Desafio 15

ME315-2S2025

brotto

2025-11-20

Sumário

§ Natureza

Desafio 15

§ Resumo

Veremos a execução de um processo em paralelo no julia, bem como comparar com sua versão serial.

§ Introdução

Loops em paralelo por vezes ajudam a calcular operações de forma mais veloz. A seguir, estão resultados de uma análise de performance entre computação serial e paralela. O problema é a estimação de π através do método de monte carlo — um processo iterativo (utiliza loops).

Julia - loops em paralelo

```
   julia_setup(JULIA_HOME = NULL)
```

```
# Pkg
import Pkg
```

```
# status
Pkg.status()
```

```
## Status `C:\Users\User\.julia\environments\v1.12\Project.toml`
##   [6e4b80f9] BenchmarkTools v1.6.3
##   [429524aa] Optim v1.13.2
##   [91a5bcdd] Plots v1.41.1
##   [6f49c342] RCall v0.14.9
##   [10745b16] Statistics v1.11.1
##   [fd094767] Suppressor v0.2.8
##   [8ba89e20] Distributed v1.11.0
## Info Packages marked with  have new versions available and may be upgradable.
```

```
# baixa
using Pkg
Pkg.add("Distributed")
Pkg.add("Statistics")
Pkg.add("BenchmarkTools")

# carrega
```

```
using Distributed, Statistics, BenchmarkTools
```

```
# serial
function monte_carlo_serial(n)
    dentro = 0
    for i in 1:n
        x, y = rand(), rand()
        if x^2 + y^2 <= 1.0
            dentro += 1
        end
    end
    return 4.0 * dentro / n
end
```

```
## monte_carlo_serial (generic function with 1 method)
```

```
# @distributed
function monte_carlo_distributed(n)
    dentro = @distributed (+) for i in 1:n
        x, y = rand(), rand()
        Int(x^2 + y^2 <= 1.0)
    end
    return 4.0 * dentro / n
end
```

```
## monte_carlo_distributed (generic function with 1 method)
```

```
# testes
function teste_distributed()
    n = 10_000_000

    tempo_serial = @elapsed pi_serial = monte_carlo_serial(n)
    tempo_parallel = @elapsed pi_parallel = monte_carlo_distributed(n)

    speedup = tempo_serial / tempo_parallel

    erro_serial = abs(pi_serial -  )
    erro_parallel = abs(pi_parallel -  )

    diferenca = abs(erro_serial - erro_parallel)

    return pi_serial, pi_parallel, tempo_serial, tempo_parallel, speedup, diferenca,
     ↪  erro_serial
end
```

```
## teste_distributed (generic function with 1 method)
```

```
nprocs() # n de processos
```

```
## 1
```

```
JuliaCall:::.julia$cmd("(nprocs() == 1) ? addprocs(3) : FALSE")
```

```
nprocs() # n de processos
```

```
## 4
```

```
# pi_serial, pi_parallel, tempo_serial, tempo_parallel, speedup, diferenca, erro_serial
resultado = teste_distributed() # testa
```

```
## (3.1415292, 3.1416568, 0.0446199, 2.8104146, 0.01587662546301887, 6.928204134837301e-
7, 6.345358979320537e-5)
```

Ganho em peformance:

$$0.0158766\text{x} \quad \text{(o “ganho”na verdade é uma perda, já que } \texttt{speedup} \leq 1\text{x)}$$

```
## O resultado gerado pela compilação do knitr é mais lento, mas o tempo de compilação de
↪  fato é mais veloz usando o processamento paralelo.
```

```
(3.1416076, 3.1412776, 0.0422801, 0.0206611, 2.0463624879604665, 0.0003001071795862842, 1.4946410206828631e-5)
```

Figura 1: Saída indicando melhora de aprox. 2.04x no tempo de execução.

Notas:

```
## O @distributed com (+) faz automaticamente:
## 1. Divide o loop 1:n entre workers
## 2. Cada worker calcula sua soma parcial
## 3. Combina todas as somas parciais com +
   dentro = @distributed (+) for i in 1:n
       # Retorna 1 ou 0 para cada ponto
       Int(rand()^2 + rand()^2 <= 1.0)
   end
```