INSTITUTO FEDERAL DE MATO GROSSO DO SUL

CAMPUS TRÊS LAGOAS – MS

CLEITON DA SILVA GUILHERMITE

LEONARDO ARAUJO ARMELIN

LUIZ FELIPE MIRANDA

STEFANY FIGUEIREDO

THREADS E PROCESSOS

TRÊS LAGOAS – MS

2024

CLEITON DA SILVA GUILHERMITE

LEONARDO ARAUJO ARMELIN

LUIZ FELIPE MIRANDA

STEFANY FIGUEIREDO

THREADS E PROCESSOS

Relatório apresentado ao curso de Engenharia de Computação, do Instituto Federal de Mato Grosso do Sul, campus Três Lagoas, como requisito para avaliação da Unidade Curricular Sistemas Operacionais.

Professor: Marco Aurélio Ferreira

TRÊS LAGOAS – MS

2024

INSTITUTO FEDERAL DE MATO GROSSO DO SUL

CAMPUS TRÊS LAGOAS – MS

BACHARELADO EM ENG. COMPUTAÇÃO

Sistemas Operacionais

Relatório nº01

Threads e processos – uma aplicação para a análise de desempenho

17/12/2024

Nome completo	RA	Assinatura
Cleiton da Silva Guilhermite	049565	
Leonardo Araújo Armelin	049577	
Luiz Felipe Miranda	049581	
Stefany P. Figueiredo da Silva	049585	

Professor(a): Marco Aurélio Ferreira

RESUMO

O presente relatório visa a documentação da implementação de um código a fim de analisar o desempenho de threads e processos em um cenário com processamento CPU-Bound e o impacto do Global Interpreter Lock (GIL) do Python e as limitações impostas por troca de contexto e escalabilidade. É realizada uma série de testes para comparar o tempo de execução de abordagens (estruturas) baseadas threads e processos em um algoritmo responsável por operações intensivas de CPU (cálculo de fatoriais) em 10^6 iterações, testado em sistemas Windows e Linux, variando o número de threads e processos. Os resultados mostram que threads são mais eficientes em ambientes CPU-bound limitados pelo GIL, enquanto processos perdem eficiência em maior escala. A análise confirma que o uso de threads ou processos deve considerar o tipo de tarefa e as características do ambiente computacional.

Palavras-chave: Threads. Processos. GIL. Escalabilidade. CPU-Bound. Desempenho.

SUMÁRIO

1. INTRODUÇÃO

A arquitetura moderna de sistemas computacionais busca constantemente otimizar a utilização dos recursos disponíveis, como a unidade central de processamento (CPU), a memória e os dispositivos de entrada e saída (E/S). Nesse contexto, os conceitos de processos e threads surgem como pilares fundamentais para a compreensão e melhoria do desempenho de aplicações e sistemas operacionais. Este trabalho tem como objetivo explorar esses conceitos, abordando suas características, relações intrínsecas com a CPU, impacto nas operações de E/S e as vantagens e desvantagens associadas a cada abordagem de processamento.

Um processo pode ser entendido como uma instância de um programa em execução, possuindo seu próprio espaço de endereçamento, recursos alocados e contexto de execução. Por outro lado, uma thread é uma unidade de execução menor dentro de um processo, compartilhando recursos como memória e arquivos abertos com outras threads do mesmo processo. Essa distinção é essencial para compreender como as threads podem oferecer uma abordagem mais leve e eficiente para a execução concorrente de tarefas, enquanto os processos garantem maior isolamento e segurança, cada uma com seu ambiente de teste.

A relação entre threads, processos e a CPU é um aspecto crucial para a compreensão de seu impacto no desempenho do sistema. A CPU é responsável por executar as instruções associadas a cada thread ou processo, e a forma como esses elementos são gerenciados pelo sistema operacional afeta diretamente a eficiência da execução. Nesse sentido, a troca de contexto é um processo relevante, pois envolve a suspensão de uma tarefa em execução para dar lugar a outra, preservando o estado da primeira para retomá-la posteriormente. Embora essencial para a multitarefa, a troca de contexto pode introduzir uma sobrecarga que impacta negativamente o desempenho, especialmente em sistemas com grande número de processos ou threads.

Além disso, o desempenho também é influenciado pelas operações de entrada e saída, que frequentemente representam gargalos no processamento. Threads podem ser vantajosas em aplicações que dependem intensamente de E/S, permitindo que uma thread lide com uma operação de E/S enquanto outras continuam executando, maximizando o uso da CPU. Por outro lado, o isolamento proporcionado por processos é útil em cenários onde a segurança e a estabilidade são prioritárias, como em servidores ou sistemas distribuídos.

Ao longo deste trabalho, serão discutidas as principais vantagens e limitações de cada abordagem, com ênfase na aplicação prática em um cenário computacional específico. Serão

apresentados exemplos e estudos que ilustram como processos e threads podem ser utilizados de forma a compreender suas vantagens, comparando níveis de desempenho entre dois dispositivos, aproveitando as peculiaridades de cada um para lidar com os desafios impostos pelos sistemas modernos. Espera-se que este estudo contribua para uma compreensão mais ampla e detalhada desses conceitos.

Este trabalho tem como objetivo comparar o desempenho de threads e processos em cenários de processamento CPU-Bound. O problema em questão se baseia no corolário:

> "Aplicação estruturada como processo para processamento CPU-Bound e Operação E/S não oferece vantagens de desempenho em relação a aplicações estruturada como threads para esse tipo de processamento; diferentemente que se estruturada como thread."

Para validar este corolário, implementou-se um algoritmo que realiza o cálculo de fatoriais repetidamente em 10^6 iterações, uma operação típica de alta demanda de CPU. Testes foram conduzidos variando o número de threads e processos, desde potências de 2 até valores incrementais.

O estudo utiliza o ambiente **Windows** e **Linux** em computadores com as seguintes especificações:

Dispositivo 1: Nitro AN515-55

- **Processador**: Intel® Core™ i5-10300H CPU @ 2.50GHz
- **Memória RAM**: 8,00 GB (utilizável: 7,83 GB)
- **Sistema**: Windows 64 bits – Windows 11 Home Single Language

Dispositivo 2: Samsung Book

- **Processador**: Intel® Core™ i3 11ª geração
- **Memória RAM**: 4,00 GB
- **Sistema**: GNU/Linux 64 bits – Debian 12

Os resultados obtidos fornecem a base para determinar o ponto de inflexão, onde os processos deixam de ser vantajosos, corroborando ou não o corolário apresentado.

1.1. OBJETIVOS

1.1.1. OBJETIVO GERAL

Identificar as circunstâncias em que processamento estruturado como threads ou, processos apresentam melhor desempenho para CPU-Bound e operações de I/O (Input e Output – entrada e saída) em um cenário do qual divida o comprimento pelo diâmetro com 10^6 iterações em uma circunferência com raio iniciando em 10^3, e em cada uma dessas iterações com incremento de 10.

1.1.2. OBJETIVOS ESPECÍFICOS

- Buscar em livros e demais materiais sobre temas relacionados ao objeto de estudos;
- Assimilar materiais de apoio;
- Interpretar o corolário;
- Construir lógica de apoio;
- Implementar algoritmo em Python;
- Validar o corolário;
- Determinar ponto de inflexão;
- Avaliar proposta com notação assintótica;

2. FUNDAMENTAÇÃO TEÓRICA

No que se refere a hardware, a palavra thread apareceu no Brasil com os primeiros modelos de processador e com múltiplos núcleos. Para que o entendimento de threads seja visualizado claramente, deve-se pontuar como se dá a execução de um programa.

Basicamente, a execução de um programa se dá em uma ação do sistema operacional. Quando aberto um aplicativo pelo usuário, é o sistema operacional que interpretará a ação e requisições dos arquivos relacionados ao software a ser executado.

Quaisquer que sejam as atividades de um SO (Sistema Operacional), elas estão sujeitas às operações do processador. Um programa, antes de ser aberto e passar por todas as requisições, ele apenas é carregado na RAM sem exigir uma atividade propriamente dita do processador. Quando efetuado esse carregamento, o Sistema Operacional trabalhará com os processos.

Processos são módulos executáveis com linhas de código para que seja realizada a execução do programa. Tanenbaum (2010) define processos de um computador como “uma abstração de um programa em execução” e comenta sobre como essa abstração foi de suma importância para a ciência da computação moderna.

Processadores conseguem trabalhar muito bem com processos, mas a execução “extrapolada” simultaneamente de processos acarreta na lentidão da CPU. Com o tempo, a evolução dos softwares e dos componentes de Hardware acarretou na necessidade de uma melhor divisão de tarefas e, é nesse contexto em que se têm *threads*. As linhas de instruções de processos adquiriram características únicas, possibilitando a separação para diferentes núcleos, essas são as threads.

Threads podem ser entendidas como uma divisão do processo principal de um programa. Corresponde à unidade básica de utilização da CPU, consistindo de contador de programa, conjunto de registradores e uma pilha de execução.

Apesar de threads serem uma solução eficiente para o processamento paralelo, há mecanismos em certas implementações de linguagens que limitam sua execução plena em sistemas multi-core. Um exemplo notável é o Global Interpreter Lock (GIL), presente em alguns interpretadores, que atua como uma trava global para gerenciar o acesso a recursos compartilhados, restringindo a execução simultânea de threads. Esse tipo de limitação ressalta a importância de compreender as nuances entre o comportamento das threads no hardware e as restrições impostas por software.

3. GIL (Global Interpreter Lock)

O GIL é um mecanismo de sincronização que controla o acesso a objetos em Python, garantindo que apenas uma thread execute em qualquer momento dado. Embora isso seja benéfico para evitar condições de corrida e simplificar a implementação de estruturas de dados em Python, ele impõe limitações significativas em sistemas multi-core. O GIL é frequentemente apontado como a causa para a falta de escalabilidade de threads em Python, especialmente para tarefas que exigem grande poder de processamento em paralelo, como cálculos científicos ou computação gráfica.

O Global Interpreter Lock (GIL) é amplamente discutido na literatura técnica e em materiais sobre a implementação de Python. Um autor frequentemente citado nesse contexto é David Beazley, que apresentou palestras detalhadas sobre o GIL, como “Understanding the Python GIL” em conferências como PyCon. Além disso, o livro Programming Python de Mark Lutz, aborda o GIL ao tratar de multithreading em Python.

4. THREADS E ESCALABILIDADE

Threads em Python compartilham o mesmo espaço de endereçamento e, portanto, compartilham também a memória global. No entanto, o GIL impede que múltiplas threads executem simultaneamente, mesmo em um sistema com múltiplos núcleos, resultando em uma escalabilidade limitada.

A escalabilidade é uma medida crítica da capacidade de um sistema de aumentar o desempenho ao adicionar mais recursos (por exemplo, núcleos de CPU). No caso de Python, o impacto do GIL significa que, para aumentar o desempenho, é necessário evitar tarefas que utilizem intensivamente a CPU dentro de um único processo. A adoção de multiprocessing é uma alternativa mais eficaz para aplicações que demandam alta performance, pois cada processo é executado em um ambiente separado, sem ser limitado pelo GIL.

5. OVERHEAD E EFICIÊNCIA

O overhead refere-se à quantidade de tempo e recursos consumidos pela gestão do sistema, incluindo a troca de contexto entre threads e processos. Tanenbaum(2014) afirma que "Threads are lighter weight than processes, but they still have overhead" concluindo há leveza nas threads mas mesmo assim o overhead é inevitável. E como link para outros sistemas,

determina-se que em sistemas multi-threading em Python, a troca de contexto entre threads, mesmo que rápida, é inevitável devido ao GIL. Isso adiciona uma sobrecarga que não ocorre em sistemas que utilizam multiprocessamento, onde cada processo tem seu próprio GIL. A eficiência de escalabilidade, portanto, pode ser comprometida, especialmente quando o overhead de contexto-switching é significativo.

6. METODOLOGIA

O experimento segue a necessidade de comprovação do corolário apresentado. Para que este seja comprovado e conclusões sejam tomadas, as seguintes etapas foram realizadas:

1. Configuração do Cenário:
    - Número total de iterações: 10^6
    - Testes realizados com dois conjuntos de valores:
        a. Potências de 2: Variação de 1 até 512 threads/processos.
        b. Valores de 300 a 1000: Avaliação de desempenho incremental.
    - Intervalo incremental de 500 a 512: Teste mais detalhado para identificar o comportamento no limite.
2. Implementação do Algoritmo:
    - Operação CPU-Bound: Com as repetições do cálculo de fatorial para a simulação do processamento pesado em CPU.
    - Execução do algoritmo foi dividida em:

Threads: Dividem as iterações, mas compartilham os recursos do núcleo da CPU (impactadas pelo GIL do Python).

Processos: São criados independentemente e conseguem utilizar múltiplos núcleos de CPU em paralelo.

3. Medição de Desempenho:
    - O tempo de execução foi registrado usando a função **time.perf_counter** para alta precisão.
    - Tabelas foram geradas comparando os tempos de execução entre threads e processos.
4. Ferramentas Utilizadas:
    - Linguagem de programação Python.
    - Bibliotecas: threading, multiprocessing, time e math.
5. Critérios de Análise:
    - Comparação dos tempos de execução das threads e processos.
    - Identificação do ponto de inflexão, onde os processos deixam de ser vantajosos.
6. Plotagem gráfica
    - A geração dos gráficos foi realizada utilizando um código em Python, conforme mostrado nos apêndices A e B.

7. IMPLEMENTAÇÃO

7.1. threadsXprocessos.py

A fim de validar o corolário, um arquivo de extensão .py foi desenvolvido no ambiente Linux, assim como seus primeiros testes. A programação de baixo nível foi fundamentada no mesmo código, devido a criação de processos isolados que executam independentemente no multiprocessamento e também o uso do multithreading.

Operações fatoriais se tornaram uma alternativa para os testes, justamente por serem operações pesadas computacionalmente e, o uso eficiente dos núcleos da CPU balanceou as tarefas, evitando gargalos. Para tratamento das falhas de sistema retornamos uma mensagem ao usuário quando os limites de seu sistema não foram atingidos.

CÓDIGO INPLEMENTADO

```
import os
import time
from math import factorial
from multiprocessing import Process
from threading import Thread

# Número de iterações e lista de testes principais
ITERACOES = 10**6
NUM_TESTES_POTENCIA_2 = [2**x for x in range(0, 10)]  # Potências de 2
NUM_SEQUENCIA = [300, 400, 500, 600, 700, 800, 900, 1000]

# Função para o cálculo CPU-Bound
def calcular(iteracoes):
    total = 0
    for _ in range(iteracoes):
        total += factorial(10)

def executar_threads(iteracoes, num_threads):
    threads = []
    for _ in range(num_threads):
        t = Thread(target=calcular, args=(iteracoes // num_threads,))
        t.start()
        threads.append(t)
    for t in threads:
        t.join()

def executar_processos(iteracoes, num_processos):
```

```
    processos = []
    for _ in range(num_processos):
        p = Process(target=calcular, args=(iteracoes // num_processos,))
        p.start()
        processos.append(p)
    for p in processos:
        p.join()

# Função para monitorar desempenho (tempo)
def monitorar_tempo():
    return time.perf_counter()

def imprimir_tabela(header, resultados, colunas):
    print(f"\n{header}")
    print(f"{'Num Threads/Processos':<25} {'Tempo Threads (s)':<25} {'Tempo
Processos (s)':<25}" if colunas == 3 else
          f"{'Num Threads/Processos':<25} {'Tempo Processos (s)':<25}")
    print("-" * (60 if colunas == 3 else 40))

    for resultado in resultados:
        num, *tempos = resultado
        tempos_formatados = [f"{tempo:.4f}" if isinstance(tempo, (float, int))
else "FALHA" for tempo in tempos]
        print(f"{num:<25} {' '.join(f'{t:<25}' for t in tempos_formatados)}")

def executar_testes(lista_tamanhos, iteracoes, resultados):
    for num in lista_tamanhos:
        try:
            inicio = time.time()
            executar_threads(iteracoes, num)
            tempo_threads = time.time() - inicio

            inicio = time.time()
            executar_processos(iteracoes, num)
            tempo_processos = time.time() - inicio
        except OSError:
            tempo_processos = "FALHA"

        resultados.append((num, tempo_threads, tempo_processos))

# Testes incrementais entre valores
def testes_incrementais(intervalo_inicio, intervalo_fim, iteracoes):
    resultados_incrementais = []
    falhou = False

    for num in range(intervalo_inicio, intervalo_fim + 1):
```

```
        if falhou:
            resultados_incrementais.append((num, "FALHA"))
            continue

        try:
            inicio = monitorar_tempo()
            executar_processos(iteracoes, num)
            tempo_processos = monitorar_tempo() - inicio
            resultados_incrementais.append((num, tempo_processos))
        except OSError:
            resultados_incrementais.append((num, "FALHA"))
            falhou = True  # Interromper tentativas subsequentes

    return resultados_incrementais


# Função principal
def main():
    resultados_1 = []
    resultados_2 = []
    resultados_incrementais = []

    print("Iniciando testes...\n")

    executar_testes(NUM_TESTES_POTENCIA_2, ITERACOES, resultados_1)
    imprimir_tabela("Testes com potências de 2:", resultados_1, colunas=3)


    executar_testes(NUM_SEQUENCIA, ITERACOES, resultados_2)
    imprimir_tabela("Testes de 300 a 1000:", resultados_2, colunas=3)

    resultados_incrementais = testes_incrementais(500, 512, ITERACOES)
    imprimir_tabela("Resultados incrementais entre 500 e 512:",
resultados_incrementais, colunas=2)


    ################## Salvar resultados em arquivo ##################
    with open("resultados.saida", "w") as f:
        f.write("Testes com potências de 2:\n")
        f.write(f"{'Num Threads/Processos':<25} {'Tempo Threads (s)':<25}
{'Tempo Processos (s)':<25}\n")
        f.write("-" * 75 + "\n")
        for resultado in resultados_1:
            num, *tempos = resultado
            tempos_formatados = [f"{tempo:.4f}" if isinstance(tempo, (float,
int)) else "FALHA" for tempo in tempos]
```

```
            f.write(f"{num:<25} {' '.join(f'{t:<25}' for t in
tempos_formatados)}\n")

        f.write("\nTestes de 300 a 1000:\n")
        f.write(f"{'Num Threads/Processos':<25} {'Tempo Threads (s)':<25}
{'Tempo Processos (s)':<25}\n")
        f.write("-" * 75 + "\n")
        for resultado in resultados_2:
            num, *tempos = resultado
            tempos_formatados = [f"{tempo:.4f}" if isinstance(tempo, (float,
int)) else "FALHA" for tempo in tempos]
            f.write(f"{num:<25} {' '.join(f'{t:<25}' for t in
tempos_formatados)}\n")

        f.write("\nResultados incrementais entre 500 e 512:\n")
        f.write(f"{'Num Threads/Processos':<25} {'Tempo Processos
(s)':<25}\n")
        f.write("-" * 50 + "\n")
        for resultado in resultados_incrementais:
            num, tempo = resultado
            tempo_formatado = f"{tempo:.4f}" if isinstance(tempo, (float,
int)) else "FALHA"
            f.write(f"{num:<25} {tempo_formatado:<25}\n")

    print("\nResultados salvos em 'resultados.saida'.")

if __name__ == "__main__":
    main()
```

O algoritmo acima segue a seguinte lógica:

Para entendimento do código, temos como as parametrizações ITERACOES, que são o número total de iterações e NUM_TESTES_POTENCIA_2 e NUM_SEQUENCIA que correspondem aos tamanhos incrementais de threads/processos a serem testados. Logo, as respectivas funções são declaradas:

1. Função calcular:
    - Realiza fatorial (10) em um loop de 10^6 iterações. E serão esses cálculos intensivos que simularão a CPU-Bound.
2. Execução com Threads e Processos:

- ***executar_threads**:* Divide as iterações entre o número de threads, executando o cálculo e "compartilhando" os recursos da CPU.
- ***executar_processos**:* Divide as iterações entre processos independentes que podem usar múltiplos núcleos da CPU.

3. Medição do Tempo:
   - A função ***monitorar_tempo*** mede o tempo de execução dos threads e processos.
4. Testes Realizados:
   - Testes com potências de 2: Número de threads/processos = 1, 2, 4, ..., 512.
   - Testes incrementais: Variação entre 300 e 1000 threads/processos.
   - Intervalo detalhado (500 a 512): Identificação precisa de um possível ponto de inflexão.
5. Resultado

   Os tempos de execução são exibidos no terminal/console em tabelas formatadas, assim como também é gerado um arquivo *resultados.saida* onde os resultados ou tabelas são salvos nesse txt.

7.2. COMPLEXIDADE E NOTAÇÃO ASSINTÓTICA

O cálculo de fatorial é O(1) para o número fixo n, que é 10. Porém, com as repetições em iterações a complexidade é O(n), visto que n corresponde ao número de iterações.

Com threads, o GIL acaba que sendo um limitador do paralelismo, mantendo o tempo próximo de uma execução sequencial. Enquanto com processos, a execução pode ser paralelizada em múltiplos núcleos, fazendo com que o tempo real da execução seja reduzido conforme o aumento do número de processos.

Dando enfoque ao uso dos recursos, threads compartilham o mesmo espaço de memória, permitindo que os mesmos sejam economizado, mas não se tornam "ineficientes" em CPU-Bound. Agora, para uma visão aos processos temos uma abordagem e conclusões diferentes visto que processos possuem seu próprio espaço de endereçamento e uma única linha de controle. Logo, um processamento orientado/estruturado em processos se torna menos eficiente em relação ao consumo de memória, que é maior do que em threads. Em contrapartida, essa estrutura realiza um paralelismo verdadeiro característico.

8. RESULTADO E ANÁLISE

Os resultados obtidos no dispositivo **AcerNitro5_ton** com especificações descritas são apresentados a seguir.

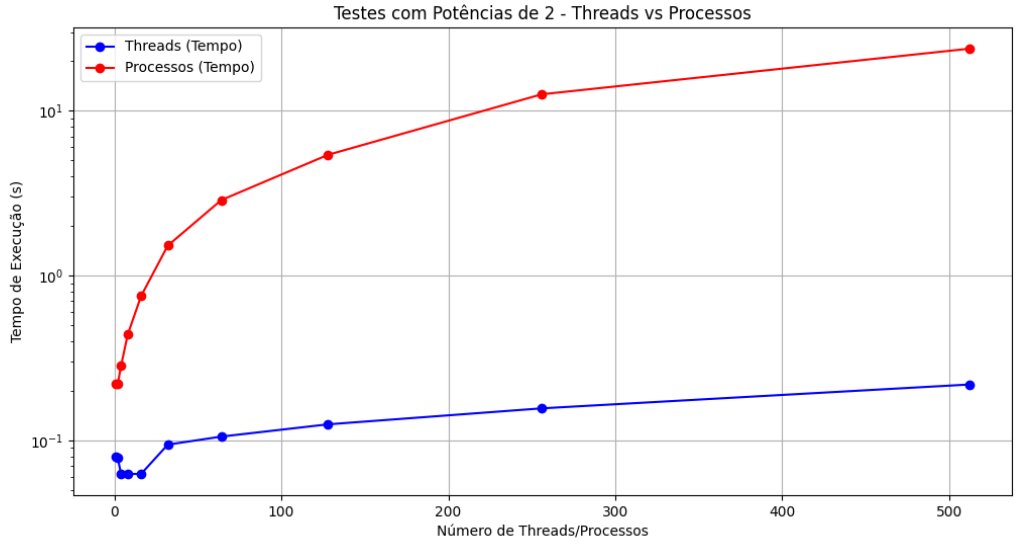
Testes com potências de 2:

Num Threads/Processos	Tempo Threads (s)	Tempo Processos (s)
1	0.0791	0.2203
2	0.0781	0.2187
4	0.0625	0.2836
8	0.0625	0.4387
16	0.0625	0.7510
32	0.0941	1.5198
64	0.1053	2.8705
128	0.1250	5.4101
256	0.1563	12.6012
512	0.2181	23.8309

- Análise:

  Com threads, o tempo de execução não cresce significativamente devido à limitação do GIL. Enquanto com processos, o tempo cresce exponencialmente após 8 processos, evidenciando a sobrecarga de criação e gerenciamento.

Gráfico 01: Teste com potências de 2 – Dispositivo1



Fonte: Os autores

Os tempos com **threads** permanecem relativamente estáveis até certo ponto, com pequenos aumentos conforme o número de threads cresce. Isso se deve ao fato de que estruturas baseadas em threads compartilham o mesmo espaço de memória, resultando em um menor overhead de criação e comunicação, que logo é refletido nos menores tempos.

Obs.: A partir dos 32 threads, o tempo aumenta progressivamente, sugerindo sobrecarga de gerenciamento de threads e perda de eficiência devido ao custo de sincronização.

Diferentemente dos threads, o tempo para **processos** aumenta consideravelmente à medida que o número de processos cresce. Processos são independentes e têm custos adicionais de Troca de Contexto (context switch) e Comunicação Inter Processo (IPC), especialmente conforme o número de processos cresce.

Obs.: O custo de criação, comunicação e gerenciamento de processos é maior do que o dos threads. Para 512 processos, o tempo chega a 23.83s, enquanto com threads é apenas de 0.2181s.

Testes de 300 a 1000:

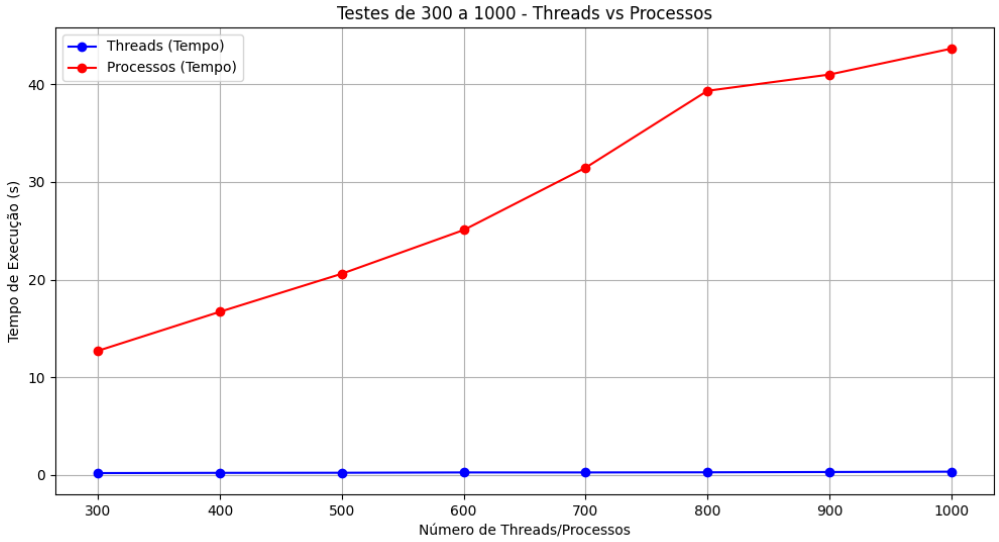
Num Threads/Processos	Tempo Threads (s)	Tempo Processos (s)
300	0.1704	12.7040
400	0.1931	16.7012
500	0.2031	20.5975

600	0.2351	25.0761
700	0.2343	31.4623
800	0.2500	39.3433
900	0.2812	41.0068
1000	0.3125	43.6622

- Análise:

  O desempenho de processos se deteriora com o aumento de tarefas devido ao alto custo de gerenciamento de múltiplos processos. Em contrapartida, threads continuam estáveis.

  Gráfico 02: Teste Threads/Processos 300 a 1000 – Dispositivo1



Fonte: Os autores

Esta estabilidade ocorre com a linearidade do crescimento, enquanto os processos são exponenciais. Alguns exemplos notáveis são:

300 processos: 12.7040s

1000 processos: 43.6622s

Esse crescimento exponencial nos tempos dos processos reforça o impacto negativo do overhead na criação, gerenciamento e comunicação de um grande número de processos.

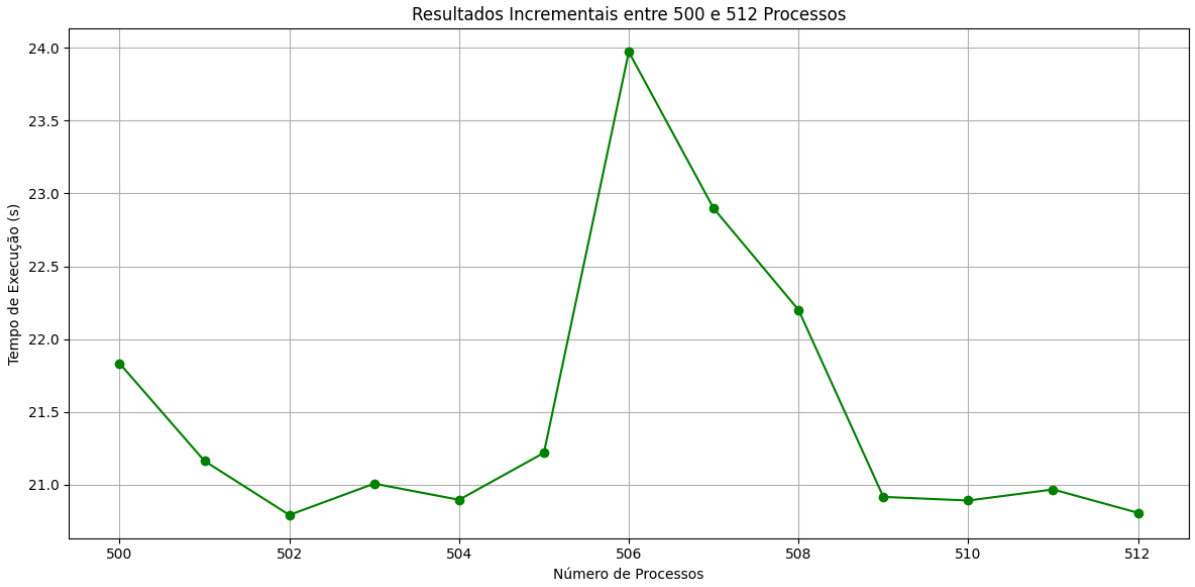
Resultados incrementais entre 500 e 512:

Num Threads/Processos	Tempo Processos (s)
500	21.8331
501	21.1642
502	20.7935
503	21.0086
504	20.8977
505	21.2200
506	23.9714
507	22.8972
508	22.2011
509	20.9176
510	20.8928
511	20.9687
512	20.8083

- Análise:

Observa-se que, mesmo para valores próximos, os processos sofrem degradação de desempenho devido ao gerenciamento de processos.

Gráfico 03: Teste Processos 500 a 512 – Dispositivo1



Fonte: Os autores

Os tempos oscilam de forma irregular, entre 20.7935 s e 23.9714 s, sem uma tendência clara de aumento ou diminuição.

Esse comportamento pode ser explicado por:

Sobrecarga do sistema operacional: A criação de processos adicionais pode sobrecarregar a CPU e outros recursos do sistema.

Fatores externos: Pequenas variações podem ocorrer devido a concorrência de recursos ou à aleatoriedade do escalonador do sistema operacional.

Os resultados obtidos pelo dispositivo Samsung Book para análise se encontram logo abaixo.

Testes com potências de 2:

Num Threads/Processos	Tempo Threads (s)	Tempo Processos (s)
1	0.1166	0.1232
2	0.1147	0.0595
4	0.1131	0.0663
8	0.1165	0.0716
16	0.1164	0.0748
32	0.1253	0.0875
64	0.1306	0.1139
128	0.1320	0.1655
256	0.1496	0.2759
512	0.1710	FALHA

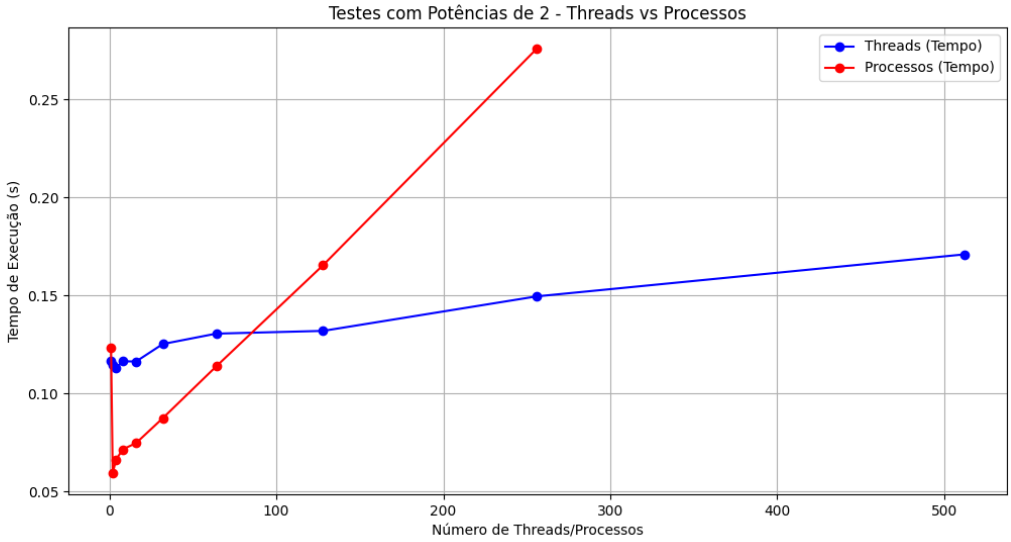
- Análise:

  Os tempos com threads aumentam de forma gradual à medida que o número de threads cresce, mesmo que esse aumento seja relativamente pequeno. Enquanto isso, nota-se que o tempo de processos diminui entre 1 e 2 processos, mas cresce à medida que os números aumentam.

Os processos apresentam uma grande ineficiência para números elevados de processo. Com 512, o teste falha no dispositivo em questão, indicando que o sistema pode ter alcançado

seu limite de recursos, como quantidade máxima de processos simultâneos ou até mesmo memória insuficiente.

Gráfico 04: Teste com potências de 2 – Dispositivo2



Fonte: Os autores

Testes de 300 a 1000:

Num Threads/Processos	Tempo Threads (s)	Tempo Processos (s)
300	0.1654	0.3236
400	0.1644	0.3971
500	0.1657	0.6033
600	0.1901	FALHA
700	0.2029	FALHA
800	0.3572	FALHA
900	0.2442	FALHA
1000	0.3285	FALHA

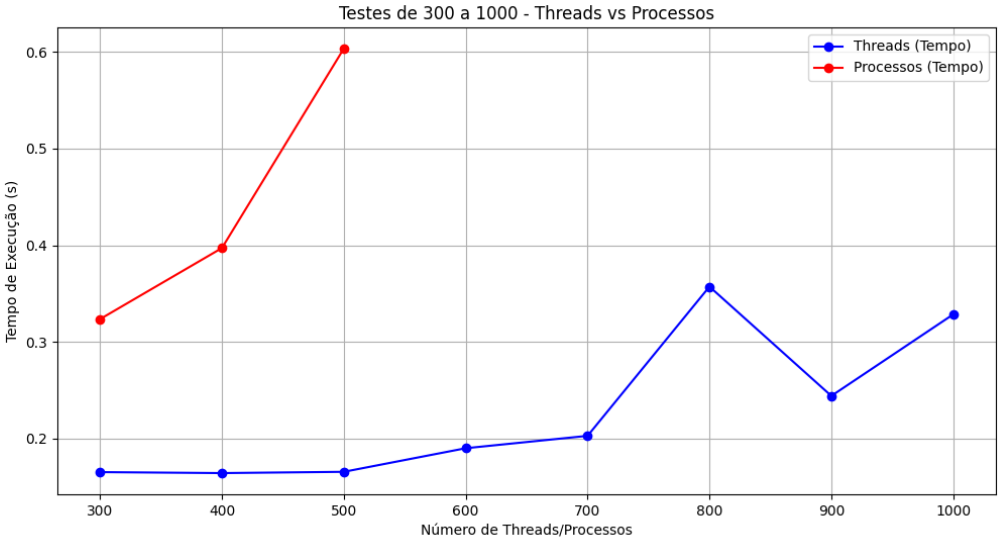
- Análise:

  Os tempos com threads permanecem estáveis até 600, mas começam a sofrer um a elevação bem instável para números acima

de 800, enquanto os processos começam a falhar com apenas 37,5% dos testes percorridos para os processos.

A instabilidade recorrente para threads pode ser explicada pelo aumento do gerenciamento de threads, enquanto para os processos, sugere-se que o sistema não consegue lidar com grandes números de processos simultâneos devido às limitações de recursos.

Gráfico 05: Teste Threads/Processos 300 a 1000 – Dispositivo2



Fonte: Os autores

Resultados incrementais entre 500 e 512:

Num Threads/Processos	Tempo Processos (s)
500	0.7716
501	0.8709
502	0.8644
503	0.8624
504	FALHA
505	FALHA
506	FALHA
507	FALHA

508	FALHA
509	FALHA
510	FALHA
511	FALHA
512	FALHA

- Análise:

Os testes de 500 a 504 são concluídos com tempos bem próximos, mas logo surtem falhas que são recorrentes.

Fatores possíveis para que esses resultados da tabela sejam adquiridos são tanto o atingimento do limite crítico dos recursos quanto a quantidade máxima de processos ou restrições impostas pelo Sistema Operacional.

Gráfico 06: Teste Processos 500 a 512 – Dispositivo2

Fonte: Os autores

9. CONCLUSÃO

Os resultados apresentados validam o corolário inicial: aplicações estruturadas como processos não oferecem vantagens significativas em operações CPU-Bound. A análise mostrou que:

1. **Threads** apresentaram desempenho consistente e estável, independentemente do número de tarefas, devido ao compartilhamento de memória e ao impacto do GIL.
2. **Processos** apresentaram um desempenho inicial competitivo, mas sofrem **crescimento exponencial no tempo de execução** conforme o número de processos aumenta, devido ao custo de criação e gerenciamento de múltiplos processos.

Assim, em cenários CPU-Bound, o uso de threads é mais eficiente em sistemas onde o GIL impõe restrições (como no Python). Processos, por outro lado, só são vantajosos em pequenas quantidades, mas perdem eficiência à medida que o paralelismo cresce.

A análise revelou que, embora o GIL seja um mecanismo que simplifica a implementação em Python e ajuda a evitar condições de corrida, ele limita severamente a escalabilidade de threads, especialmente para tarefas CPU-Bound. Portanto, para melhorar a eficiência e o desempenho de aplicações **Python** que envolvem cálculos intensivos, o uso de multiprocessing é geralmente preferido.

Esta metodologia e os resultados refletem uma perspectiva técnica mais detalhada sobre o impacto do GIL, o overhead nos processos e a eficácia de diferentes abordagens de paralelismo em Python, alinhando-se com as concepções acadêmicas de context-switching, paralelismo verdadeiro e concorrência.

Além das limitações observadas, é importante considerar os desafios adicionais que podem surgir em ambientes de concorrência, como deadlocks (“abraço fatal”) e condições de corrida (quando duas ou mais threads ou processos acessam simultaneamente um recurso compartilhado, impossibilitando a depuração).

Embora o código não apresente riscos de deadlock, cenários complexos como essa implementação poderiam levar à essas situações. No algoritmo desenvolvido a implementação de uma fila compartilhada para coordenação de tarefas poderia introduzir esse problema, caso as travas não sejam gerenciadas adequadamente, o que não ocorreu.

Os riscos devem ser descartados, pois cada thread ou processo executa de forma independente e não há uso de travas compartilhadas, além de não existir dependências ou comunicação entre as estruturas.

Com relação às condições de corrida, no código as threads compartilham o mesmo espaço de memória, o que é eficiente para tarefas independentes, mas perigoso para manipulações concorrentes de variáveis globais ou recursos compartilhados. Embora o GIL em Python reduza a probabilidade de condições de corrida, isso não elimina completamente os problemas em sistemas que requerem sincronização explícita.

10. REFERÊNCIAS

TANENBAUM, A. S. Sistemas operacionais modernos / Andrew S. Tanenbaum; tradução Ronaldo A.L. Gonçalves, Luís A. Consularo, Luciana do Amaral Teixeira; revisão técnica Raphael Y. de Camargo. – 3. Ed – São Paulo: Prentice-Hall Do Brasil, 2010. 653 p. ISBN 9788576052371.

JORDÃO, Fabio. O que são threads em um processador?. Tecmundo, 18 abr. 2011. Disponível em: https://www.tecmundo.com.br/9669-o-que-sao-threads-em-um-processador-.htm. Acesso em: 18 dez. 2024.

BEAZLEY, David. Understanding the Python GIL. PyCon 2010, Atlanta, GA, 2010. Disponível em: https://www.youtube.com/watch?v=Obt-vMVdM8s. Acesso em: 18 dez. 2024.

TANENBAUM, A. S.; BOS, H. Modern Operating Systems. 4ª ed. Pearson, 2014.

STUART, Brian L. Princípios de sistemas operacionais: projetos e aplicações. São Paulo: Cengage Learning, 2011. ISBN 9788522107339.

APÊNDICE

APÊNDICE A

```
# inserir o comando caso não possua a biblioteca: pip install matplotlib
import matplotlib.pyplot as plt

# Dados fornecidos
# Testes com potências de 2
threads_potencias2 = [0.0791, 0.0781, 0.0625, 0.0625, 0.0625, 0.0941, 0.1053,
0.1250, 0.1563, 0.2181]
processos_potencias2 = [0.2203, 0.2187, 0.2836, 0.4387, 0.7510, 1.5198,
2.8705, 5.4101, 12.6012, 23.8309]
num_potencias2 = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

# Testes de 300 a 1000
threads_300_1000 = [0.1704, 0.1931, 0.2031, 0.2351, 0.2343, 0.2500, 0.2812,
0.3125]
processos_300_1000 = [12.7040, 16.7012, 20.5975, 25.0761, 31.4623, 39.3433,
41.0068, 43.6622]
num_300_1000 = [300, 400, 500, 600, 700, 800, 900, 1000]

# Resultados incrementais entre 500 e 512
processos_incrementais = [21.8331, 21.1642, 20.7935, 21.0086, 20.8977,
21.2200, 23.9714, 22.8972, 22.2011, 20.9176, 20.8928, 20.9687, 20.8083]
num_incrementais = list(range(500, 513))

# Gráfico 1: Testes com potências de 2
plt.figure(figsize=(12, 6))
plt.plot(num_potencias2, threads_potencias2, marker='o', label='Threads
(Tempo)', color='blue')
plt.plot(num_potencias2, processos_potencias2, marker='o', label='Processos
(Tempo)', color='red')
plt.title('Testes com Potências de 2 - Threads vs Processos')
plt.xlabel('Número de Threads/Processos')
plt.ylabel('Tempo de Execução (s)')
plt.yscale('log')  # Log scale para observar a diferença exponencial
plt.legend()
plt.grid()

# Gráfico 2: Testes de 300 a 1000
plt.figure(figsize=(12, 6))
plt.plot(num_300_1000, threads_300_1000, marker='o', label='Threads (Tempo)',
color='blue')
plt.plot(num_300_1000, processos_300_1000, marker='o', label='Processos
(Tempo)', color='red')
plt.title('Testes de 300 a 1000 - Threads vs Processos')
```

```
plt.xlabel('Número de Threads/Processos')
plt.ylabel('Tempo de Execução (s)')
plt.legend()
plt.grid()

# Gráfico 3: Resultados incrementais entre 500 e 512 processos
plt.figure(figsize=(12, 6))
plt.plot(num_incrementais, processos_incrementais, marker='o', color='green')
plt.title('Resultados Incrementais entre 500 e 512 Processos')
plt.xlabel('Número de Processos')
plt.ylabel('Tempo de Execução (s)')
plt.grid()

# Mostrar os gráficos
plt.tight_layout()
plt.show()
```

APÊNDICE B

```
import matplotlib.pyplot as plt

threads_potencias2 = [0.1166, 0.1147, 0.1131, 0.1165, 0.1164, 0.1253, 0.1306,
0.1320, 0.1496, 0.1710]
processos_potencias2 = [0.1232, 0.0595, 0.0663, 0.0716, 0.0748, 0.0875,
0.1139, 0.1655, 0.2759, None]
num_potencias2 = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

threads_300_1000 = [0.1654, 0.1644, 0.1657, 0.1901, 0.2029, 0.3572, 0.2442,
0.3285]
processos_300_1000 = [0.3236, 0.3971, 0.6033, None, None, None, None, None]
num_300_1000 = [300, 400, 500, 600, 700, 800, 900, 1000]

processos_incrementais = [0.7716, 0.8709, 0.8644, 0.8624, None, None, None,
None, None, None, None, None]
num_incrementais = list(range(500, 513))

plt.figure(figsize=(12, 6))
plt.plot(num_potencias2, threads_potencias2, marker='o', label='Threads
(Tempo)', color='blue')
plt.plot(num_potencias2, [t if t is not None else float('nan') for t in
processos_potencias2],
         marker='o', label='Processos (Tempo)', color='red')
plt.title('Testes com Potências de 2 - Threads vs Processos')
plt.xlabel('Número de Threads/Processos')
plt.ylabel('Tempo de Execução (s)')
```

```
plt.legend()
plt.grid()

plt.figure(figsize=(12, 6))
plt.plot(num_300_1000, threads_300_1000, marker='o', label='Threads (Tempo)',
color='blue')
plt.plot(num_300_1000, [t if t is not None else float('nan') for t in
processos_300_1000],
         marker='o', label='Processos (Tempo)', color='red')
plt.title('Testes de 300 a 1000 - Threads vs Processos')
plt.xlabel('Número de Threads/Processos')
plt.ylabel('Tempo de Execução (s)')
plt.legend()
plt.grid()

plt.figure(figsize=(12, 6))
plt.plot(num_incrementais, [t if t is not None else float('nan') for t in
processos_incrementais],
         marker='o', color='green')
plt.title('Resultados Incrementais entre 500 e 512 Processos')
plt.xlabel('Número de Processos')
plt.ylabel('Tempo de Execução (s)')
plt.grid()

plt.tight_layout()
plt.show()
```