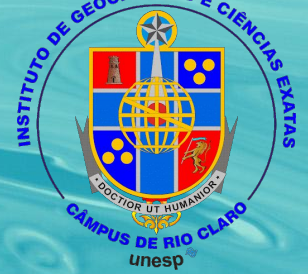
# Paralelismo em Linguagens Modernas de Programação

**Lucas Bagatini do Nascimento**

Orientador:
Prof. Dr. Alexandro José Baldassin




unesp

# Sumário

# 01

# INTRODUÇÃO

## OBJETIVOS

Comparação quantitativa e qualitativa entre abordagens de 4 linguagens de programação.

## LEI DE MOORE

A cada 2 anos a densidade de transistores de um processador dobra.

Paralelismo como alternativa!

## EXPERIMENTO EM 2 ETAPAS

**1)** Dois problemas trivialmente paralelizáveis foram resolvidos por cada uma das linguagens
**2)** Traçar um perfil da eficiência e ganho de desempenho e análise qualitativa

# Sumário

02

# LEI DE MOORE

E seus limites

# LEI DE MOORE

"Densidade de transistores dobra a cada dois anos"

- Descritiva, não preditiva
- Tentativa de acompanhar demanda aumentando potência
- Limite superior: calor, energia, tunelamento
- Alternativa 1: inovação tecnológica (cada vez mais desafiador)
- **Alternativa 2: paralelismo.**

## CONCORRÊNCIA

"(...) trabalho sobre múltiplas tarefas por um mesmo agente, alternadamente."

## PARALELISMO

"(...) simultaneidade dos passos das diversas tarefas
por diversos agentes."

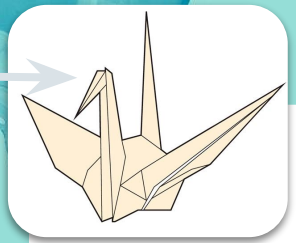
## CONCORRÊNCIA

"(...) trabalho sobre múltiplas tarefas por um mesmo agente, alternadamente."


## PARALELISMO

"(...) simultaneidade dos passos das diversas tarefas
por diversos agentes."

# CONTEXTUALIZAÇÃO TEÓRICA

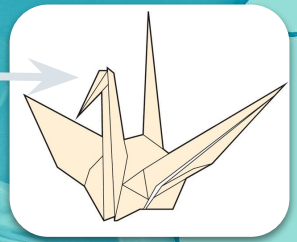



## CONCORRÊ

"(...) trabalho sobre múltiplas tarefas por um mesmo agente, alternadamente."

## PARALELISMO

"(...) simultaneidade dos passos das diversas tarefas
por diversos agentes."

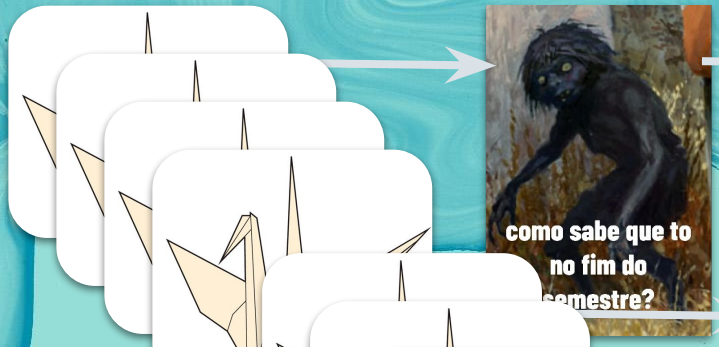



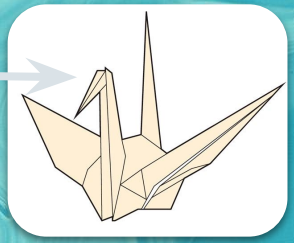
## CONCORRÊNCIA

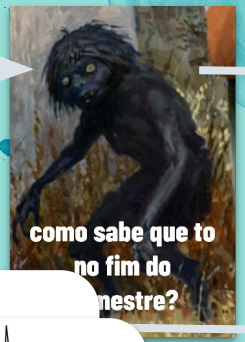
"(...) trabalho sobre múltiplas tarefas por um mesmo agente, alternadamente."

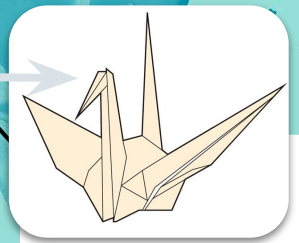
## PARALELISMO

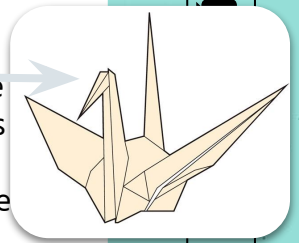
"(...) simultaneidade dos passos das diversas tarefas
por diversos agentes."





## CONCORRE

"(...) trabalho sobre múltiplas tarefas por um mesmo agente, alternadamente."

## LISMO

itaneidade
as diversas

rsos agente

CONTEXTUALIZAÇ

## CONCORRÊNCIA

"(...) trabalho sobre múltiplas tarefas por um mesmo agente, alternadamente."

## PARALELISMO

"(...) simultaneidade dos passos das diversas tarefas
por diversos agentes."

# Sumário

# 03

# PROBLEMAS CLÁSSICOS

De sincronização e paralelização

# PROBLEMAS CLÁSSICOS

Condição de corrida (seções críticas)

- 2 *Threads* acessando o mesmo recurso.
- Não há garantia de quem acessa primeiro.
- Resultado corrompido!

# PR LÁSSIC


rida (seções críticas)
cessando o mesmo rec
há garantia de quem acessa p
ltado corrompido!

# PROBLEMAS CLÁSSICOS

Condição de corrida (seções críticas)

- 2 *Threads* acessando o mesmo recurso.
- Não há garantia de quem acessa primeiro.
- Resultado corrompido!
- **Solução: Atomicidade**
  - Mutexes
  - Wait

# PROBLEMAS CLÁSSICOS

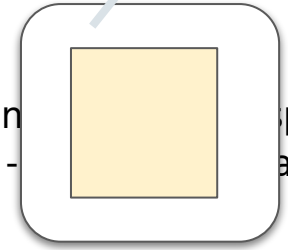
# PROBLEMAS CLÁSSICOS

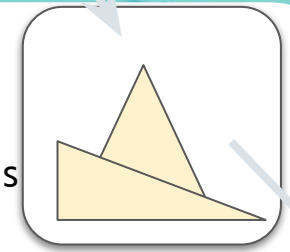
Sincronização e esperas

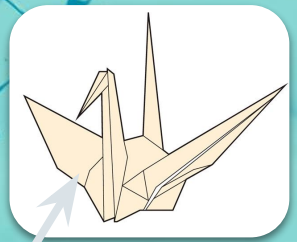
- Threads aguardando sinais de outras threads


# PROBLEMAS CLÁSSICOS

Sin speras
- ardando sinais







# PROBLEMAS CLÁSSICOS

Sincronização e esperas

- Threads aguardando sinais de outras threads

# Sumário

# 04

# MODELOS DE PARALELISMO

XXXX

# MODELOS DE PARALELISMO

**Troca de Mensagens**

- ➔ Canais de comunicação (*channel* em Go)
- ➔ Operações atômicas **envio-recebimento.**
  - ◆ recebimento é bloqueante em Go.
- ➔ Garante segurança ao segregar os dados
- ➔ Requer *retrofit* de grande parte do algoritmo

# MODELOS DE PARALELISMO

# MODELOS DE PARALELISMO

**Promessas e Objetos Futuros**

- ➔ Abstração
- ➔ Computação delegada a uma thread
- ➔ Valor é computado em *background* e recuperado quando necessário (promessa de conclusão).
  - ◆ Similar a um join...

# MODELOS DE PARALELISMO

# MODELOS DE PARALELISMO

**Threads Verdes e Threads Verdadeiras**

- ➔ Threads têm seus próprios contextos (stacks, variáveis locais, etc)
- ➔ OS Threads (**threads verdadeiras) são custosas** para construir, destruir e em memória
  - ◆ Criadas em *userspace*
  - ◆ Escalonadas em *userspace*
- ➔ Threads verdes são uma abstração:
  - ◆ Criadas em *userspace*
  - ◆ Escalonadas em *userspace*
  - ◆ Muito mais leves!

# MODELOS DE PARALELISMO

# MODELOS DE PARALELISMO

**Thread Pooling**

- ➔ OS Threads são custosas.
- ➔ Em suma:
  - ◆ criar threads uma única vez
  - ◆ Delegar trabalho para a **Reserva de Threads**
  - ◆ Manter threads vivas, em estase, para o futuro

# MODELOS DE PARALELISMO

# MODELOS DE PARALELISMO

**Corrotinas**

- ➔ Abstração para computação concorrente
  - ◆ Rotinas que podem ser interrompidas (troca de contexto).

# Sumário

05

# LINGUAGENS UTILIZADAS

E suas principais características

# LINGUAGENS UTILIZADAS

## 01

### PYTHON

Interpretada
Multi-paradigma
Orientada a Objeto
Tipagem Dinâmica
Inferência de Tipos
Coleta de Lixo

## 02

### GO

Compilada
Estruturada
Concorrente
Tipagem Estática
Inferência de Tipos
Coleta de Lixo

## 03

### RUST

Compilada
Estruturada
Linguagem de Sist.
Tipagem Estática
Inferência de Tipos
Borrow-Checker

## 04

### KOTLIN

Compilada para JVM
Orientada a Objeto
Concorrente
Tipagem Estátiac
Inferência de Tipos
Coleta de Lixo (JVM)

## GLOBAL INTERPRETER LOCK

Trava do interpretador, permite somente uma thread operar por vez (libera em operações de IO)

## POOLING

Reserva de threads, para as quais os *workloads* são despachados. Implementa Thread Pooling e Process Pooling

## FUTURES

Abstração, implementação de Objetos Futuros

PYTHON

## GO ROUTINES

Go Routines são basicamente corrotinas automaticamente despachadas para threads verdes.

## THREADS VERDES

Go espera que muitas sejam inicializadas através das Go Routines.

## CANAIS

Go possui grande preferência por comunicação entre threads via mensagens. Canais são fáceis de usar e a leitura é bloqueante.

GO

## BORROW-CHECKER

Gerenciamento de memória baseado em *ownership*. Donos únicos -> segurança.

## ARCS

Atomically-referenced counters. Envelopam variáveis e tornam as operações sobre elas atômicas.

## CLOSURES

Funções in-line, extremamente convenientes.

RUST

## KOTLIN E JAVA

Mercury is the closest planet to the Sun and the smallest one in our Solar System

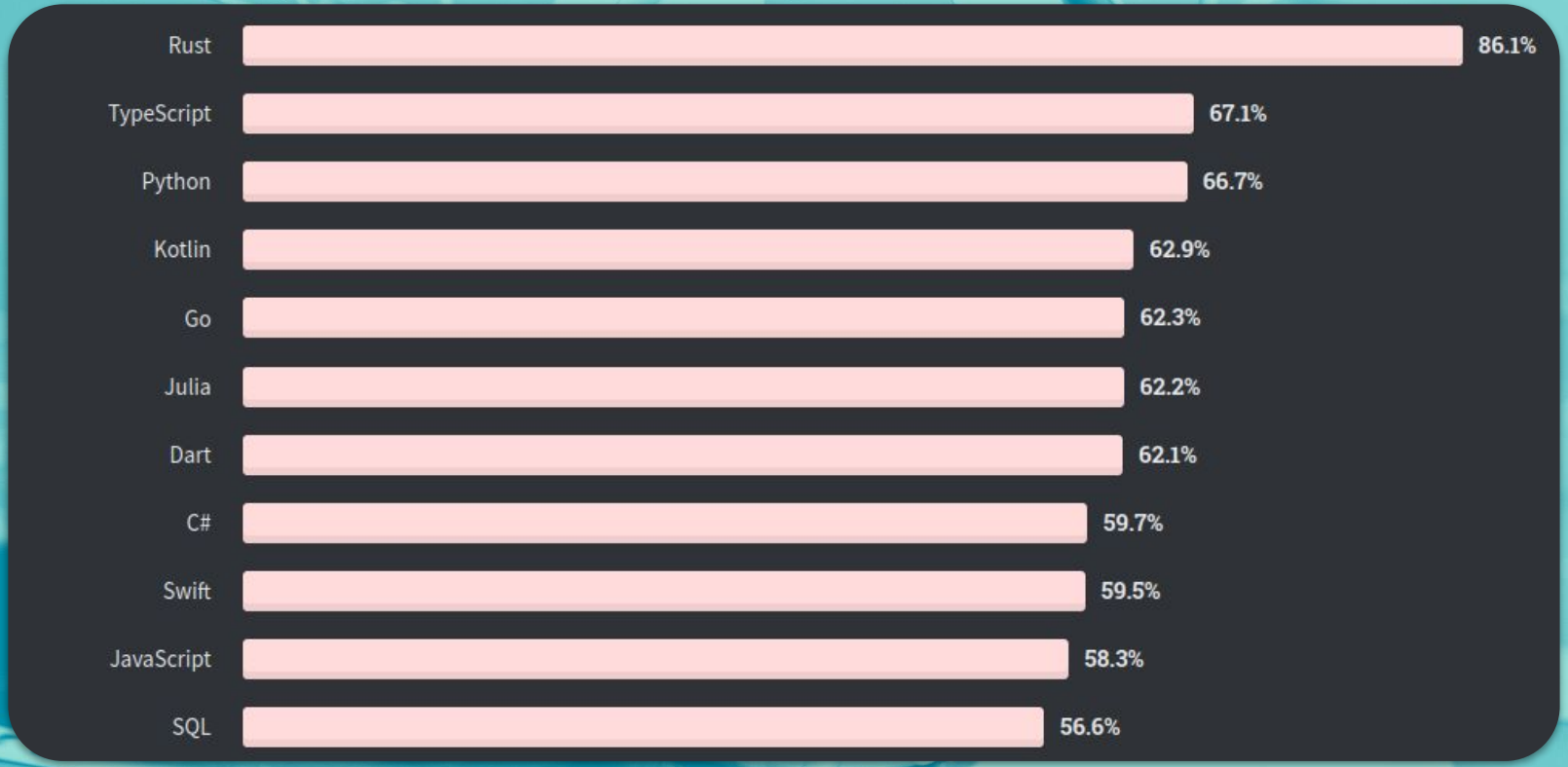
## CORROTINAS

Saturn is the ringed one. It’s a gas giant, composed mostly of hydrogen and helium

## DISPATCHERS

Similar a thread pooling. Trata-se de threads especializadas em certos tipos de tarefas, para as quais os *workloads* são despachados

KOTLIN




Developer Survey 2020, StackOverflow.  Most Loved Programming Languages

# Sumário

# 06

# EXPERIMENTOS

E resultados

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

## CPU-BOUND

Gargalo são as operações no processador.

Computação do número pi através da Série de Leibniz com 1 bilhão de iterações.

## MEMORY-BOUND

Gargalo são as operações de leitura-escrita.

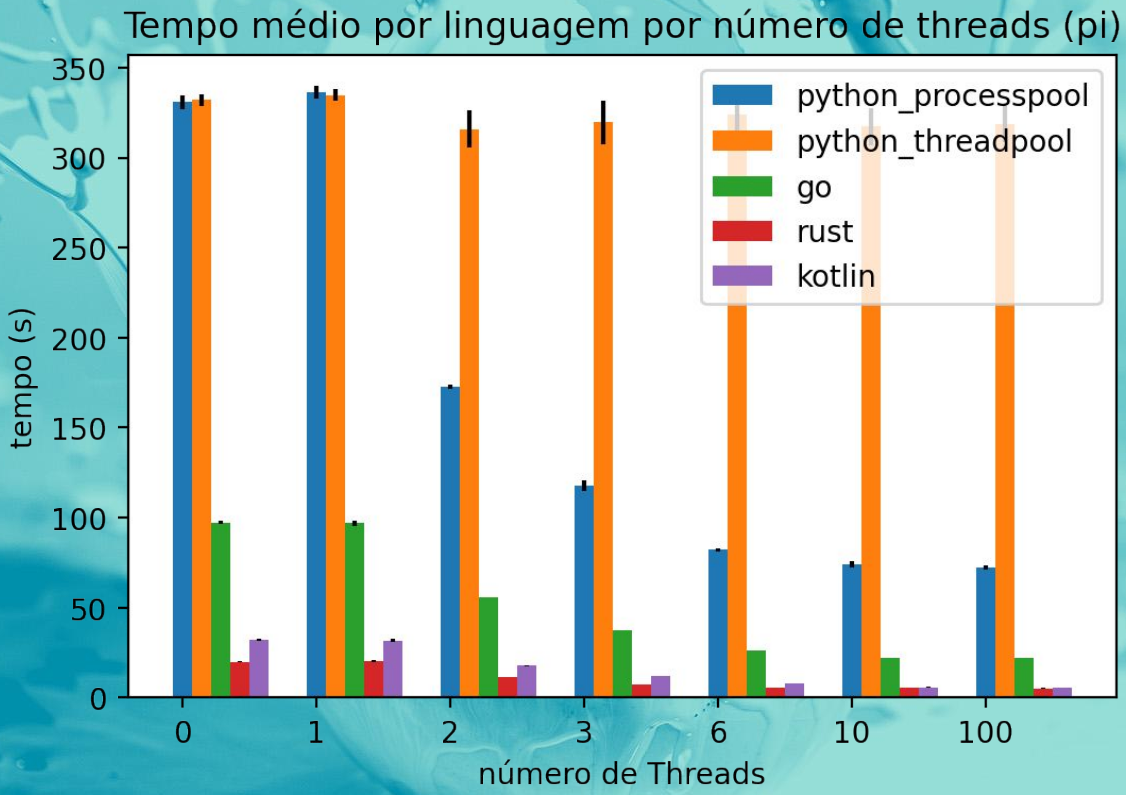
Multiplicação de duas matrizes 1000x1000.

# EXPECTATIVAS

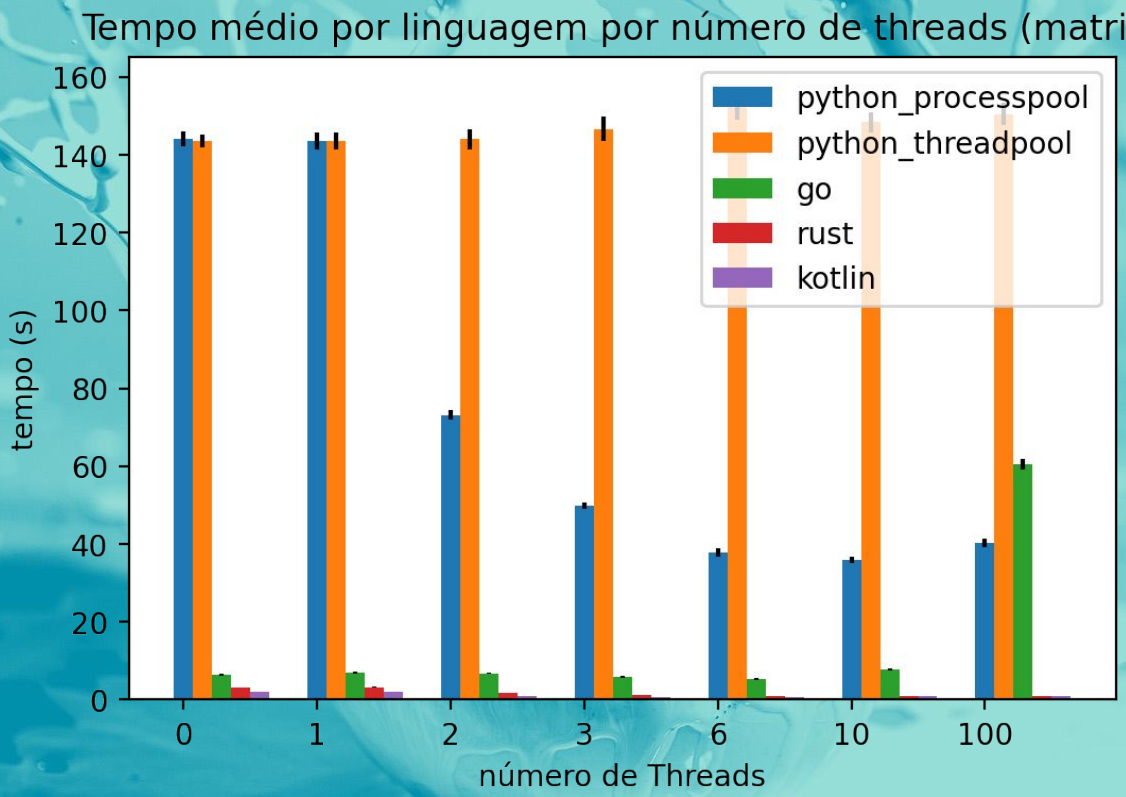
- Rust seria a mais veloz
  - Rust seria o mais trabalhoso
- Python seria o mais lento
  - Python seria o mais ergonômico
- 1 Thread é mais lento que sequencial (devido ao overhead)
- Até 6 threads há speedup
  - O Speedup entre 10 e 100 Threads não existe ou é negativo

# EXPECTATIVAS

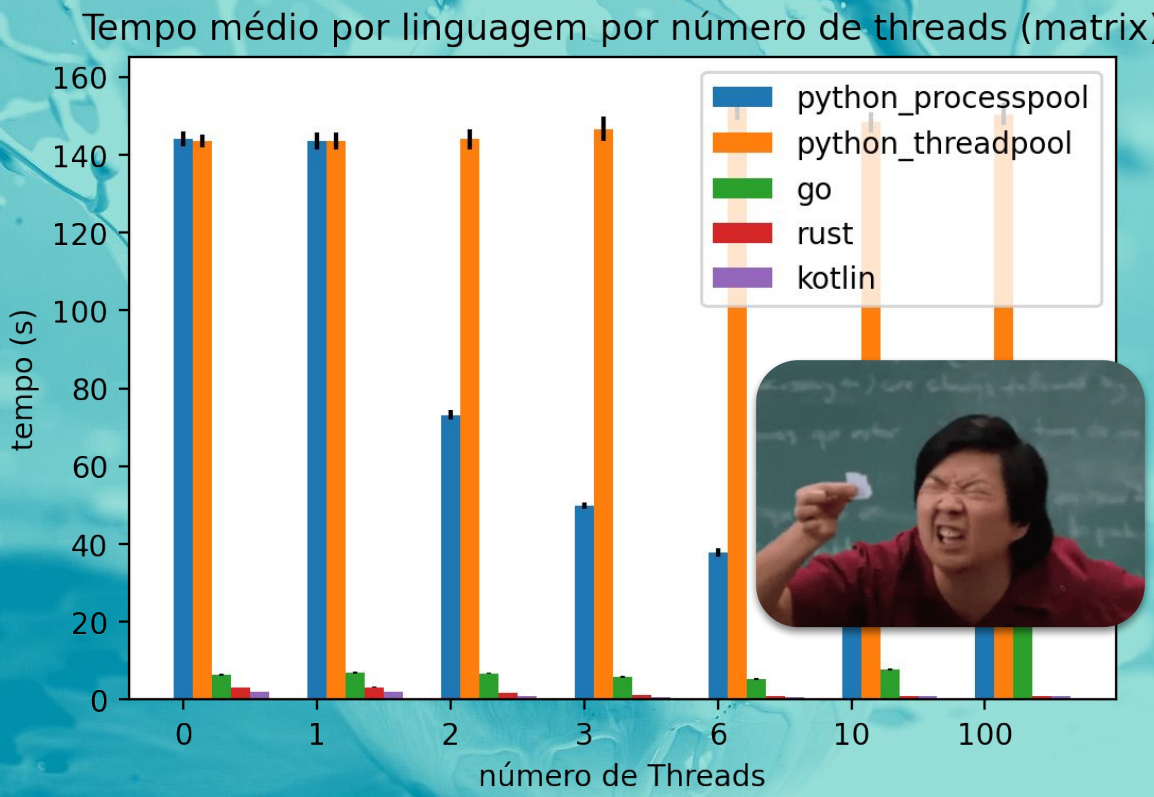
- ➔ Rust seria a mais veloz
  - ◆ Rust seria o mais trabalhoso ✅
- ➔ Python seria o mais lento
  - ◆ Python seria o mais ergonômico❌ - **Go levou o prêmio**
- ➔ 1 Thread é mais lento que sequencial (devido ao overhead)
- ➔ Até 6 threads há speedup
  - ◆ O Speedup entre 10 e 100 Threads não existe ou é negativo



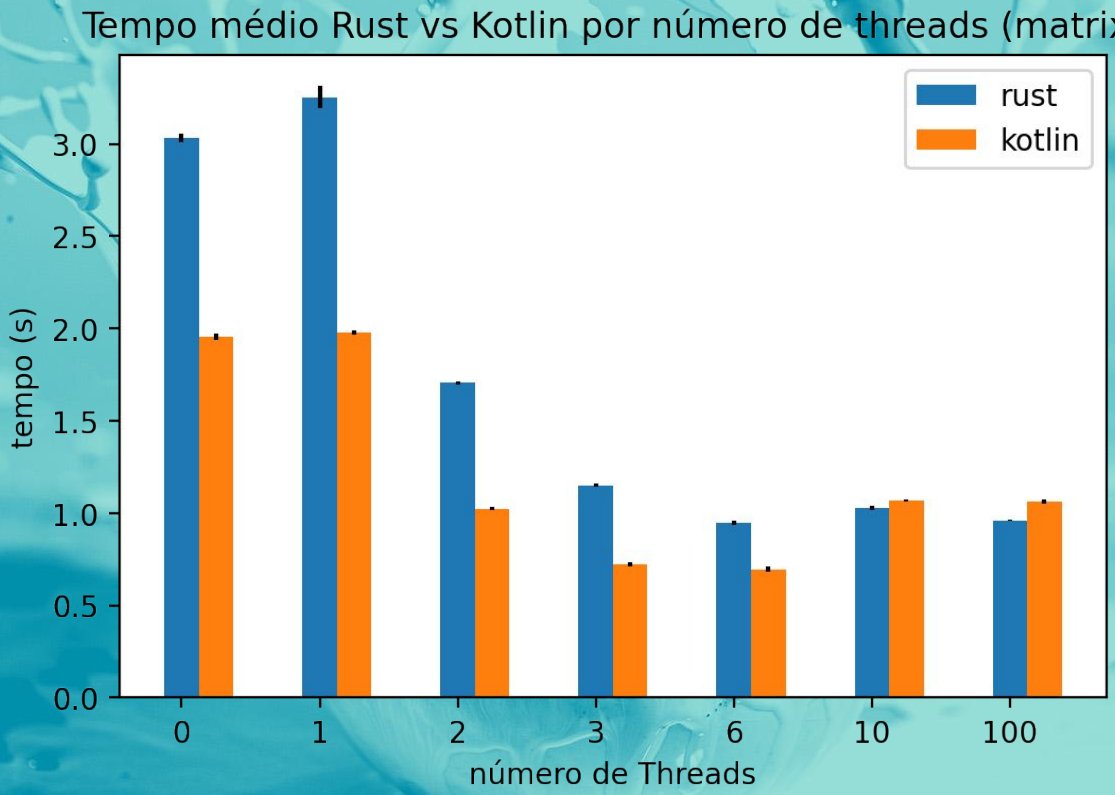

Tempo médio, cálculo do número pi




Tempo médio, multiplicação de matrizes



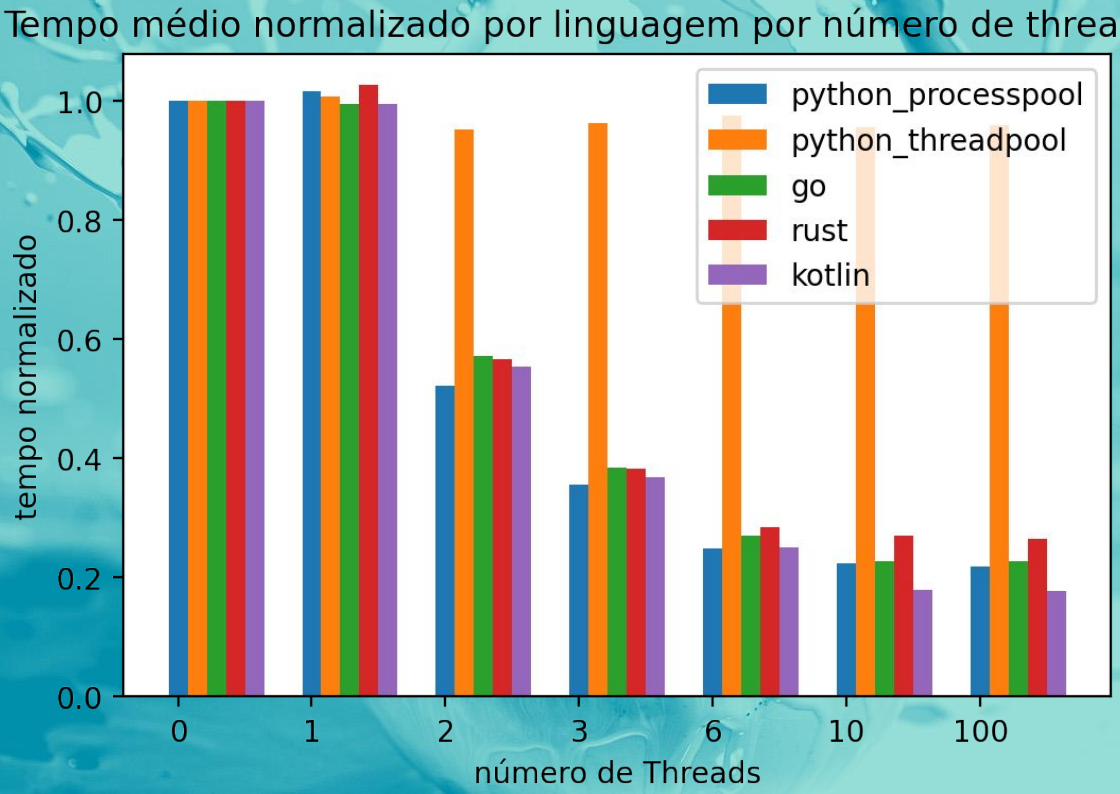

Tempo médio, multiplicação de matrizes



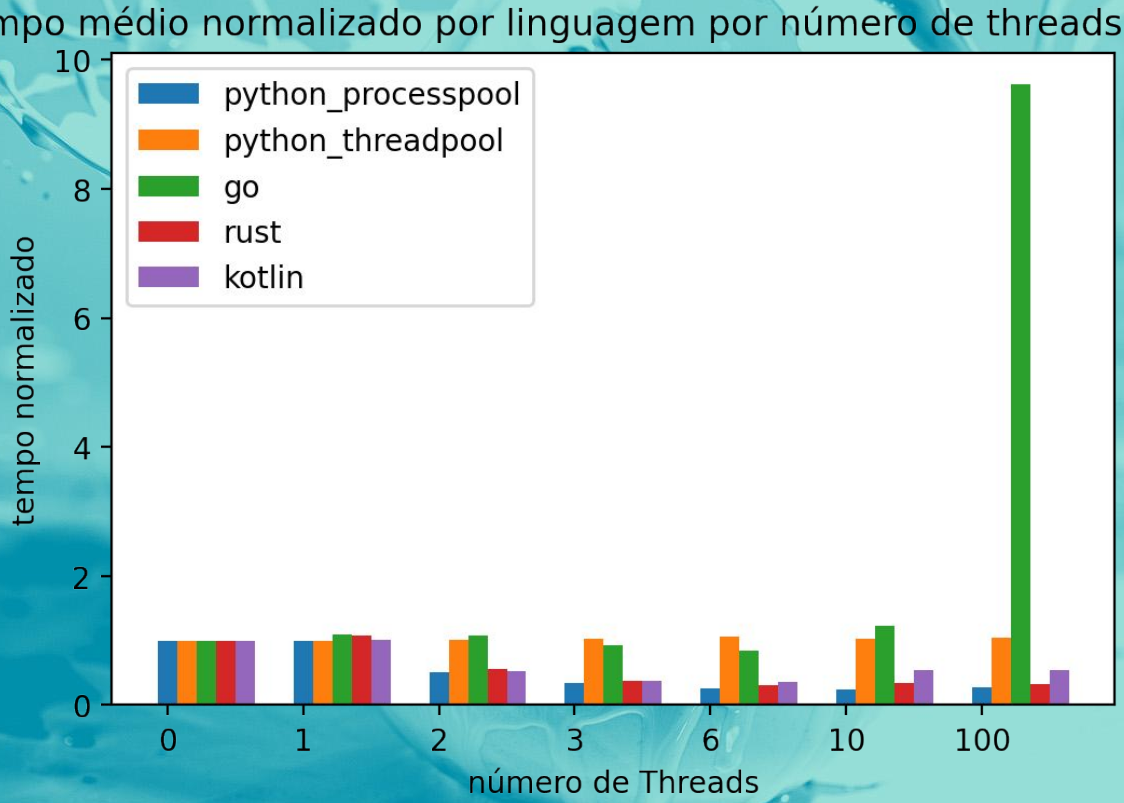

Tempo médio, multiplicação de matrizes, Rust e Kotlin somente.

# EXPECTATIVAS

- ➔ Rust seria a mais veloz ✅❌ **Kotlin obteve um desempenho pouco melhor na multiplicação de matrizes.**
  - ◆ Rust seria o mais trabalhoso ✅
- ➔ Python seria o mais lento ✅
  - ◆ Python seria o mais ergonômico❌ - **Go levou o prêmio**
- ➔ 1 Thread é mais lento que sequencial (devido ao overhead)
- ➔ Até 6 threads há speedup
  - ◆ O Speedup entre 10 e 100 Threads não existe ou é negativo

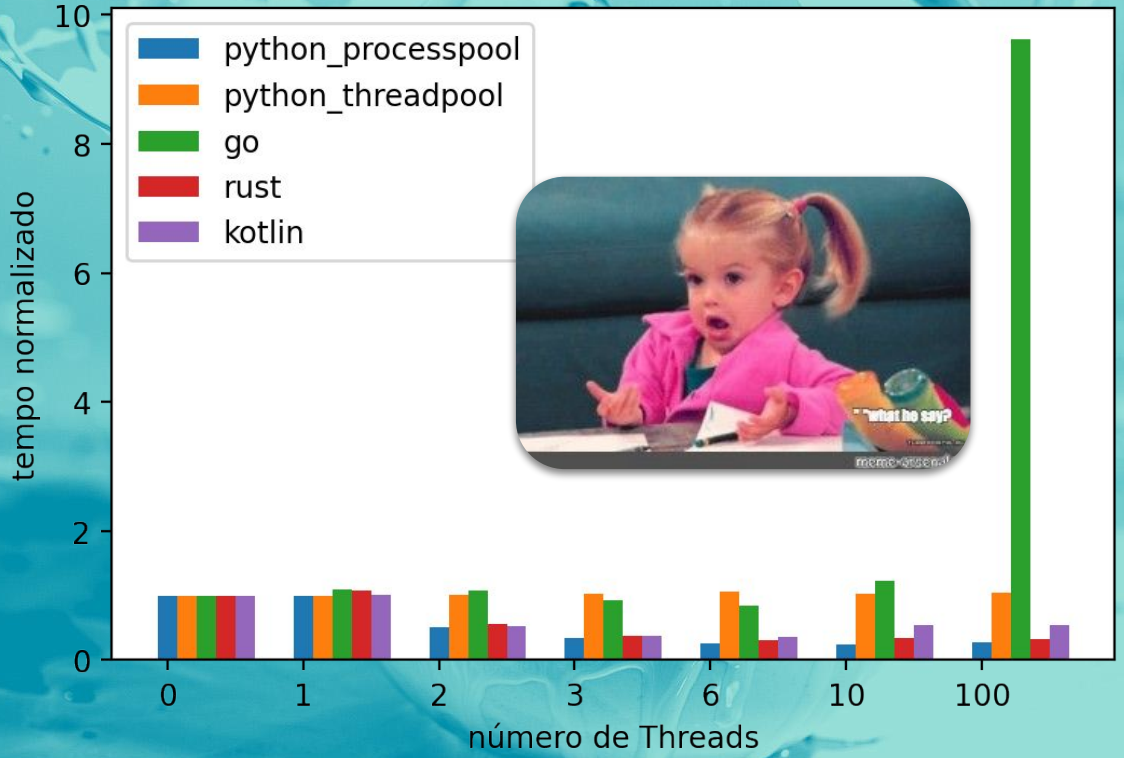



Tempo médio normalizado,  cálculo do número pi



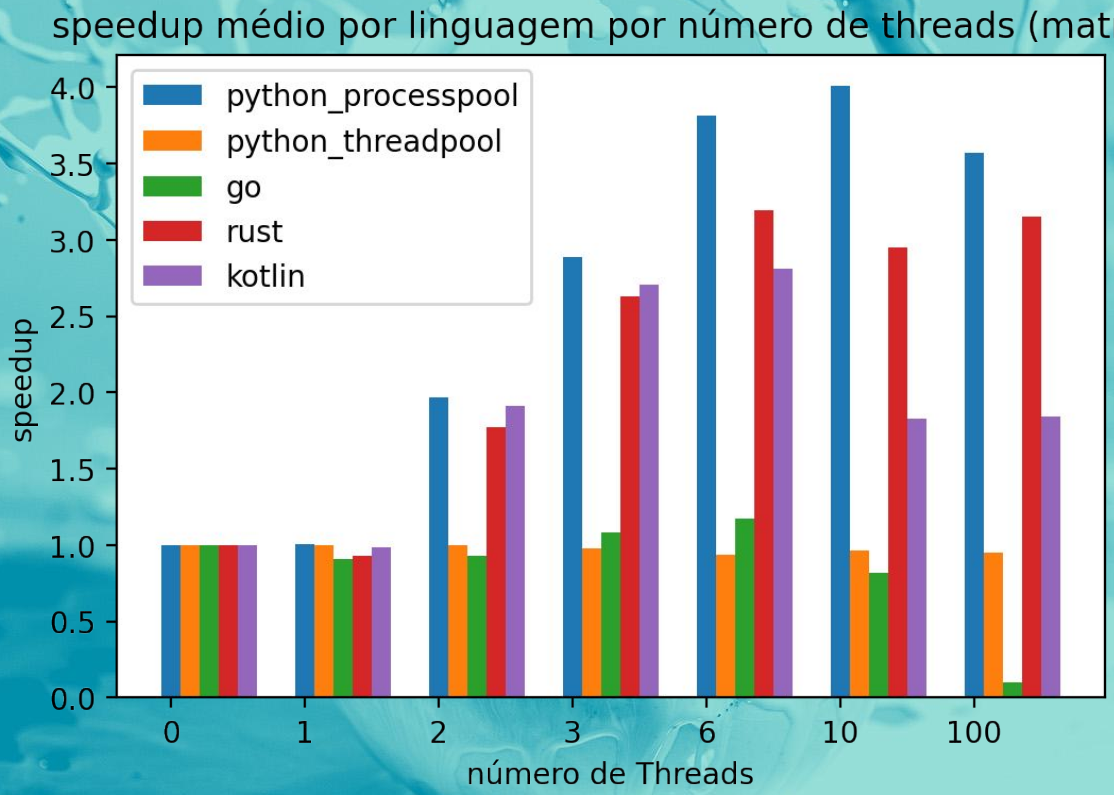

Tempo médio normalizado, multiplicação de matrizes

Tempo médio normalizado por linguagem por número de threads (matrix)

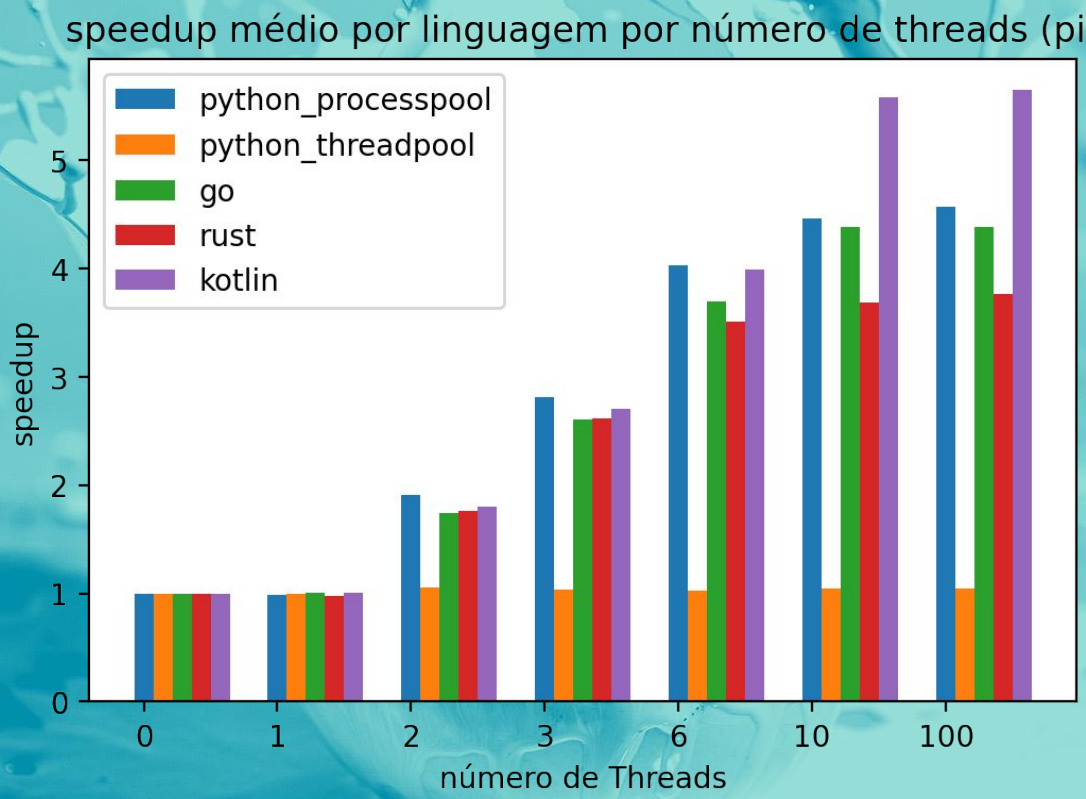



Overhead??

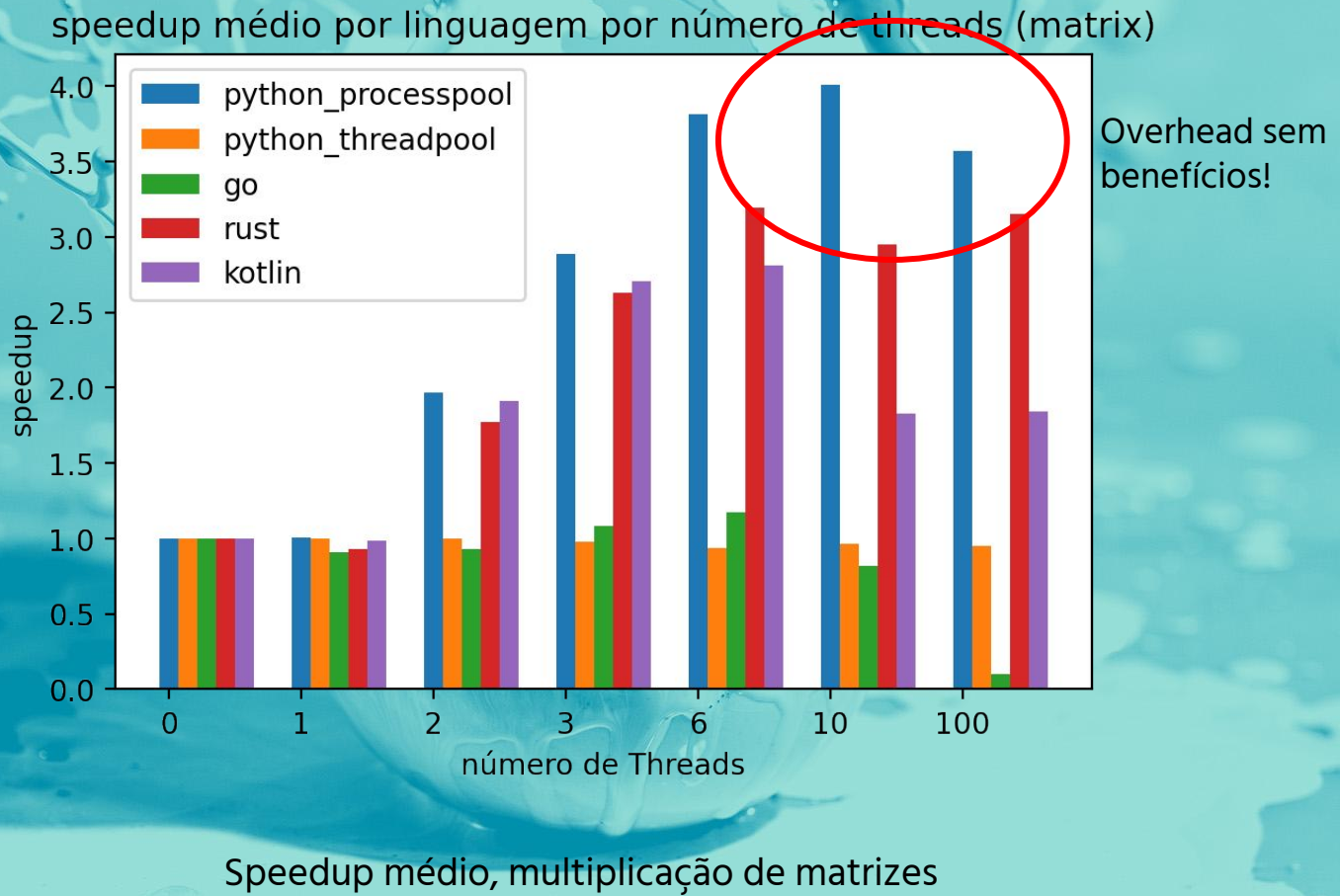
Tempo médio normalizado, multiplicação de matrizes




Speedup médio, multiplicação de matrizes




Speedup médio,  cálculo do número pi

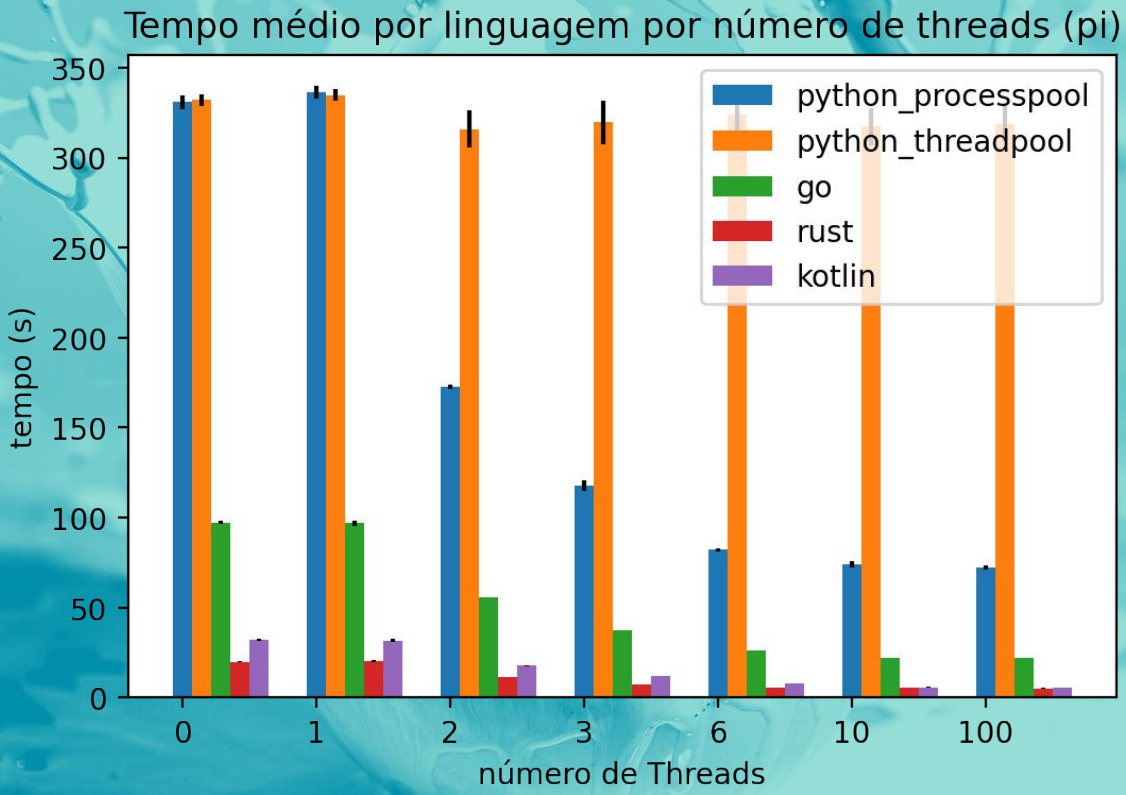



Speedup médio, multiplicação de matrizes

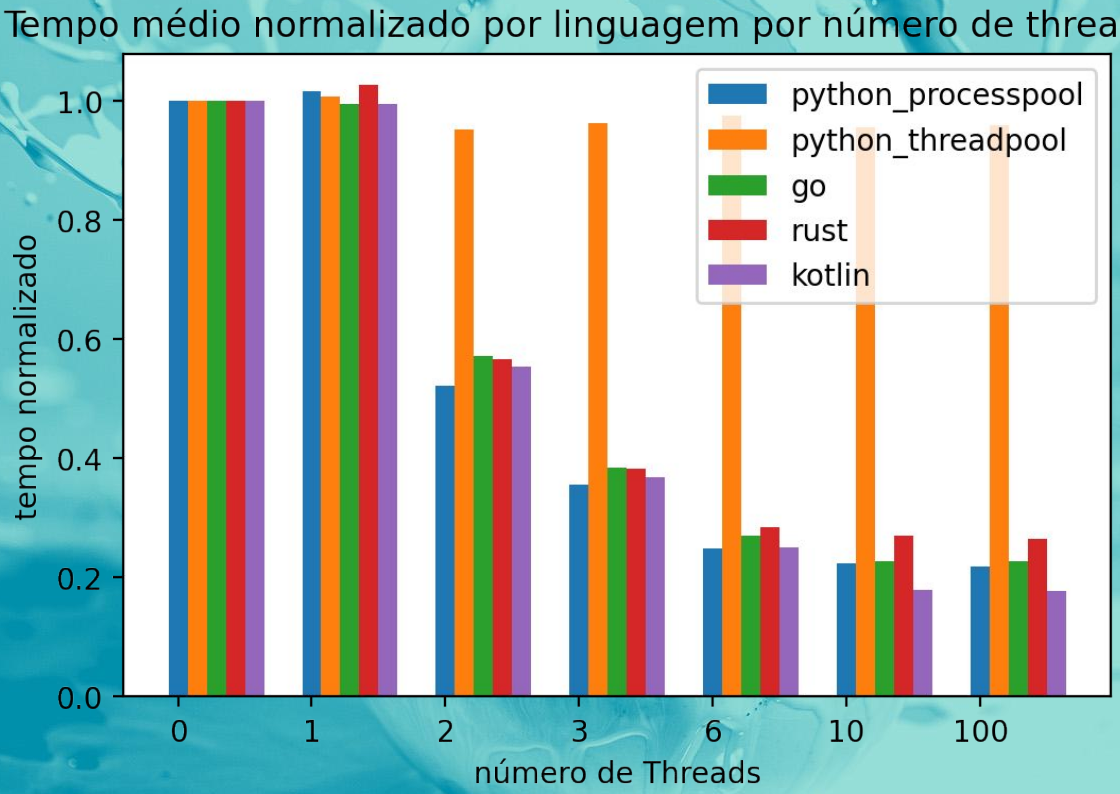
# EXPECTATIVAS

- ➔ Rust seria a mais veloz ✅❌ **Kotlin obteve um desempenho pouco melhor na multiplicação de matrizes.**
  - ◆ Rust seria o mais trabalhoso ✅
- ➔ Python seria o mais lento ✅
  - ◆ Python seria o mais ergonômico❌ - **Go levou o prêmio**
- ➔ 1 Thread é mais lento que sequencial (devido ao overhead)
- ➔ Até 6 threads há speedup ✅
  - ◆ O Speedup entre 10 e 100 Threads não existe ou é negativo ✅

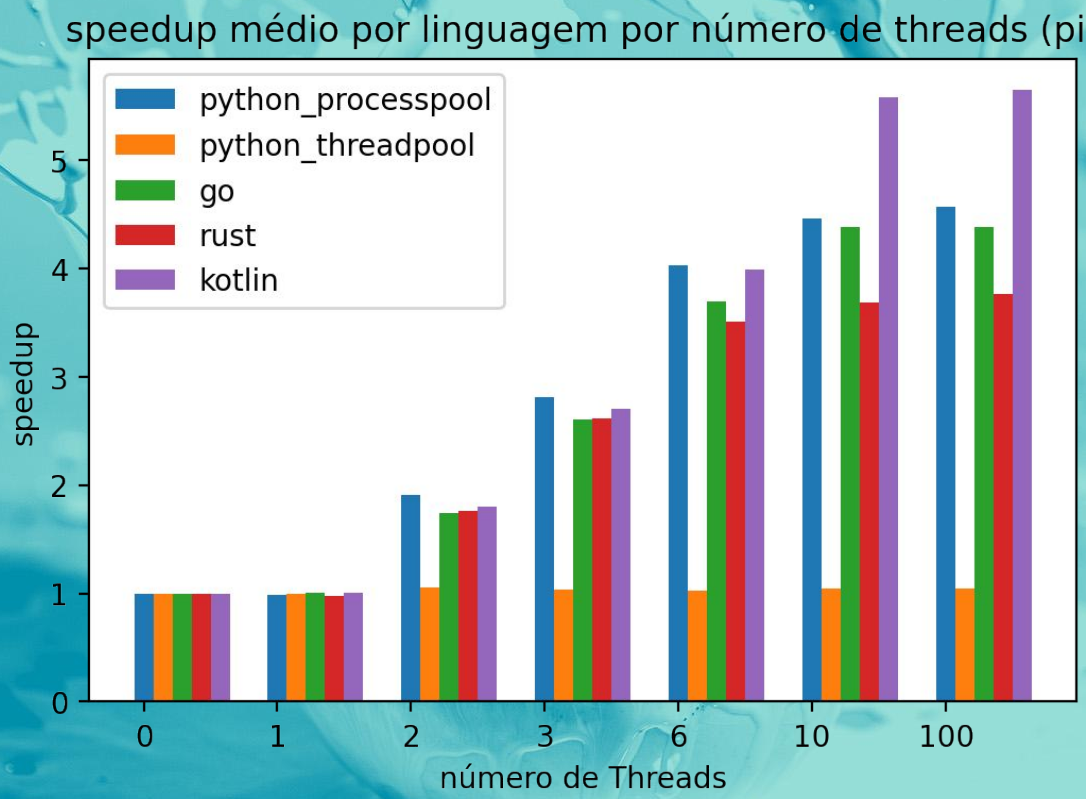
# Todos os Gráficos:



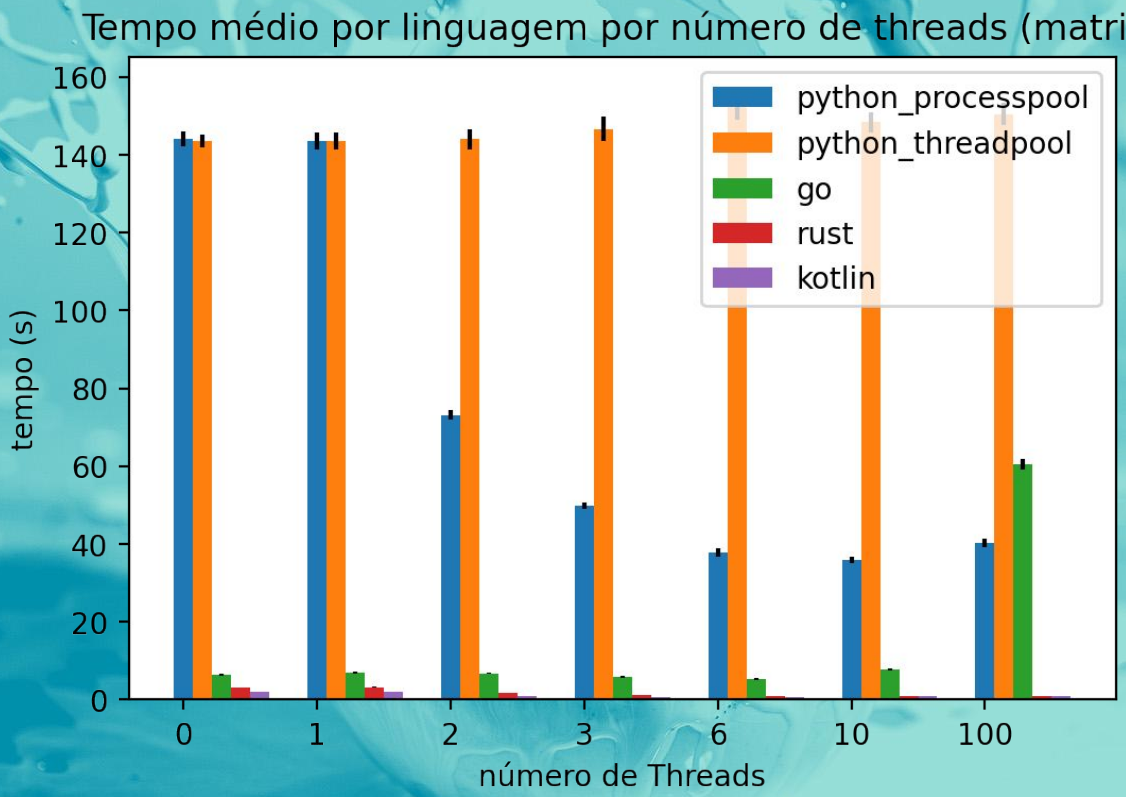

Tempo médio, cálculo do número pi



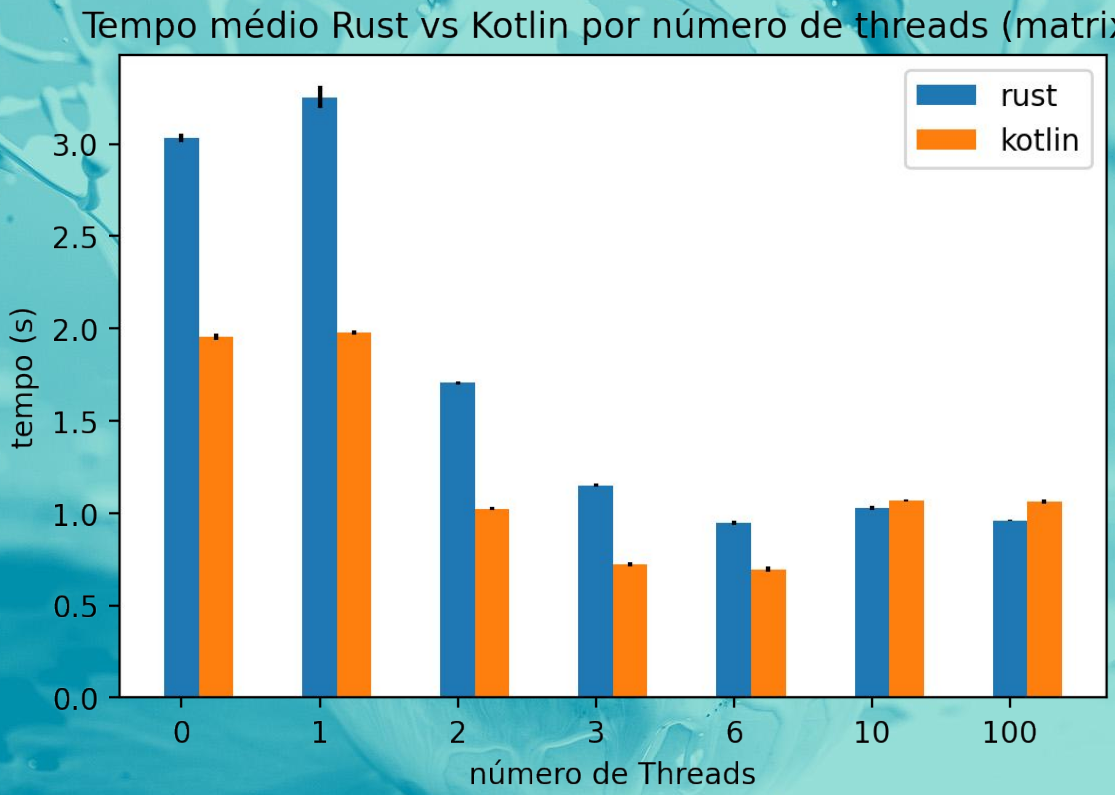

Tempo médio normalizado,  cálculo do número pi



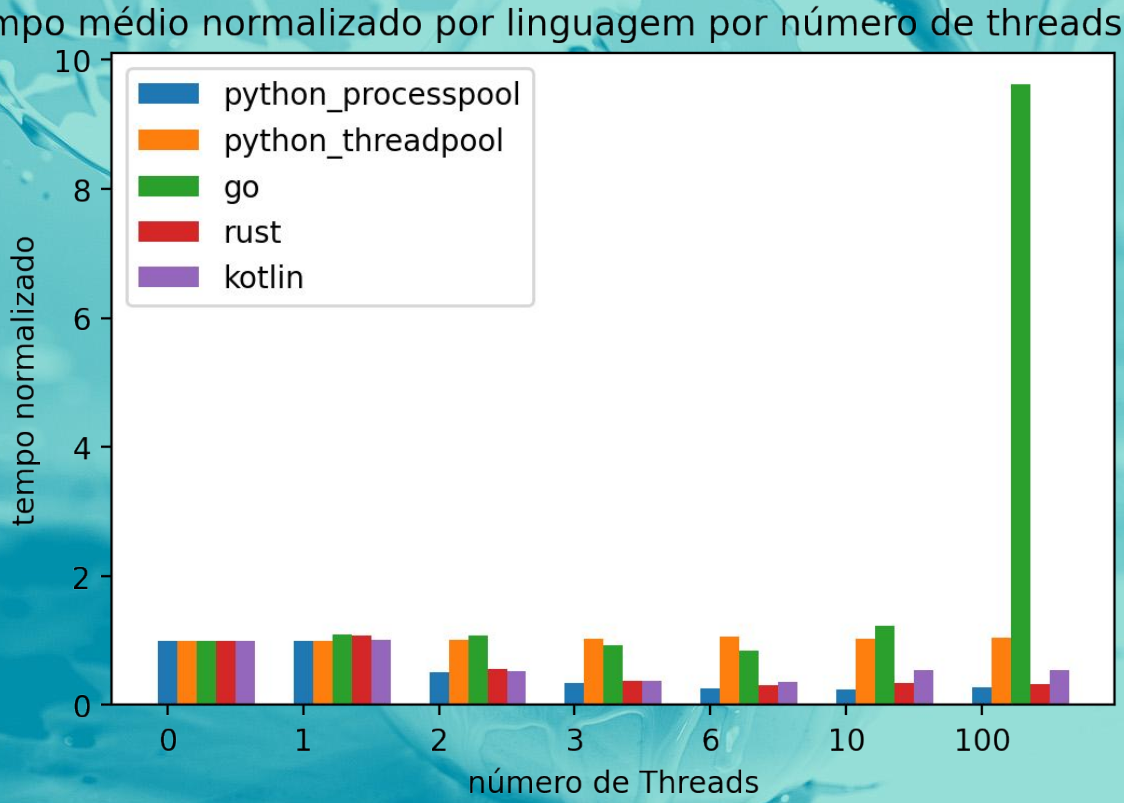

Speedup médio,  cálculo do número pi



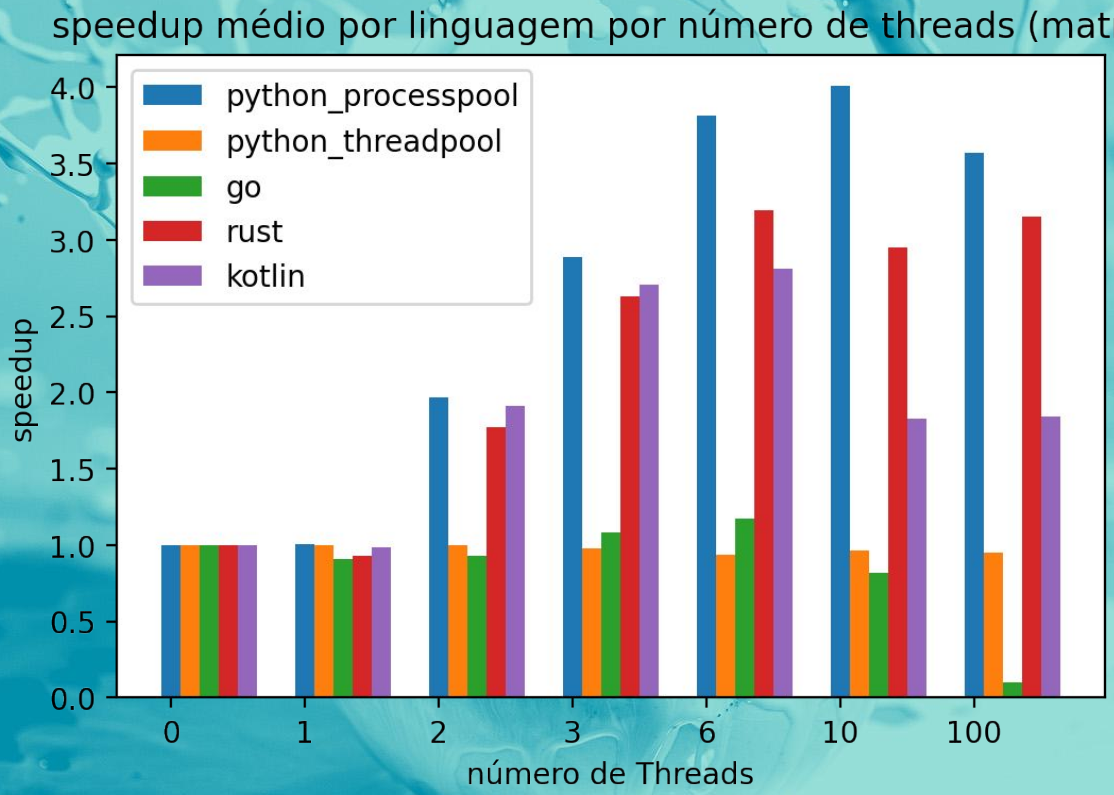

Tempo médio, multiplicação de matrizes




Tempo médio, multiplicação de matrizes, Rust e Kotlin somente.




Tempo médio normalizado, multiplicação de matrizes




Speedup médio, multiplicação de matrizes

# Sumário

# CONCLUSÃO

- **Linguagens competentes**: entregaram o que se pretendiam.

- **Futuro promissor**: linguagens tornando paralelismo acessível e seguro.

# Obrigado pela Atenção!

Lucas Bagatini do Nascimento
lucas.bagatini@unesp.br