FAESA

SISTEMA FAESA DE EDUCAÇÃO

# Primeiro Trabalho - Sistemas Operacionais 2025/01

**ENTREGA:** 26/03/2025
**VALOR:** 40% de C1
**PROFESSOR:** Gabriel Soares Baptista

# Descrição

Este trabalho consiste em um roteiro didático para o desenvolvimento prático de Programação de Processos. O material utiliza a linguagem Python como referência, entretanto, vocês têm liberdade para implementar as soluções em outras linguagens de programação de sua preferência.

O roteiro está organizado em duas etapas principais:

**1. Fundamentação Teórico-Prática:**
Nesta primeira parte, será introduzido o funcionamento de processos em sistemas computacionais, acompanhado de exemplos práticos de implementação em Python. O conteúdo servirá como um guia complementar aos tópicos abordados (e parcialmente antecipará conceitos futuros) nas aulas teóricas.

**2. Atividade de Aplicação:**
A segunda etapa consiste em um exercício prático que deverá ser entregue para assegurar a conclusão do projeto e a adequada execução das etapas propostas. O desafio envolve a implementação de duas versões do algoritmo Merge Sort: Uma versão sequencial (sem uso de processos) e outra versão paralelizada (com processos).

Ao final, os alunos deverão realizar uma análise estatística comparativa entre ambas as implementações. O objetivo é demonstrar empiricamente os benefícios da paralelização por meio de processos neste contexto.

FAESA

# 1. Programando Processos em Python

Em termos técnicos, um processo em Python corresponde a uma instância do interpretador em execução. Cada processo é responsável por converter o código Python em instruções compreensíveis pela máquina e gerenciar sua execução. Todo processo contém pelo menos uma thread principal (conhecida como "**main thread**"), porém é possível criar threads adicionais dentro do mesmo processo para executar tarefas concorrentes. Para esclarecer:

- **Processo:** Instância do interpretador Python que opera como um ambiente isolado, contendo ao menos uma thread (a principal). Cada processo possui seu próprio espaço de memória e recursos alocados.
- **Thread:** Unidade básica de execução dentro de um processo. Representa o fluxo de controle do programa e compartilha o mesmo espaço de memória com outras threads do mesmo processo.

Python oferece duas classes estruturalmente semelhantes para gerenciar paralelismo e concorrência:

- **multiprocessing.Process** (para criação de processos independentes)
- **threading.Thread** (para criação de threads dentro de um mesmo processo)

Apesar da sintaxe similar, essas classes diferem significativamente em aspectos como isolamento de memória, consumo de recursos e modelos de concorrência, distinções críticas que serão exploradas adiante.

Embora as APIs das classes **multiprocessing.Process** e **threading.Thread** sejam semelhantes, processos e threads possuem diferenças fundamentais. Um processo é uma abstração de nível superior: representa um programa em execução com seu próprio espaço de memória, enquanto uma thread é uma unidade de execução dentro de um processo. Essa distinção reflete como o sistema operacional gerencia recursos: threads compartilham memória dentro do mesmo processo, permitindo acesso direto a variáveis e estados globais (modelo de "memória compartilhada"). Já processos, por terem memória isolada, exigem mecanismos explícitos de comunicação, como **multiprocessing.Queue** ou **multiprocessing.Pipe**, que serializam dados para transferência entre ambientes.

Um aspecto crítico em Python é o **Global Interpreter Lock** (**GIL**), que limita a execução paralela de threads: apenas uma thread por processo pode executar código Python em um dado momento. Isso torna o multithreading ineficiente para tarefas intensivas em CPU, já que não há ganho real de paralelismo. Porém, processos escapam dessa limitação, pois cada um tem seu próprio GIL, permitindo execução verdadeiramente paralela em múltiplos núcleos.

Para tarefas vinculadas a E/S (como operações de rede ou leitura de arquivos), threads são mais adequadas, pois permitem concorrência sem bloquear a aplicação. Por outro lado, processos são ideais para tarefas computacionais pesadas que exigem paralelismo real.

Neste trabalho, utilizaremos a biblioteca **multiprocessing** para alinhar com o conteúdo abordado em aula. Contudo, é importante destacar que, para a implementação do algoritmo Merge Sort, threads poderiam ser mais vantajosas em cenários específicos. Isso ocorre porque o Merge Sort envolve divisão recursiva de dados e combinação de subconjuntos ordenados, operações que podem se beneficiar do acesso direto à memória compartilhada entre threads, evitando a sobrecarga de serialização e comunicação entre processos. Ainda assim, o uso de processos permite explorar paralelismo em nível de sistema, útil para análises comparativas de desempenho, que é o foco principal desta atividade. Notem que essa limitação do GIL é oriunda do Python, dessa forma, outras linguagens seriam mais beneficiadas pela utilização de threads do que de processos para implementar o Merge Sort.

Em resumo, a escolha entre processos e threads depende da natureza da tarefa. Enquanto processos oferecem paralelismo verdadeiro para tarefas intensivas em CPU, threads são mais eficientes para operações concorrentes com alto grau de compartilhamento de estado.

## 1.2. O Básico do módulo Python Multiprocessing

O ciclo de vida de um processo em Python abrange três estágios: criação, execução e término. Vamos detalhar cada fase com exemplos práticos.

### 1.2.1. Criação do Processo

Um novo processo é criado ao instanciar a classe **multiprocessing.Process**. Quando atribuímos essa instância a uma variável, um processo filho é gerado. Por padrão, o processo não executa nenhuma ação até que seja configurado. Para definir sua funcionalidade, especificamos uma função alvo (**target**) e argumentos (**args**), se necessário.

```
from multiprocessing import Process

# Função alvo sem parâmetros
def exemplo():
    pass
```

```
processo = Process(target=exemplo)
print("Processo criado, mas ainda não iniciado.")
```

Se a função exigir parâmetros, eles são passados como uma tupla no argumento args:

```
def saudacao(mensagem):
    print(mensagem)

processo_com_args = Process(target=saudacao, args=("Olá, mundo!",))
```

### 1.2.2. Execução

O processo entra em execução ao chamar o método **start()**. Internamente, isso dispara o método **run()**, que executa a função definida em target. Cada processo filho possui sua própria thread principal, responsável por executar o código designado.

Para verificar se o processo está ativo, usamos is_alive():

```
processo.start()
print("Processo em execução:", processo.is_alive())  # Saída: True
```

**Coloque um time.sleep(5) antes de printar a mensagem (dentro da função alvo "saudacao") para que o processo não termine antes de você fazer a verificação.**

### 1.2.3. Término

Um processo termina naturalmente quando sua função alvo é concluída ou se ocorrer um erro não tratado. Após a finalização, is_alive() retorna False:

```
processo.join()  # Aguarda a conclusão do processo
print("Processo finalizado:", not processo.is_alive())  # Saída: True
```

Em casos excepcionais, um processo pode ser finalizado forçadamente com **terminate()** ou **kill()**, mas isso não é recomendado, pois pode deixar recursos alocados sem liberação adequada:

```
if processo.is_alive():
```

```
    processo.terminate()
    print("Processo interrompido forçadamente.")
```

### 1.2.4. Exemplos Simples

Com o que aprendemos até o momento podemos implementar uma função que simula uma tarefa com tempo de espera. O código utiliza uma função sem parâmetros que demora 1 segundo para executar, simulando, por exemplo, um envio de resposta pela rede ou uma tarefa computacionalmente demorada:

```
from multiprocessing import Process
import time

def tarefa_demorada():
    print("Iniciando tarefa...")
    time.sleep(1)
    print("Tarefa concluída.")

if __name__ == "__main__":
    processo1 = Process(target=tarefa_demorada)
    processo2 = Process(target=tarefa_demorada)

    processo1.start()
    processo2.start()

    processo1.join()
    processo2.join()

    print("Todos os processos finalizados.")
```

O método **join()** garante que o processo principal aguarde a conclusão dos processos filhos antes de prosseguir. Sem ele, o código da função pode não ter terminado de rodar, pois o processo principal não espera os filhos.

Por partes, temos:

- **Definição da Função tarefa_demorada**:
    - Esta função imprime uma mensagem inicial, aguarda 1 segundo (time.sleep(1)) e, em seguida, imprime uma mensagem de conclusão. Simula uma tarefa que demanda tempo para ser executada.
- **Bloco Principal (if __name__ == "__main__")**:
    - Este bloco assegura que o código dentro dele seja executado apenas quando o script é executado diretamente, e não quando importado como módulo em outro script. Isso é crucial ao trabalhar com multiprocessing para evitar a criação de processos redundantes.
- **Criação dos Processos:**

    - **processo1 = Process(target=tarefa_demorada)**: Cria uma instância de Process, especificando que a função tarefa_demorada será executada quando o processo for iniciado.
    - **processo2 = Process(target=tarefa_demorada)**: Cria outra instância de Process com o mesmo alvo.
- **Início dos Processos:**
    - **processo1.start()**: Inicia a execução de processo1 em um novo processo.
    - **processo2.start()**: Inicia a execução de processo2 em outro novo processo.
- **Sincronização dos Processos:**
    - **processo1.join()**: O processo principal aguarda a conclusão de processo1 antes de continuar.
    - **processo2.join()**: O processo principal aguarda a conclusão de processo2 antes de prosseguir.
- **Mensagem Final:**
    - Após ambos os processos serem concluídos, a mensagem "Todos os processos finalizados." é exibida.

## 1.2.5. Exemplos Intermediários

### 1.2.5.1. PID e PPID

Quando utilizamos a classe Process, cada instância criada representa um processo separado que possui seu próprio espaço de memória e seu próprio identificador (**PID**). Um ponto interessante é que, dentro do processo filho, é possível consultar o ID do processo pai (**PPID**), o que permite visualizar a hierarquia de execução. Nos exemplos abaixo, mostramos primeiro como dois processos podem ser criados para exibir seus próprios IDs e os do processo pai.

```
from multiprocessing import Process
import os
import time

def mostrar_ids():
    print("Filho: PID =", os.getpid(), "PPID =", os.getppid())
    time.sleep(1)
    print("Filho finalizando: PID =", os.getpid())

if __name__ == "__main__":
    print("Pai: PID =", os.getpid())
    p1 = Process(target=mostrar_ids)
    p2 = Process(target=mostrar_ids)
    p1.start()
```

```
    p2.start()
    p1.join()
    p2.join()
    print("Processo pai finalizado")
```

No exemplo acima, dois processos filhos são iniciados. Cada um imprime seu próprio **PID** e o **PPID** (ID do processo pai), permitindo que vocês visualizem a relação entre eles. Essa técnica é útil para monitorar e depurar a execução de processos paralelos. Observem ainda que conseguimos esses dados pelo módulo de sistemas operacionais do Python (módulo os).

#### 1.2.5.2. CPU-Bound e Medição de Tempo

Um uso interessante dos processos em Python é na utilização de processos CPU-bound. No próximo exemplo, distribuímos o cálculo do fatorial de números distintos entre dois processos. Cada processo recebe um número e, de forma independente, calcula o fatorial usando uma função do módulo math. Esse exemplo evidencia como a divisão de tarefas pode reduzir o tempo de processamento quando há operações intensivas de cálculo.

```
from multiprocessing import Process
import time
import math

def calcular_operacao_intensiva(n):
    for _ in range(100):
        resultado = math.factorial(30000)
    print(f"Processo {n} completou operações intensivas")

if __name__ == "__main__":
    inicio_paralelo = time.perf_counter()

    p1 = Process(target=calcular_operacao_intensiva, args=(1,))
    p2 = Process(target=calcular_operacao_intensiva, args=(2,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    fim_paralelo = time.perf_counter()

    # Versão sequencial
    inicio_sequencial = time.perf_counter()
```

```
    calcular_operacao_intensiva(3)
    calcular_operacao_intensiva(4)

    fim_sequencial = time.perf_counter()

    # Resultados
    print(f"Tempo paralelo: {fim_paralelo - inicio_paralelo:.2f} segundos")
    print(f"Tempo sequencial: {fim_sequencial - inicio_sequencial:.2f}
segundos")
    diferenca = (fim_sequencial - inicio_sequencial) - (fim_paralelo -
inicio_paralelo)
    print(f"Diferença: {diferenca:.2f} segundos a favor do paralelismo")
```

No código acima, a função **calcular_operacao_intensiva(n)** executa um loop que calcula o fatorial de 30.000 cem vezes. Essa repetição foi inserida propositalmente para tornar a operação intensiva o suficiente para que a diferença entre execução sequencial e paralela seja perceptível. Caso o cálculo fosse realizado apenas uma vez, a sobrecarga de criação de processos poderia mascarar os benefícios do paralelismo. Além disso, ao rodar o código várias vezes, é interessante observar que os processos p1 e p2 alternam sua execução, pois o sistema operacional gerencia os processos concorrentes dinamicamente, distribuindo a carga entre os núcleos disponíveis do processador. Esse comportamento pode variar a cada execução, dependendo da política de escalonamento do sistema operacional. No final, ao comparar os tempos de execução sequencial e paralelo, espera-se que a abordagem paralela seja mais rápida, pois divide a carga de trabalho entre múltiplos núcleos, reduzindo o tempo total necessário para a conclusão das operações intensivas.

Dessa forma, as duas primeiras chamadas são distribuídas entre dois processos distintos, permitindo que sejam executadas simultaneamente pelo sistema operacional e reduzindo o tempo total de execução. Já as chamadas subsequentes (3 e 4) ocorrem de maneira sequencial dentro do mesmo processo, o que significa que a segunda só começa após a conclusão da primeira, resultando em um tempo de execução significativamente maior em comparação com a abordagem paralela.

A função **time.perf_counter()** é uma ferramenta de medição de tempo de alta resolução, ideal para benchmarking de trechos de código. Essa função retorna um valor de contagem que inclui tanto o tempo de processamento quanto eventuais pausas do sistema, como o tempo de sono, oferecendo assim uma medição precisa do intervalo decorrido. No exemplo acima, utilizamos **time.perf_counter()** para capturar o tempo imediatamente antes e depois da execução dos processos paralelos e, novamente, para a execução sequencial. Essa abordagem permite comparar diretamente o desempenho das duas abordagens, evidenciando o ganho de tempo obtido com o paralelismo.

Ao incorporar essas medições em seus experimentos, vocês poderão adaptar o método para avaliar a eficiência dos algoritmos implementados, como a versão paralela do Merge Sort. **Basta marcar o tempo no início e no fim da execução do algoritmo para calcular a diferença total**, o que possibilitará a criação de gráficos estatísticos, como boxplots, para uma análise detalhada dos resultados. Essa prática é essencial para identificar gargalos e otimizar o desempenho das aplicações paralelas.

#### 1.2.5.3. Process Pool

Além do uso direto da classe Process, o módulo multiprocessing oferece o objeto Pool, que é extremamente útil para gerenciar um conjunto fixo de processos (os "trabalhadores") e distribuir automaticamente as tarefas entre eles. O uso do Pool reduz a sobrecarga de criar e destruir processos repetidamente, especialmente quando se tem um grande número de tarefas independentes. Ao reutilizar os processos do Pool, o desempenho é otimizado e a implementação torna-se mais simples, pois o Pool cuida do balanceamento das tarefas.

A criação de novos processos envolve um custo significativo de tempo e recursos, tornando-a pouco eficiente para aplicações que exigem execuções frequentes. Nessas situações, em vez de criar e destruir processos repetidamente, é mais vantajoso manter um conjunto fixo de processos em memória, reutilizando-os conforme necessário. É exatamente para esse propósito que existe o Pool de processos, uma estrutura que gerencia múltiplos processos de forma eficiente, distribuindo automaticamente as tarefas entre eles e reduzindo a sobrecarga associada à criação de novos processos.

**O método anterior (apenas utilizando Process) necessita de recursos de IPC que iremos estudar mais a frente, portanto, iremos utilizar o Pool de processos, pois nele podemos retornar dados da função e os recuperar posteriormente.**

No exemplo, usamos o método map para aplicar uma função que calcula o quadrado de cada número em uma lista. Aqui, o Pool é configurado para utilizar dois processos, garantindo que as tarefas sejam distribuídas de forma paralela.

```
from multiprocessing import Pool
import os

def square(n):
    return (os.getpid(), n, n * n)
```

```
if __name__ == "__main__":
    numeros = [1, 2, 3, 4, 5]
    with Pool(2) as pool:
        results = pool.map(square, numeros)
    for pid, n, sq in results:
        print("Processo", pid, "calculou o quadrado de", n, "=", sq)
```

Note que o parâmetro 2 passado para o **Pool** é a quantidade de processos que serão criados e mantidos em memória para serem utilizados na paralelização da tarefa.

#### 1.2.5.4. Process Pool com starmap

O método **starmap** é semelhante ao **map**, mas com uma diferença crucial: ele permite que você passe múltiplos argumentos para a função alvo. Em vez de enviar uma lista de valores individuais, você fornece uma lista de tuplas, onde cada tupla contém os argumentos que serão desempacotados e passados para a função. Isso é extremamente útil quando a função que você deseja aplicar requer mais de um parâmetro, pois elimina a necessidade de criar funções auxiliares para desempacotar os argumentos.

Portanto, utilize **starmap** quando você tem uma função que precisa de vários argumentos e deseja distribuir essas chamadas de forma paralela entre os processos do Pool. Essa abordagem simplifica o código e torna a distribuição das tarefas mais clara e direta.

```
from multiprocessing import Pool
import time

def calcular_potencia(base, expoente):
    return base ** expoente

if __name__ == "__main__":
    # Lista de tuplas onde cada tupla contém (base, expoente)
    argumentos = [(2, 5), (3, 4), (5, 3), (7, 2), (10, 6), (8, 3)]

    with Pool(processes=3) as pool:
        inicio = time.perf_counter()
        resultados = pool.starmap(calcular_potencia, argumentos)
        fim = time.perf_counter()

    print("Resultados:", resultados)
    print(f"Tempo de execução: {fim - inicio:.4f} segundos")
```

Neste exemplo, a função **calcular_potencia** recebe dois parâmetros: base e expoente, e retorna o valor da exponenciação. A lista argumentos contém várias

tuplas com os valores a serem usados em cada chamada. Ao utilizar **pool.starmap(calcular_potencia, argumentos)**, o Pool distribui as tarefas entre três processos. Cada processo desempacota a tupla e chama a função com os argumentos correspondentes. O uso de starmap torna o código mais legível e elimina a necessidade de escrever um loop manual para passar os múltiplos argumentos.

#### 1.2.5.5. Método cpu_count

O método cpu_count é utilizado para determinar o número de núcleos de CPU disponíveis na máquina. Essa informação é fundamental para ajustar dinamicamente o número de processos que você deseja criar no Pool, garantindo que sua aplicação utilize os recursos de hardware de forma eficiente, sem sobrecarregar o sistema.

Ao utilizar **cpu_count**, você pode configurar o Pool para criar exatamente o número de processos que o seu sistema pode suportar, maximizando o paralelismo e evitando a criação excessiva de processos. Isso ajuda a otimizar o desempenho e a eficiência do seu programa, especialmente em aplicações que realizam tarefas intensivas em CPU. (Não teríamos um ganho substancial criando mais processos do que podem ser executados paralelamente pelo nosso hardware)

```python
from multiprocessing import Pool, cpu_count
import time
import math

def calcular_fatorial(n):
    return math.factorial(n)

if __name__ == "__main__":
    # Lista de números para os quais o fatorial será calculado
    numeros = [50000, 40000, 30000, 20000, 10000, 5000]

    # Obtém o número de núcleos disponíveis na CPU
    num_processos = cpu_count()
    print(f"Usando {num_processos} processos")

    with Pool(processes=num_processos) as pool:
        inicio = time.perf_counter()
        resultados = pool.map(calcular_fatorial, numeros)
        fim = time.perf_counter()

    print("Resultados calculados!")
    print(f"Tempo de execução: {fim - inicio:.4f} segundos")
```

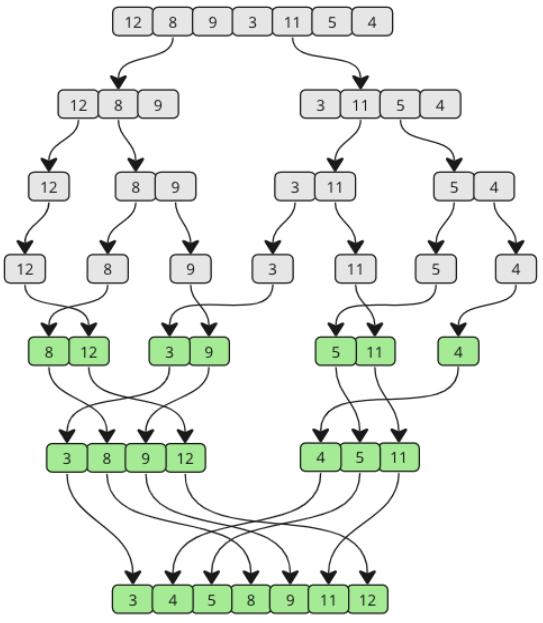
Neste exemplo, a função **calcular_fatorial** utiliza o módulo **math** para calcular o fatorial de um número. A lista de números contém diversos valores grandes, que demandam operações intensivas em CPU. Ao chamar **cpu_count()**, o programa descobre quantos núcleos de CPU estão disponíveis e, em seguida, cria um **Pool** com exatamente esse número de processos. Dessa forma, o trabalho é distribuído de forma otimizada entre todos os núcleos, aproveitando o máximo do paralelismo possível. O método pool.map aplica a função **calcular_fatorial** a cada elemento da lista, e o tempo de execução é medido com **time.perf_counter()** para fornecer uma avaliação precisa do desempenho.

# 2. Tarefa

Com a fundamentação sobre como programar utilizando processos, agora discutiremos a tarefa proposta: implementar duas versões do algoritmo de ordenação Merge Sort, uma versão tradicional (sequencial) e outra paralela (usando processos). Essa abordagem permitirá que vocês explorem a vantagem do paralelismo na ordenação de grandes conjuntos de dados.

## 2.1. Merge Sort

O Merge Sort é um algoritmo de ordenação que segue o paradigma "dividir para conquistar". A lista inicial é sucessivamente dividida em sub-listas menores até que cada sub-lista contenha apenas um elemento (ou esteja vazia), momento em que são consideradas ordenadas por definição. Em seguida, ocorre a etapa de junção (merge), que combina cada par de sub-listas ordenadas em uma lista maior e também ordenada. Esse processo de divisão e junção se repete recursivamente, até que toda a lista seja reconstruída e ordenada.



Na ilustração acima, os blocos em cinza representam as listas resultantes das divisões sucessivas. Quando cada sub-lista atinge tamanho 1, inicia-se a etapa de merge, destacada em verde, na qual duas sub-listas ordenadas são combinadas em uma nova lista ordenada. É possível imaginar esse procedimento como ter duas mãos cheias de cartas ordenadas e ir empilhando, no monte final, sempre a menor carta disponível no topo de cada mão.

Geralmente, o Merge Sort é implementado por meio de recursão. Uma função recursiva recebe uma lista, verifica se ela contém mais de um elemento e, em caso positivo, a divide em duas metades. Em seguida, a própria função é chamada para cada metade, garantindo que cada parte seja ordenada separadamente. Ao final, as duas sub-listas ordenadas são unidas por meio de uma função de merge, resultando em uma lista única e ordenada.

Abaixo está um pseudocódigo que descreve o funcionamento básico do Merge Sort:

```
função MergeSort(A):
    se comprimento(A) <= 1:
        retorne A
    senão:
        meio = comprimento(A) // 2
        esquerda = MergeSort(A[0...meio-1])
        direita = MergeSort(A[meio...fim])
        retorne Merge(esquerda, direita)

função Merge(esquerda, direita):
    resultado = lista vazia
    enquanto esquerda não estiver vazia E direita não estiver vazia:
        se primeiro elemento de esquerda <= primeiro elemento de direita:
            adicionar primeiro elemento de esquerda em resultado
            remover primeiro elemento de esquerda
        senão:
            adicionar primeiro elemento de direita em resultado
            remover primeiro elemento de direita
    adicionar todos os elementos restantes de esquerda em resultado
    adicionar todos os elementos restantes de direita em resultado
    retorne resultado
```

Para uma visualização interativa do algoritmo, recomendo visitar o site w3schools. Ali, vocês poderão acompanhar passo a passo como o Merge Sort divide uma lista em sub-listas e, posteriormente, as combina em um processo iterativo até obter a lista completamente ordenada. Além disso, um vídeo que pode ajudá-los é Programação Dinâmica.

## 2.2. Box Plot

Um box plot, ou diagrama de caixa, é uma representação gráfica que resume a distribuição de um conjunto de dados numéricos, destacando suas principais características estatísticas. Ele é amplamente utilizado na análise exploratória de dados para visualizar a dispersão e identificar possíveis outliers (valores atípicos).

**Componentes de um Box Plot:**

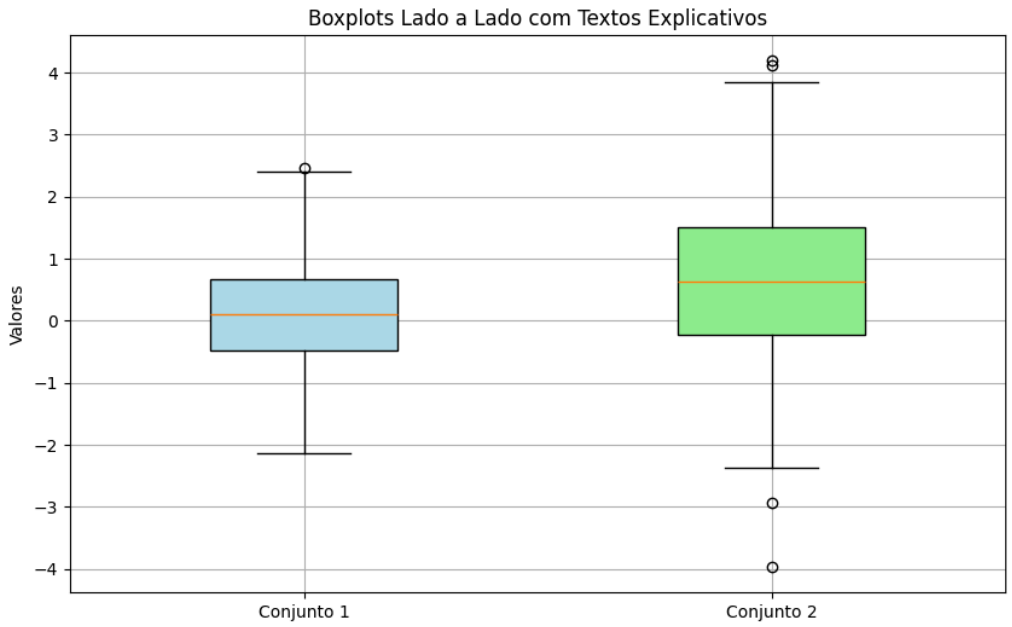
- **Primeiro Quartil (Q1)**: Também conhecido como quartil inferior, representa o ponto do qual 25% dos dados se encontram.
- **Mediana (Q2)**: O valor central que divide o conjunto de dados em duas metades iguais, com 50% dos dados abaixo e 50% acima.
- **Terceiro Quartil (Q3)**: Conhecido como quartil superior, é o ponto do qual 75% dos dados se encontram.
- **Amplitude Interquartil (IQR):** É a diferença entre o terceiro e o primeiro quartil (IQR = Q3 - Q1) e representa a dispersão dos 50% centrais dos dados.
- **Limites (Whiskers)**: São linhas que se estendem a partir da caixa até os valores máximos e mínimos dentro de 1,5 vezes a amplitude interquartil a partir dos quartis. Valores além desses limites são considerados outliers.
- **Outliers:** São pontos de dados que se situam além dos limites dos whiskers e podem indicar variabilidade excepcional ou possíveis erros nos dados.

Para ilustrar a construção de um box plot em Python, utilizaremos a biblioteca matplotlib, que oferece ferramentas robustas para visualização de dados.

```
import matplotlib.pyplot as plt
import numpy as np

# Gerando dados de exemplo
np.random.seed(10)
dados1 = np.random.normal(0, 1, 100)
dados2 = np.random.normal(0.5, 1.5, 100)

# Criando a figura e os eixos
fig, ax = plt.subplots(figsize=(10, 6))
# Criando os boxplots lado a lado
boxplots = ax.boxplot([dados1, dados2], patch_artist=True, widths=0.4)

# Personalizando as cores das caixas
cores = ['lightblue', 'lightgreen']
for patch, cor in zip(boxplots['boxes'], cores):
   patch.set_facecolor(cor)

# Configurando os rótulos dos eixos
ax.set_xticklabels(['Conjunto 1', 'Conjunto 2'])
```

```
ax.set_ylabel('Valores')
ax.set_title('Boxplots Lado a Lado com Textos Explicativos')
# Exibindo o gráfico
plt.grid(True)
plt.show()
```

Este código gera um gráfico com dois boxplots posicionados lado a lado. Essa visualização facilita a comparação entre os conjuntos de dados e destaca informações estatísticas relevantes diretamente no gráfico, nos permitindo observar não apenas o tempo médio que nossas execuções do merge sort terão mas também a dispersão que nossas implementações tiveram. Ao executar o código, será exibido uma figura como abaixo.




O box plot é uma ferramenta poderosa para compreender rapidamente a distribuição de dados, identificar assimetrias e detectar valores atípicos, sendo essencial na análise estatística e na visualização de dados.

**Essa explicação é bem sucinta, para que vocês possam entender melhor o gráfico, peço que leia o artigo da professora Fernanda.**

**Note que os dados1 seria uma lista com os tempos de seu algoritmo sequencial e dados2 seria os tempos para o seu algoritmo paralelo.**

## 2.3. Tarefa

Juntamente com este documento, você encontrará um arquivo **zip** contendo duas pastas com cinco arquivos de teste cada. Cada arquivo consiste em um número em

cada linha. Sua tarefa é ler cada arquivo, carregar os números em uma lista (ou vetor) e aplicar duas versões do algoritmo Merge Sort:

- **Versão Sequencial (Padrão)**: Implementação tradicional do Merge Sort.
- **Versão Paralela**: Implementação do Merge Sort utilizando processamento paralelo.

Para cada conjunto de dados, você deve executar ambas as versões do algoritmo pelo menos dez vezes. O tempo de execução de cada rodada deve ser medido e armazenado para posterior análise estatística. A comparação entre as duas abordagens será feita por meio de diagramas de box plot, que permitem visualizar a distribuição dos tempos de execução, identificando tendências e variações de desempenho. Para isso, é necessário gerar cinco box plots para cada versão do algoritmo, totalizando dez gráficos. Cada conjunto de gráficos deve seguir um padrão de cores consistente, garantindo que seja possível identificar claramente as diferenças entre as execuções do Merge Sort sequencial e paralelo. A única distinção entre os gráficos será nos rótulos do eixo X, que devem indicar a identificação de cada execução.

Após a geração dos gráficos, você deve elaborar um relatório em formato PDF contendo o código Python utilizado, devidamente comentado, as figuras dos box plots e uma análise detalhada dos resultados. O relatório deve discutir o desempenho de cada versão do algoritmo, as diferenças observadas entre elas e as possíveis razões para variações nos tempos de execução. Questões como a sobrecarga da criação de processos, o impacto da paralelização em diferentes tamanhos de entrada e a eficiência do uso de múltiplos núcleos devem ser abordadas na sua análise.

**Você deverá fazer esse procedimento para as duas pastas, listas_pequenas e listas_grandes, gerando duas imagens de box plots, discuta sobre a diferença sobre as duas.**

Aqui eu forneço um exemplo das figuras esperadas como resposta, em ambas as imagens, os cinco primeiros box plots são referentes a versão sequencial e as cinco últimas em relação às versões paralelas:

Figura para listas pequenas:

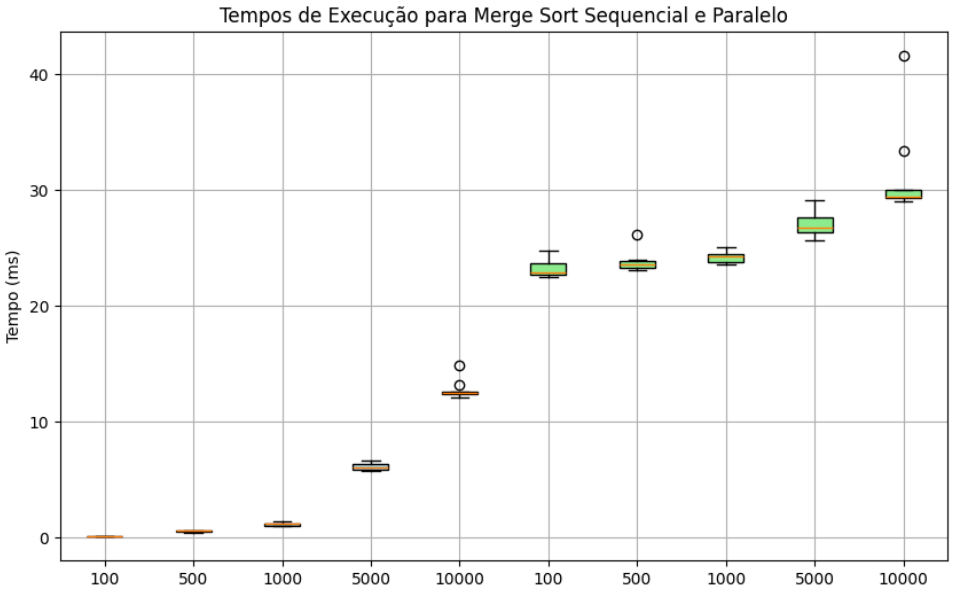
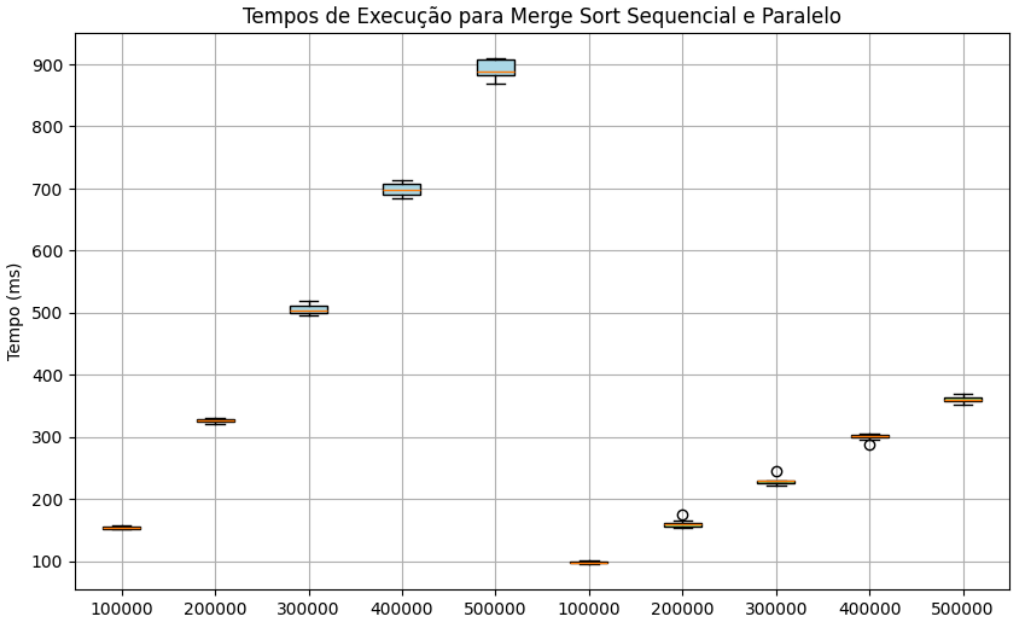



Figura para listas grandes:




**Note que a implementação sequencial tem o tempo menor do que a paralelo para listas menores, mas não para as maiores. Por que isso ocorre? Discuta na sua resposta.**

**Uma dica importante:** não tentem criar um processo para cada divisão do Merge Sort, pois o sistema operacional não consegue lidar com tantos processos simultaneamente, especialmente para os arquivos grandes. Para paralelizar o

algoritmo, **divida inicialmente os elementos em partes iguais, de acordo com o número de CPUs do seu computador**. Assim, você terá N listas ordenadas, que podem ser combinadas utilizando a própria função de merge que você implementou. Essa junção também pode ser feita de forma paralela. Tanto a separação quanto a junção podem ser realizadas com um **Pool de processos, onde o tamanho do Pool corresponde ao número de CPUs disponíveis**.

### 2.4. Pontuação

Você deverá colocar seu código python no arquivo PDF e comentá-lo (explicar cada função e cada método no próprio arquivo PDF, em parágrafos). Além disso, coloque ambas figuras e as discuta.

A pontuação será toda dada nesse arquivo PDF, que será feito de forma **individual**, devendo ser entregue no dia **26/03/2025**, para cada dia de atraso terá 10% da nota reduzida até que o trabalho seja zerado.

# Referências


PYKES, K. Python Multiprocessing: A Guide to Threads and Processes. Disponível em: <https://www.datacamp.com/tutorial/python-multiprocessing-tutorial>.

W3SCHOOLS. DSA Merge Sort. Disponível em: <https://www.w3schools.com/dsa/dsa_algo_mergesort.php>.

AVILA, P. Algoritmos e Programação de Computadores Ordenação: Merge Sort. [s.l: s.n.]. Disponível em: <https://www.ic.unicamp.br/~sandra/pdf/class/2019-1/mc102/2019-06-17-MC102KLMN-Aula27.pdf>.

W3SCHOOLS. DSA Merge Sort. Disponível em: <https://www.w3schools.com/dsa/dsa_algo_mergesort.php>.

SUPER FAST PYTHON. Python Multiprocessing: 7-Day Crash Course - Super Fast Python - Medium. Disponível em: <https://medium.com/@superfastpython/python-multiprocessing-7-day-crash-course-6edb73e8ae2f>.

GEEKSFORGEEKS. Multiprocessing in Python | Set 2 (Communication between processes). Disponível em: <https://www.geeksforgeeks.org/multiprocessing-python-set-2/>.

Multiprocessing in Python | Set 1 (Introduction). Disponível em: <https://www.geeksforgeeks.org/multiprocessing-python-set-1/>.