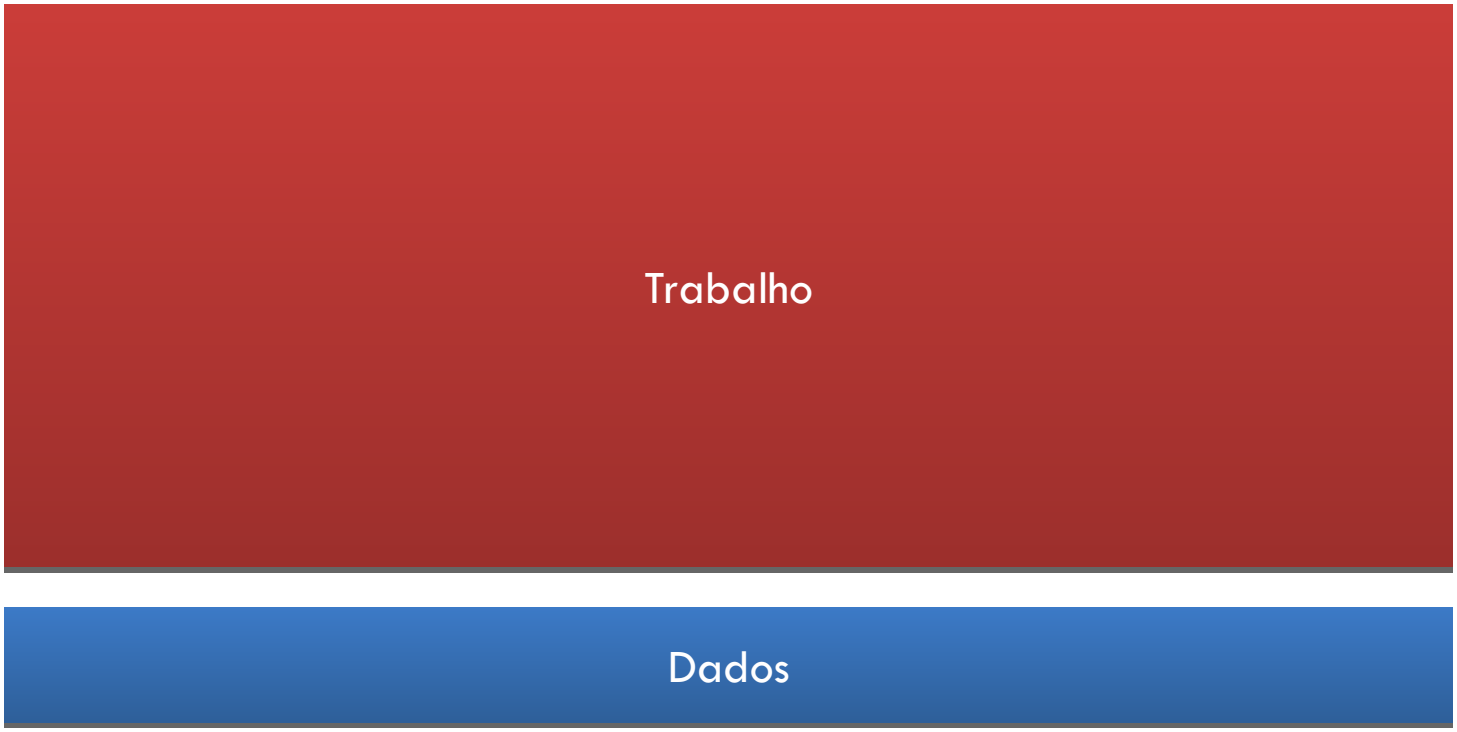
PROGRAMAÇÃO PARALELA

Considerado **programação paralela** quando este é visto como um conjunto de partes que podem ser resolvidas concorrentemente.

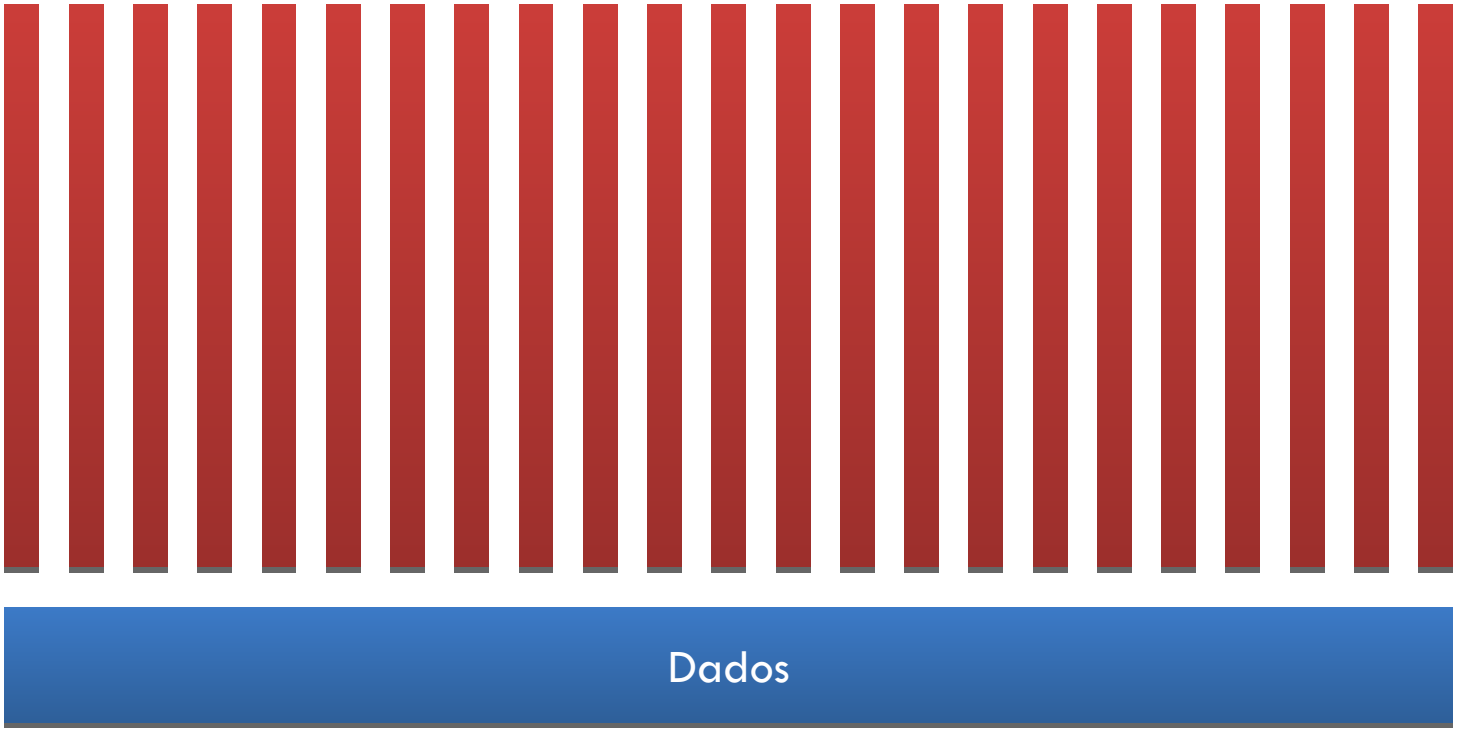
Cada parte é igualmente constituída por uma série de instruções sequenciais, mas que no seu conjunto podem ser executadas simultaneamente.



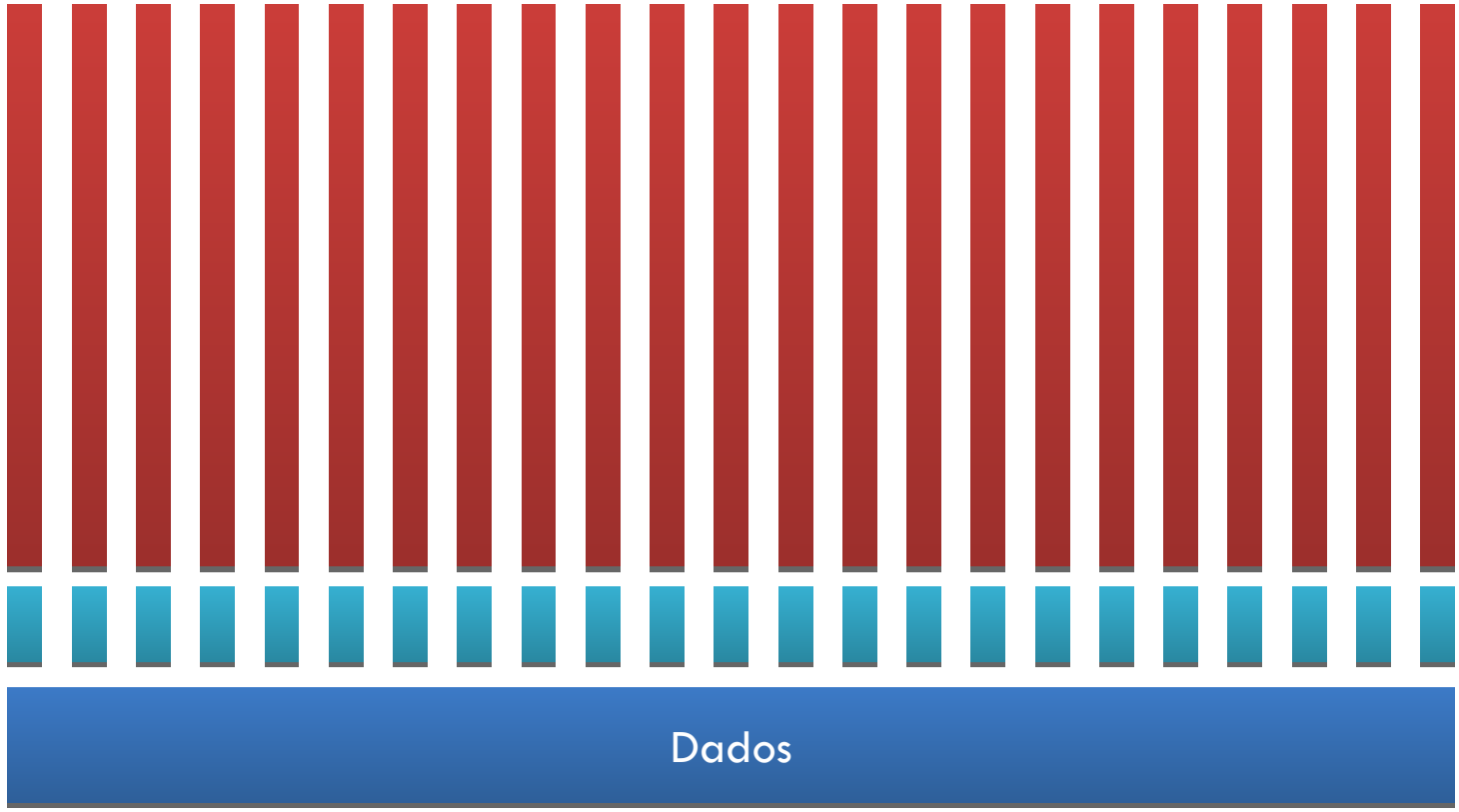
COMO IREMOS PARALELIZAR? **PENSANDO!**



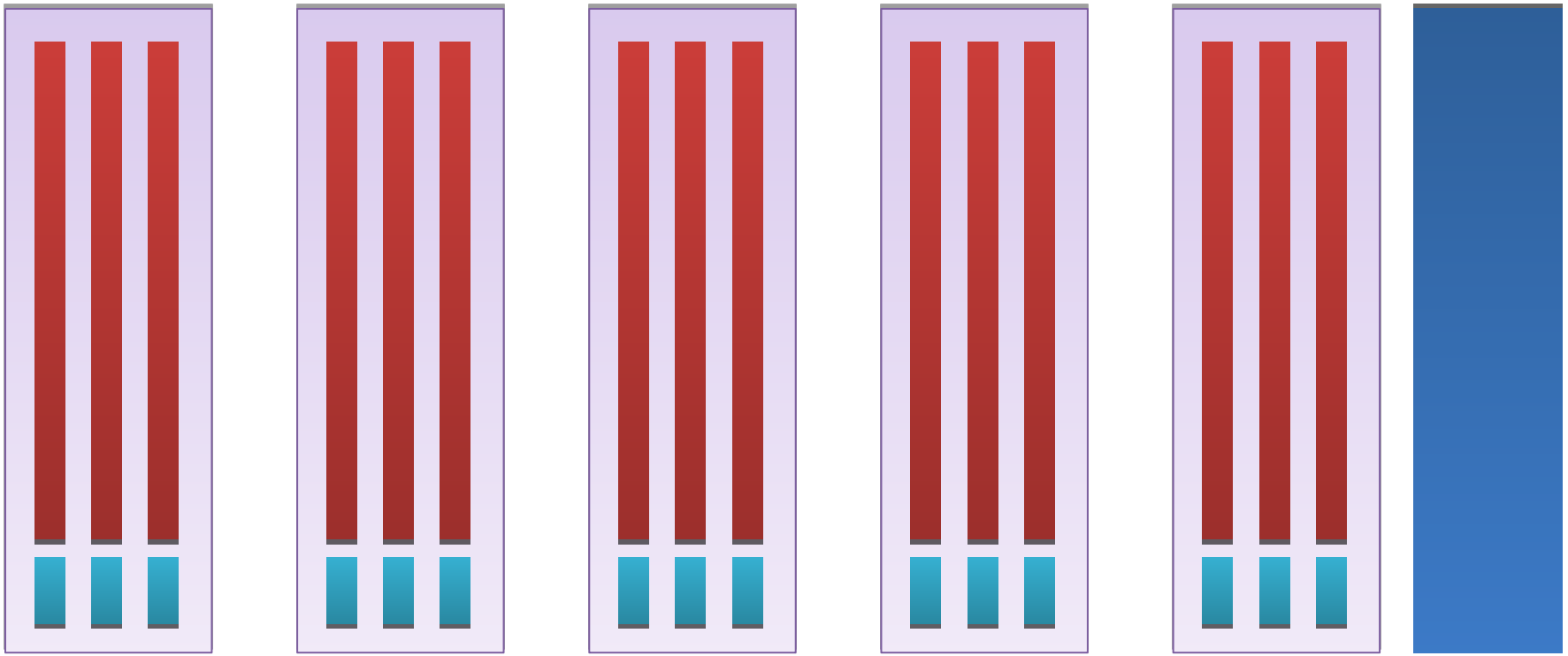
COMO IREMOS PARALELIZAR? PENSANDO!



COMO IREMOS PARALELIZAR? PENSANDO!



COMO IREMOS PARALELIZAR? PENSANDO!



Divisão e Organização lógica do nosso algoritmo paralelo

ERAD | RS 2022

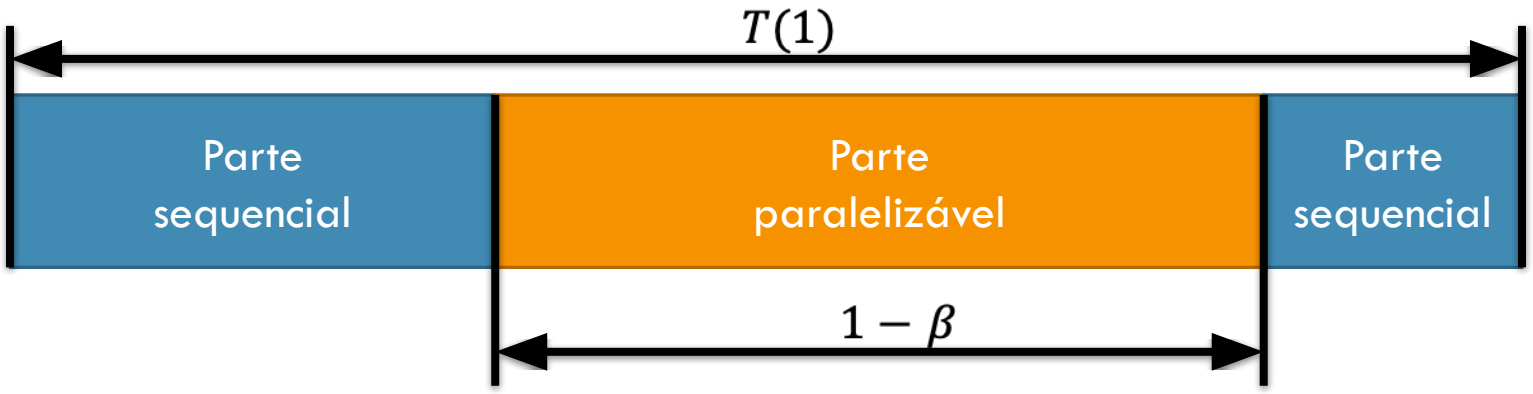
LEI DE AMDAHL (1967)

Minicurso 4

Computação de Alto Desempenho em Julia

LEI DE AMDAHL

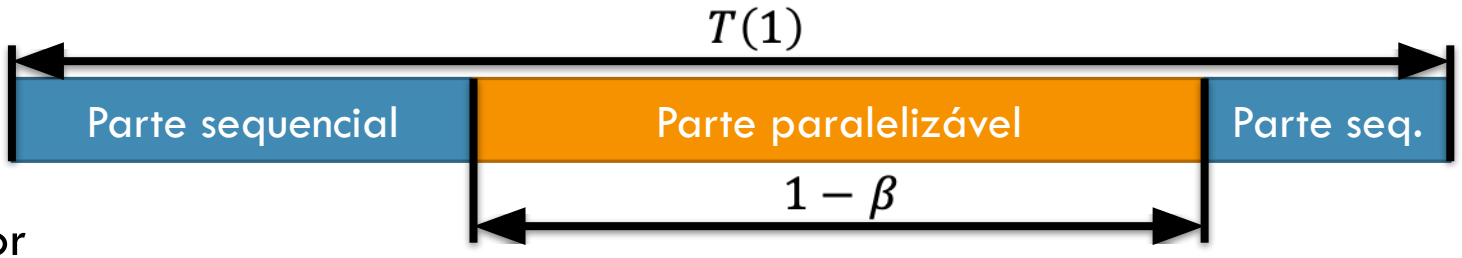
Tempo de execução em um único processador:



β = fração de código que é puramente sequencial

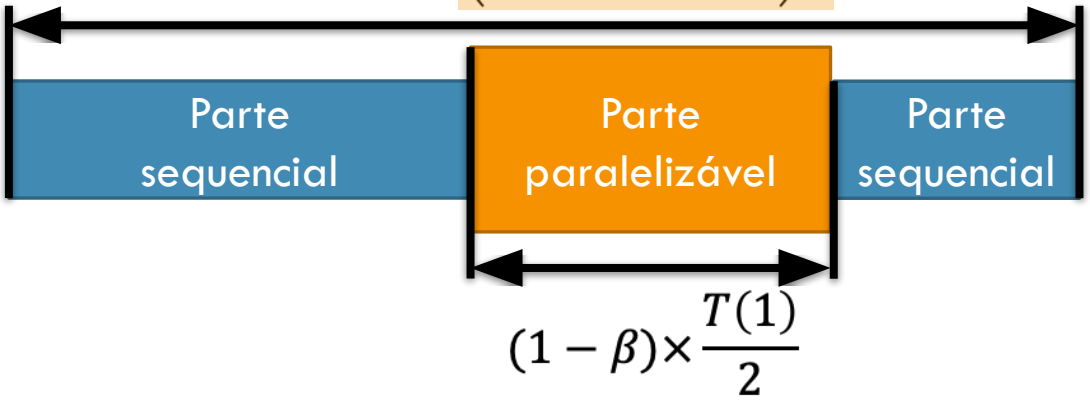
LEI DE AMDAHL

Tempo de execução em um único processador



Tempo de execução em 2 processadores

$$T(2) = T(1)\times\beta + \left((1-\beta)\times\frac{T(1)}{2}\right)$$



ERAD | RS 2022

PROGRAMAÇÃO EM JULIA

Minicurso 4

Computação de Alto Desempenho em Julia

Baixando o Julia

https://julialang.org/downloads/

DOCUMENTAÇÃO OFICIAL

Documentação Oficial do Julia

https://docs.julialang.org/en/v1/

BIBLIOGRAFIA DE JULIA

Think Julia: How to think like a Computer Scientist

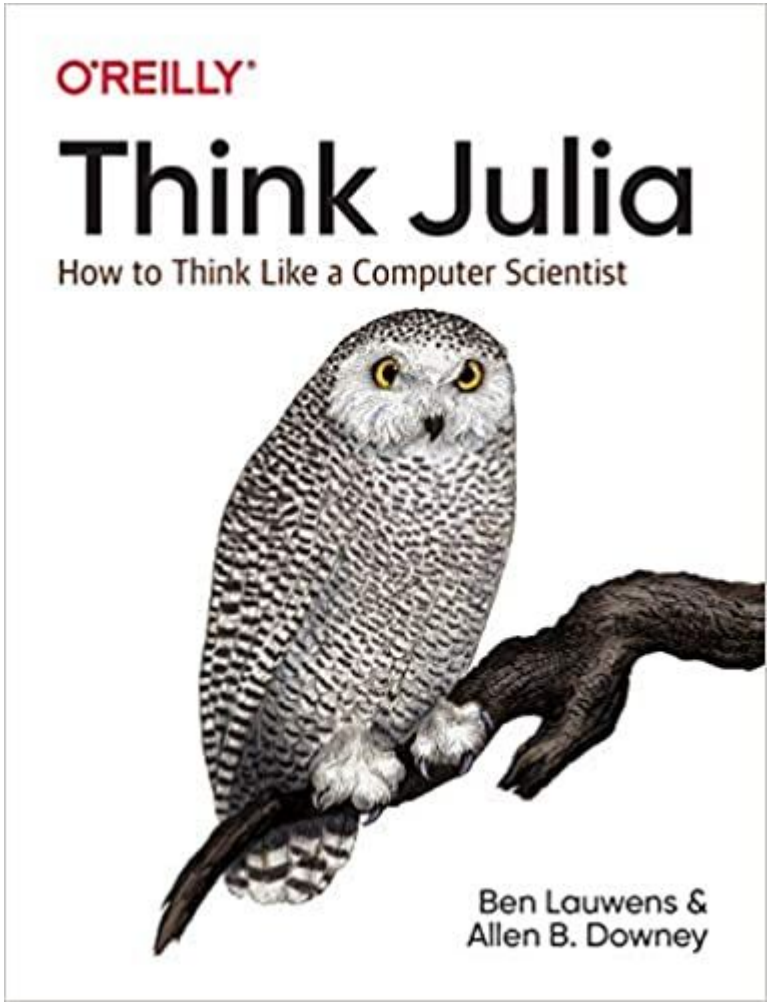
Autores: Ben Lauwen,
Allen B. Downey
Editora: O' Reilly

Ano: 2019

Disponível gratuitamente em:

https://benlauwens.github.io/

ThinkJulia.jl/latest/book.html

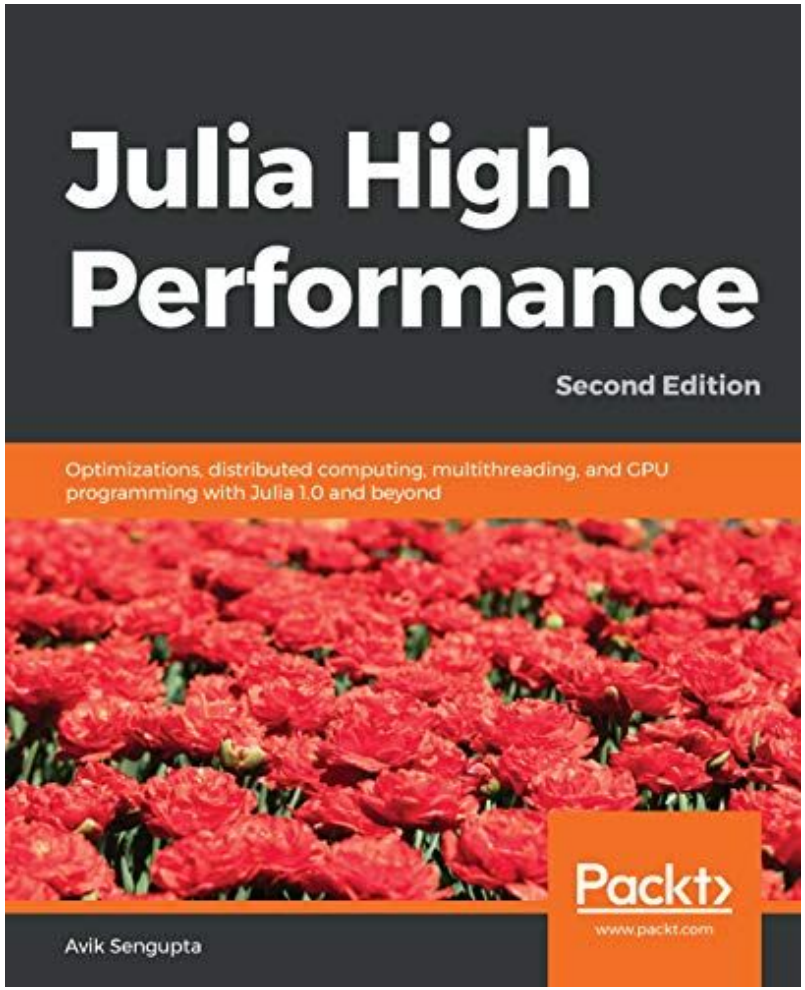


BIBLIOGRAFIA DE JULIA

Julia High Performance: Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond

Autores: Avik Sengupta
Editora: Packt

Ano: 2019



REVISÃO SOBRE JULIA

Comando ?

```
help?> println
search: println printstyled print sprint isprint

  println([io::IO], xs...)

  Print (using print) xs followed by a newline. If io is not supplied, prints to stdout.

  See also printstyled to add colors etc.

  Examples
  ==========

  julia> println("Hello, world")
  Hello, world

  julia> io = IOBuffer();

  julia> println(io, "Hello", ',', " world.")

  julia> String(take!(io))
  "Hello, world.\n"
```

REVISÃO SOBRE JULIA

Comando ;

```
shell> ls
1-helloWorld    2-vectorSum     3-selectionSort 4-monteCarlo    5-antColony     6-vectorAdd     7-mandelbrot
```

REVISÃO SOBRE JULIA

Comando]

```
(@v1.7) pkg> add LinearAlgebra
   Resolving package versions...
  No Changes to `~/.julia/environments/v1.7/Project.toml`
  No Changes to `~/.julia/environments/v1.7/Manifest.toml`
```

```
1  using Base.Threads
2
3  function vector_sum(v, N)
4      sum = 0.0
5
6      @threads for i in 1:N
7          sum += v[i]
8      end
9
10     return Int128(sum)
11
12 end
13
14
15 function main()
16     N = 10^8
17
18     vector = ones(Int64, N)
19
20     sumv = @time vector_sum(vector, N)
21
22     println("Sum is $(sumv)")
23 end
24
25 main()
```

using para importar bibliotecas

function para definir funções

function, for, if, while precisam de um fechando com **end**

Int128 converte uma variável para inteiro

ones cria um vetor de 1's

@time mede o tempo de execução de uma função

println é um print com quebra de linha automática

curiosidade: o operador ÷ retorna uma divisão inteira

ERAD | RS 2022

Threads

Minicurso 4

Computação de Alto Desempenho em Julia

INTRODUÇÃO

A macro é um dos modelos de programação paralelas mais usados hoje em dia.

Esse modelo é relativamente fácil de usar, o que o torna um bom modelo para iniciar o aprendizado sobre escrita de programas paralelos.

Observações:

- Assumo que todos sabem programar em linguagem Julia ou similar, como Python, Matlab ou R.

VISÃO GERAL JULIA:

OpenMP: Uma API para escrever aplicações Multithreaded

Um conjunto de **diretivas do compilador** e **biblioteca de rotinas** para programadores de aplicações paralelas

+ variáveis de ambiente

Simplifica muito a escrita de programas multi-threaded (MT)

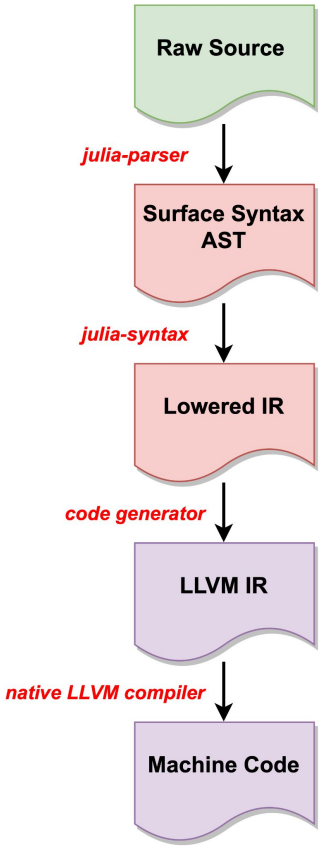
Padroniza 20 anos de prática SMP

Compilação JIT da Julia:

O processo de compilação pode ser desmembrado em 4 estágios (figura ao lado). Os três primeiros gerenciados pela própria Julia e o último por um compilador LLVM.

No **primeiro,** o código fonte é analisado pela Julia, e é criada uma *Abstract Syntax Tree* (AST). Por exemplo, é nesse momento que operadores são transformados em suas respectivas chamadas a funções. É constituída de expressões (*Expr*) e símbolos (*Symbols*).

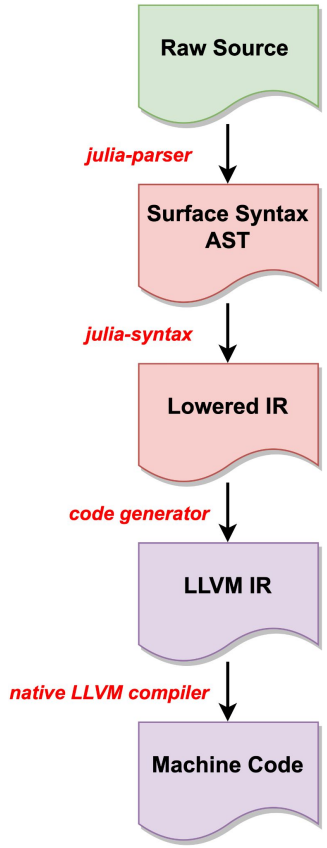
```
julia> dump(:(1+2-3))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol -
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol +
        2: Int64 1
        3: Int64 2
    3: Int64 3
```



Compilação JIT da Julia:

No **segundo**, a Julia AST é transformada numa representação intermediária (*Lowered Intermediate Representation*). São aplicadas algumas otimizações como *inlining*, porém o principal é a inferência de tipos (*type inference*). O procedimento de inferência de tipos é um ponto crucial para o desempenho de Julia, uma vez que garante um código final (*machine code*) tipado, portanto com uma eficiência similar a de uma linguagem compilada, como a linguagem C.

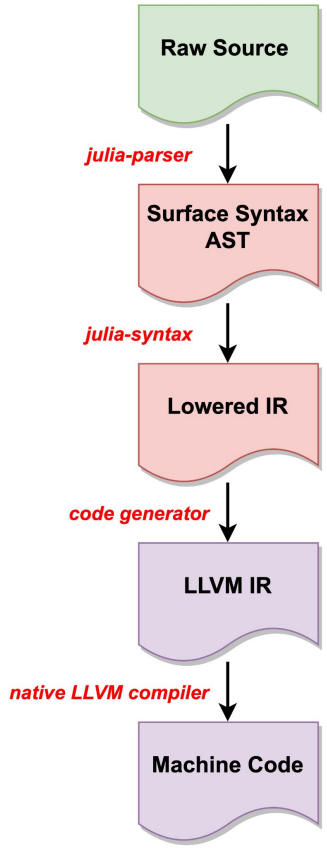
```
julia> function my_sum(n)
           sum = 0; for i in 1:n sum+=i end
           return sum
       end
my_sum (generic function with 2 methods)

julia> @code_typed my_sum(5)
```



Compilação JIT da Julia:

No **terceiro**, o código é transformado numa representação intermediária com vistas a compilação pelo compilador LLVM. Ele pode ser inspecionado com a macro @code_llvm, porém seu entendimento já é mais complexo.

No **quarto**, e último estágio, o código é transformado numa representação de baixo nível nativa, e específica para a arquitetura, chamada *LLVM assembly code*. Ele pode ser inspecionado com a macro @code_native. Nela podemos visualizar, por exemplo, otimizações aplicadas pelo compilador, como o uso de registradores e instruções vetoriais.



Recomendações de eficiência:

- Respeitar o layout de memória. Julia armazena array multidimensionais do último índice para o primeiro, por exemplo, matrizes são armazenadas por coluna.
- Sempre que possível pré-aloque espaço para o resultado.
- Iterações que você pode garantir o acesso seguro, use a macro @inbounds para evitar a verificação.
- *Slicing* de arrays promove cópias internas, evite-as usando a macro @view em arrays.
- Em operações elemento a elemento use *broadcasting* por meio do operador ponto (por exemplo, a .+ b, onde a e b são arrays).

SINTAXE BÁSICA - Threads

Chamada da biblioteca de paralelização:

using Base.Threads

A maioria das construções são feitas por macro.

Threads.@threads begin ... end

- Exemplo:

```
Threads.@threads for i = 1:nthreads
    a[i] = 1
end
```

NOTAS DE INICIALIZAÇÃO

```
$ export JULIA_NUM_THREADS = 4
$ julia -t 4
$ julia -t auto
```

Para shell bash

A partir do Julia 1.7

Também funciona no Windows!

Até mesmo no Visual Studio!

Mas vamos usar Linux ☺

EXEMPLO, PARTE A

Verifique se seu ambiente funciona

Escreva um programa que escreva “hello world”.

```
    println("Hello World!")
```

EXEMPLO, PARTE B

Verifique se seu ambiente funciona

Escreva um programa que escreva “hello world”.

```
nthreads = 1

for thread = 1:nthreads
    println("$(thread) of $(nthreads) - Hello World!")
end
```

EXEMPLO, PARTE C

Verifique se seu ambiente funciona

Vamos **adicionar o número da thread** ao “hello world”.

```
using Base.Threads

@threads for thread = 1:nthreads()
    println("$(threadid()) of $(nthreads()) - Hello World!")
end
```

EXEMPLO

```
using Base.Threads

@threads for thread = 1:nthreads()
    id =threadid()
    println(“$(id) of  $(nthreads()) - Hello World”)
end
```

Região paralela com um número de threads

Funções da biblioteca que retornam o thread ID e número de threads

Fim da região paralela

ERAD | RS 2022

ESCOPO DAS VARIÁVEIS

Minicurso 4

Computação de Alto Desempenho em Julia

ESCOPO PADRÃO DE DADOS

A maioria das variáveis são compartilhadas por padrão

Região paralela
Outside Global
Inside Privado

Heap Global
Stack Privado

Variáveis globais são compartilhadas entre as threads:

- Variáveis de escopo de arquivo e estáticas
- Variáveis alocadas dinamicamente na memória (malloc, new)

Mas nem tudo é compartilhado:

- Variáveis da pilha de funções chamadas de regiões paralelas são privadas
- Variáveis declaradas dentro de blocos paralelos são privadas

COMPARTILHAMENTO DE DADOS: MUDANDO OS ATRIBUTOS DE ESCRITA

Em Julia há duas formas de tratar escopo de variáveis

- GLOBAL
- LOCAL

Em qualquer laço ou função, mesmo serial, o escopo das variáveis é local, a menos que use a keyword global na definição da variável

COMPARTILHAMENTO DE DADOS:

```
local a = zeros(10)
Threads.@threads for i = 1:N
    a[i] = Threads.threadid()
end
```

Erro! Vetor "a" não está definido

```
global a = zeros(10)
Threads.@threads for i = 1:N
    a[i] = Threads.threadid()
end
```

Executa o código corretamente e é o padrão

COMPARTILHAMENTO DE DADOS:

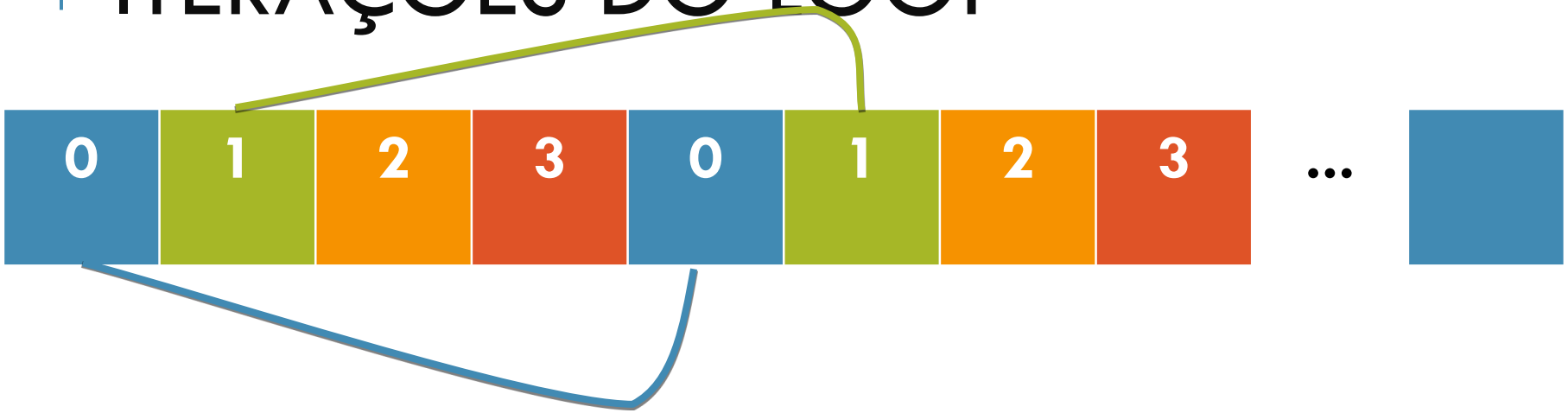
```
local a = zeros(10)
function paralelize()
    Threads.@threads for i = 1:N
        a[i] = Threads.threadid()
    end
end
paralelize()
```

Erro! Vetor "a" não está definido

```
global a = zeros(10)
function paralelize()
    Threads.@threads for i = 1:N
        a[i] = Threads.threadid()
    end
end
paralelize()
```

Executa o código corretamente

DISTRIBUIÇÃO CÍCLICA DE ITERAÇÕES DO LOOP



A macro @threads divide o espaço de interação do loop de forma cíclica

DISTRIBUIÇÃO CÍCLICA DE ITERAÇÕES DO LOOP

```
using Base.Threads

N = 10
a = zeros(N)
Threads.@threads for i = 1:N
    a[i] = Threads.threadid()
end
display(a)
```

```
10-element Vector{Float64}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

EXERCÍCIO 2.1: VECTOR SUM

```
function vector_sum(v, N)
    sum = 0.0
    for i in 1:N
        sum += v[i]
    end
    return sum
end
```

SOLUÇÃO 2.2 VECTOR SUM

```
function vector_sum(v, N)
    sum = 0.0
    @threads for i in 1:N
        sum += v[i] # Race Condition
    end
    return sum
end
```

COMO AS THREADS INTERAGEM?

OpenMP é um modelo de *multithreading* de memória compartilhada.

- Threads se comunicam através de variáveis compartilhadas.

Compartilhamento não intencional de dados causa **condições de corrida**.

- Condições de corrida: quando a saída do programa muda quando a threads são escalonadas de forma diferente.

Apesar de este ser um aspectos mais poderosos da utilização de threads, também pode ser um dos mais problemáticos.

O problema existe quando dois ou mais *threads* tentam acessar/alterar as mesmas estruturas de dados (condições de corrida).

Para controlar condições de corrida:

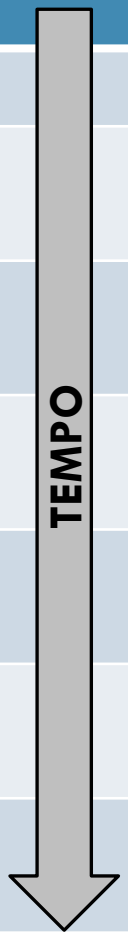
- Usar sincronização para proteger os conflitos por dados

Sincronização é cara, por isso:

- Tentaremos mudar a forma de acesso aos dados para minimizar a necessidade de sincronizações.

CONDIÇÕES DE CORRIDA: EXEMPLO

	Thread 0	Thread 1	sum
			0
	Leia sum 0		0
		Leia sum 0	0
		Some 0, 5 5	0
	Some 0, 10 10		0
		Escreva 5, sum 5	5
	Escreva 10, sum 10		10



15 !?

CONDIÇÕES DE CORRIDA: EXEMPLO

	Thread 0	Thread 1	sum
			0
		Escreva 5, sum 5	5
	Escreva 10, sum 10		10

Devemos garantir que **não importa a ordem de execução (escalonamento)**, teremos sempre um resultado consistente!

15 !?

SINCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

As duas formas mais comuns de sincronização são:

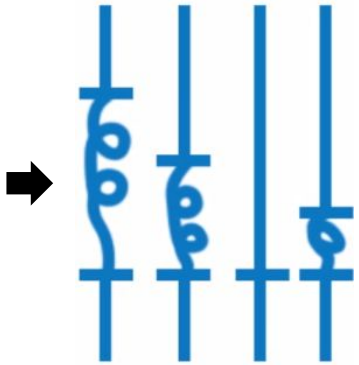
SINCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

As duas formas mais comuns de sincronização são:

Barreira: Cada *thread* espera na barreira

até a chegada de todas as demais



SINCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

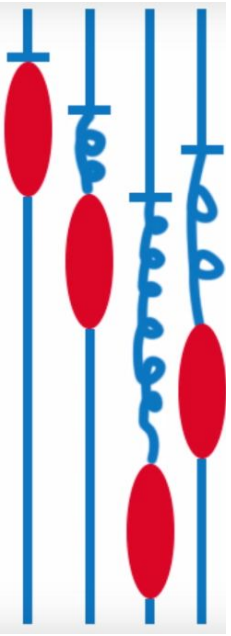
As duas formas mais comuns de sincronização são:

Barreira: Cada *thread* espera na barreira

até a chegada de todas as demais

Exclusão mútua: Define um bloco de código

onde apenas uma *thread* pode executar por vez.



SINCRONIZAÇÃO: BARRIER

Barrier: Cada *thread* espera até que as demais cheguem.

```
N = 1000
acc2 = Ref(0) # Pointer with value 0
l = Threads.ReentrantLock()
@threads for i in 1:N
    Threads.lock(l)
    acc2[] += 1 # Add 1 without race condition
    Threads.unlock(l)
end
println("Sum right is $(acc2[])")
```

SINCRONIZAÇÃO: ATOMIC

atomic prove exclusão mútua para operações específicas.

```
acc3 = Atomic{Int64}(0) # Atomic pointer with value 0
@threads for i in 1:1000
    atomic_add!(acc3, 1) # Add 1 thread safely
end
println("Sum atomic is $(acc3[])")
```

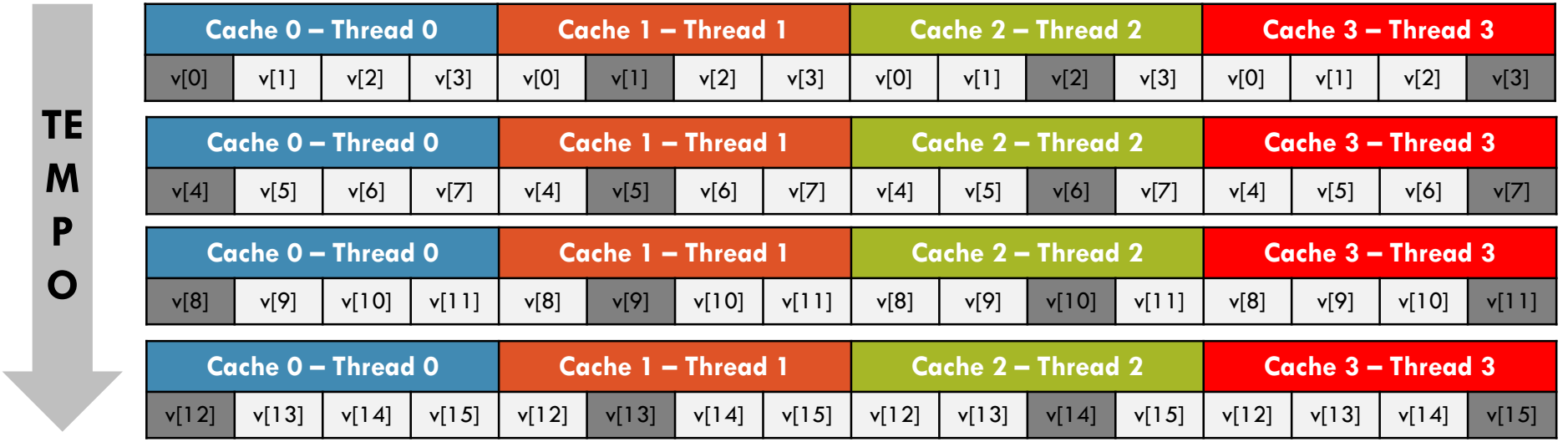
PRINCÍPIO DA LOCALIDADE

Programas repetem trechos de código e acessam repetidamente dados próximos.

Localidade Temporal: posições de memória, uma vez acessadas, tendem a ser acessadas novamente em um espaço curto de tempo.

Localidade Espacial: se um item é referenciado, itens cujos endereços sejam próximos dele tendem a ser referenciados em um espaço curto de tempo.

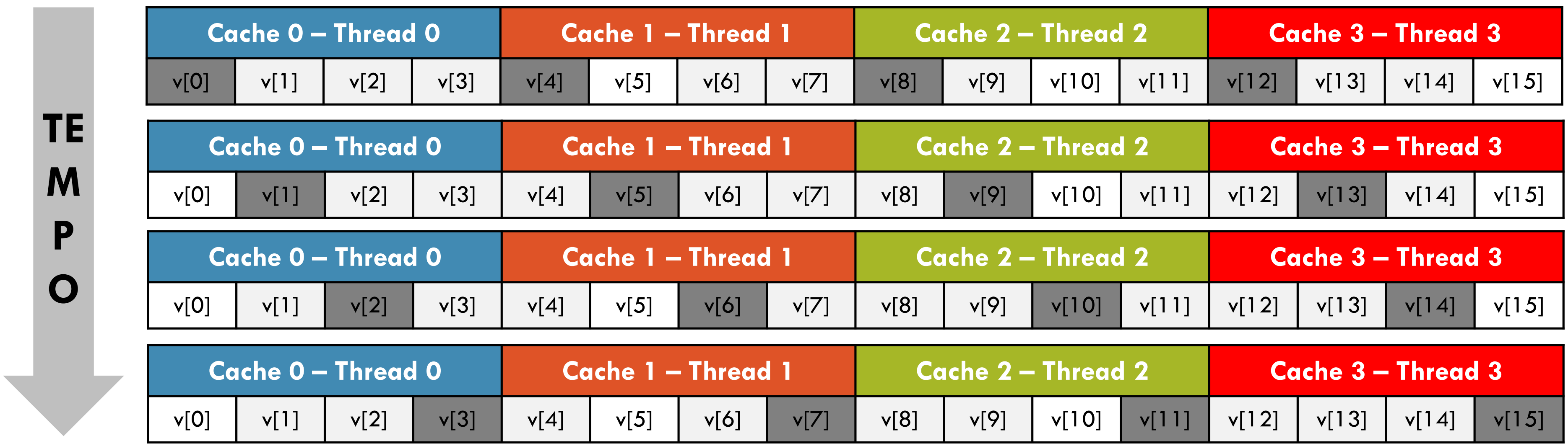
ACESSOS INTERCALADOS




```
for(i = myid;  i < N;  i += nthreads)
```

25% do conteúdo trazido para a cache é utilizado.

ACESSOS CONSECUTIVOS




```
for(i = ini;  i < end;  i++)
```

100% do conteúdo trazido para a cache é utilizado.

EXERCÍCIO 2.3 VECTOR SUM

```
function vector_sum(v, N)
    sum = 0.0
    for i in 1:N
        sum += v[i]
    end
    return sum
end
```

REDUÇÃO

Combinação de variáveis locais de uma *thread* em uma variável única.

- Essa situação é bem comum, e chama-se **redução**.
- O suporte a tal operação é fornecido pela maioria dos ambientes de programação paralela.

DIRETIVA REDUCTION

reduction(op : list_vars)

Dentro de uma região paralela ou de divisão de trabalho:

- Será feita uma cópia local de cada variável na lista
- Será inicializada dependendo da **op** (ex. 0 para +, 1 para *).
- Atualizações acontecem na cópia local.
- Cópias locais são “reduzidas” para uma única variável original (global).

```
1  using FLoops
2
3  @floop for i in 1:N
4      @reduce(sum += v[i])
5  end
```

EXERCÍCIO 2.4: VECTOR SUM

```
function vector_sum(v, N)
    sum = 0.0
    for i in 1:N
        sum += v[i]
    end
    return sum
end
```

EXERCÍCIO 3: SELECTION SORT

```
function selection_sort!(v, N)
    for i in 1:N-1
        j_min = i
        for j in i+1:N
            if v[j] < v[j_min]
                j_min = j
            end
        end
        tmp  = v[i]
        v[i] = v[j_min]
        v[j_min] = tmp
    end
    return v
end
```

ERAD RS 2022

Computação Distribuída

Minicurso 4

Computação de Alto Desempenho em Julia

Inicialização

Uma forma de trabalhar com programação distribuída em Julia é pelo pacote Distributed

```
$ julia - p 2

julia> using Distributed

julia> nworkers()
    2
```

```
$ julia

julia> using Distributed

julia> addprocs(2)

julia> nworkers()
    2
```

Macro distributed

julia -p 4

```
@sync @distributed for i in
1:10
    println("Worker $(myid()),
working on i=$i")
end
```

```
From worker 3:    Worker 3, working on i=4
From worker 3:    Worker 3, working on i=5
From worker 3:    Worker 3, working on i=6
From worker 4:    Worker 4, working on i=7
From worker 4:    Worker 4, working on i=8
From worker 5:    Worker 5, working on i=9
From worker 5:    Worker 5, working on i=10
From worker 2:    Worker 2, working on i=1
From worker 2:    Worker 2, working on i=2
From worker 2:    Worker 2, working on i=3
```

Função pmap

julia -p 4

```
pmap(i -> println("Worker $(myid()), working on i=$i"), 1:10)
```

```
From worker 3:    Worker 3, working on i=4
From worker 3:    Worker 3, working on i=5
From worker 3:    Worker 3, working on i=6
From worker 4:    Worker 4, working on i=7
From worker 4:    Worker 4, working on i=8
From worker 5:    Worker 5, working on i=9
From worker 5:    Worker 5, working on i=10
From worker 2:    Worker 2, working on i=1
From worker 2:    Worker 2, working on i=2
From worker 2:    Worker 2, working on i=3
```

Função pmap

julia -p 4

```
pmap(i -> println("Worker $(myid()), working on i=$i"), 1:10)
```

```
From worker 3:    Worker 3, working on i=4
From worker 3:    Worker 3, working on i=5
From worker 3:    Worker 3, working on i=6
From worker 4:    Worker 4, working on i=7
From worker 4:    Worker 4, working on i=8
From worker 5:    Worker 5, working on i=9
From worker 5:    Worker 5, working on i=10
From worker 2:    Worker 2, working on i=1
From worker 2:    Worker 2, working on i=2
From worker 2:    Worker 2, working on i=3
```

A função map faz a mesma coisa só que em serial

Diferença entre pmap e distributed

A macro @distributed deve ser usada para computações rápidas e a função pmap para execuções mais demoradas

Chamando as funções em baixo nível

Para chamar um código em um nó específico, basta usar:

- Macro @spawnat para rodar o código
- Função fetch() para pegar o resultado

Chamando as funções em baixo nível

```
r = @spawnat :any rand(2,2)
s = @spawnat :any 1 .+ fetch(r)
fetch(s)
```

```
2×2 Matrix{Float64}:
 1.25215  1.56702
 1.20486  1.79631
```

Executa em qualquer nó, poderíamos ter escolhido um nó específico (1,2,3,...)

EXERCÍCIO 4: MONTE CARLO

```
function darts_in_circle(N)
    sum = 0
    for i in 1:N
        if rand()^2 + rand()^2 < 1
            sum += 1
        end
    end
    return sum
end

function pi_mc(N, loops)
    darts = sum(map(x->darts_in_circle(N), 1:loops))
    return 4 * darts / (N * loops)
end
```

EXERCÍCIO 5: ANT COLONY

```
function ACO(dist_mat, ...)
    ...

    for k in 1:nants
        start_node = rand(1:n_nodes)
        end_node = start_node
        path = travel(P, start_node)
        cost = sum(edge_distances(dist_mat, path))
        push!(solutions, (cost, path))
    end

    ...
end
```

ERAD RS 2022

Programando em GPU

Minicurso 4

Computação de Alto Desempenho em Julia

Bibliotecas para GPU

- Existem diversas bibliotecas para programação em GPU: CUDA.jl, AMDGPU.jl e oneAPI.jl
- Vamos focar na utilização da biblioteca CUDA.jl pois é a utilizada para GPUs da NVIDIA
- Todas essas bibliotecas são de alto nível, então o usuário não precisa ter extenso conhecimento em GPU para usar com Julia.

SINTAXE BÁSICA - CUDA.jl

Chamada da biblioteca de paralelização:

using CUDA

A maioria das construções são feitas por macro.

CUDA.@sync begin ... end

- Exemplo:

CUDA.@sync a .+= 1

Essa operação sincroniza a expressão (a .+= 1) na GPU

Exemplo

Multiplicação de vetores

```
using CUDA
function gpu_mult!(y, x)
    CUDA.@sync y .*= x
    return
end

x_d = CUDA.fill(5, N)
y_d = CUDA.fill(7, N)
bench3 = @benchmark gpu_mult!($x_d,$y_d)
```

O símbolo $ é usado para interpolação, nesse caso. O valor dentro do $(...) é precomputado antes do benchmark, isso evita medições incorretas de tempo

EXERCÍCIO 6: VECTOR ADD

```
function vector_add!(output, input)
    for i in eachindex(output, input)
        output[i] += input[i]
    end
end
```

EXERCÍCIO 7: MANDELBROT

```
function get_steps(c::Complex, max_steps::Int64)
    z = Complex(0.0, 0.0) # 0 + 0im
    for i=1:max_steps
        z = z^2+c
        if abs2(z) >= 400
            return i
        end
    end
    return max_steps+1
end
```

ERAD | RS 2022

Computação de Alto Desempenho em Julia

Minicurso 4

Roberto M. Velho, Rafael B. Klausner, Matheus S. Serpa, Adriano M. A. Côrtes
roberto.velho@gmail.com, rafa.bench@gmail.com, msserpa@inf.ufrgs.br, adriano@nacad.ufrj.br