PDF Novel Converter

Fernanda da Silva Freire
Mariana Fernandes Cabral

Relatório Final

Programação Concorrente (ICP-361) — 2023/1

1. Descrição do problema

O problema escolhido consiste em criar um código que transforme os interesses literários dos usuários que estão disponibilizados de forma online em sites de leitura em um formato compatível com e-readers, como PDF.

A solução sequencial envolve a leitura de um arquivo de texto com uma lista de URLs que representam os livros e novels desejados. Essas URLs são utilizadas para realizar web scraping em cada link, obtendo os sublinks correspondentes aos capítulos do livro ou da novel. Cada capítulo é armazenado como arquivo de HTML em uma pasta dedicada ao livro e, ao fim do processamento, é feita a junção e a conversão dos capítulos em um único arquivo PDF. Após essa etapa, o cache dos capítulos é apagado. Em suma, é necessário que o usuário passe como entrada o arquivo de texto com a lista de URLs dos livros e das novels e, como saída, é esperado obter um PDF de cada link informado.

No entanto, no caso do programa sequencial, a alta demanda imposta ao web server pode levar ao bloqueio do IP de origem. Para evitar isso, é necessário adicionar um atraso de 5 segundos entre as leituras de capítulos para não ser considerado malicioso. Esse atraso torna o código normal extremamente lento.

Uma solução concorrente pode melhorar significativamente o desempenho do código, permitindo que vários processos sejam executados simultaneamente. Mesmo com o atraso de 5 segundos entre os processos, eles podem iniciar e executar de forma concorrente, possibilitando a requisição e a leitura de mais de um capítulo ao mesmo tempo. Assim, enquanto se espera a resposta do site para as requisições, é possível fazer um grande volume de novas requisições, de modo a já ter novas para processar quando o servidor terminar de processar outras, para então atingir mais rapidamente o limite de requisições que o servidor não considera malicioso, aguardar o delay de 5 segundos e repetir todo o processo até terminar a tarefa. Isso reduzirá o tempo total gasto, pois vários capítulos podem ser lidos e salvos paralelamente e isso será feito de forma mais rápida e otimizada, já que não será

preciso a resposta da solicitação ao site para fazer novas requisições.

Em síntese, o ganho de desempenho é gerado devido ao fato de ser um programa de I/O bound, que envolve um grande volume de operações de entrada e saída, com tarefas que possuem certo grau de independência entre si, o que viabiliza a implementação paralela de certas seções do algoritmo. Essas etapas podem ser subdivididas entre threads ou processos, permitindo que várias requisições ao servidor possam ser feitas enquanto ainda aguarda o retorno dos dados solicitados ao site em vez de realizar apenas uma por vez, precisando esperar pela resposta do site antes de realizar outra requisição, como seria no caso sequencial.

2. Projeto da solução concorrente

Nesse sentido, tem-se como problema central transformar todo o livro ou novel disponibilizado na internet em um arquivo no formato PDF. As subtarefas divididas entre os processos corresponderão à leitura e armazenamento dos capítulos.

Para a implementação do projeto, foi decidido fazer o código em Python, visto que é a melhor linguagem para web scraping, contando com uma vasta gama de bibliotecas para isso. Como principais ferramentas, escolheu-se utilizar a biblioteca nativa requests junto a BeautifulSoup.

Foi optado por usar multiprocessing no lugar de multithreading por ser capaz de fazer mais requisições ao servidor em relação à multithreading em virtude da escolha de Python para a implementação. Devido ao Global Interpreter Lock (GIL) do Python, que impede duas ou mais threads de executarem ao mesmo tempo, apenas uma thread de cada vez é executada e faz as requisições ao site com o uso de multithreading. Por outro lado, no multiprocessing, é possível que todos os processadores da máquina executem os processos e façam as requisições simultaneamente, ou seja, com paralelismo real. Dessa maneira, chega-se ao limite de requisições no qual o servidor não considera malicioso de forma mais rápida em relação ao multithreading, fazendo um grande volume de uma só vez, e espera o delay imposto no código antes de já poder fazer outro novo lote de requisições concorrentemente. Por fim, vale pontuar que as tarefas que serão implementadas com concorrência possuem certo grau de independência entre si, então não há necessidade de ter a memória compartilhada das threads, exceto para gerar o resultado final, que demanda certa sincronização entre os processos.

Para a solução concorrente do problema, as duas melhores estratégias de divisão estática da tarefa principal entre fluxos de execução independentes são o agrupamento e a alternância do salvamento de capítulos. Como o processamento dos capítulos é basicamente o mesmo, não há necessidade de implementar uma divisão de tarefas dinâmica.

No agrupamento, os processos são divididos em conjuntos consecutivos de capítulos, de modo que cada processo responsável por baixar e processar um bloco específico de capítulos. Por exemplo, o processador 1 pode baixar os capítulos 1, 2, 3, 4 e 5, enquanto o processador 2 pode baixar os capítulos 6, 7, 8, 9 e 10.

Na alternância, os processos alternam entre os capítulos a serem baixados. Por exemplo, o processo 1 pode baixar o capítulo 1, 3 e 5, enquanto o processo 2 baixa o capítulo 2, 4 e 6, e assim por diante. Não é necessário seguir uma ordem sequencial estrita, pois os processos podem estar em filas de processadores diferentes.

As duas estratégias de divisão de tarefa estática oferecem uma distribuição semelhante de capítulos entre os processos, permitindo um melhor balanceamento de carga. Isso evita a ocorrência de um único processo ser responsável por uma grande quantidade de capítulos, o que poderia causar atrasos significativos.

Considerando ambas acima, a forma alternada é a melhor para se implementar neste programa, visto que o webserver iria conseguir receber uma determinada quantidade de requisições que é capaz de processar. Assim, é obtido um maior ganho de desempenho, uma vez que ele não teria sempre um intervalo de 5 segundos entre capítulos ou threads.

O projeto está dividido em várias etapas.

Algoritmo do PDF Novel Converter:

- **Passo 1:** Abrir o arquivo de texto com as URLs das novels desejadas e criar diretório para cada novel
- **Passo 2:** Abrir todos os capítulos de cada novel por meio de sua URL e fazer a leitura e escrita do HTML de cada um deles no diretório correspondente
- **Passo 3:** Juntar todos os HTMLs associados aos capítulos de uma mesma novel
- **Passo 4:** Transformar o arquivo HTML em PDF

Execução do algoritmo sequencial

1 novel de 145 capítulos

Passo executado	Tempo de execução
1	1.3565669159753418
2	**852.1488165855408**
3	0.44127964973449707
4	84.90272617340088

3 novels

Passo executado	Tempo de execução
1	
2	
3	
4	
5	
6	

Tendo em vista os resultados acima obtidos na saída do programa, observa-se que o passo 2, correspondente à leitura e escrita do HTML de cada capítulo, é o que possui maior tempo de execução, principalmente considerando o fato que se trata da seção do algoritmo mais carregada de I/O bound, ou seja, permanece em estado de espera por um longo período durante a comunicação com o servidor.

A tarefa principal de ler e escrever o HTML de todos os capítulos das novels pode ser facilmente dividida em subgrupos de capítulos e novels, visto que as operações feitas em cada capítulo não possuem dependência entre si. Dessa forma, foi possível notar a possibilidade de implementar a concorrência neste algoritmo.

Segue o pseudocódigo do algoritmo sequencial.

```
função converter_html_para_pdf(nome_arquivo_html,
nome_arquivo_pdf):
    // Converter arquivo HTML para PDF


função verificarLista():
    novelsABaixar = []
    // Abrir arquivo 'novelLinks.txt'
    // Para cada linha 'novel' no arquivo:
        // Adicionar 'novel' à lista novelsABaixar
    retornar novelsABaixar


início = registrar tempo atual
novels = verificarLista()
createdPaths = []


para cada novel em novels:
    // Extrair informações da URL da novel


      // Criar diretório com o nome da novel, se ainda não
existir
    // Adicionar nome da novel à lista createdPaths
```

```
    para cada capítulo em range(1, qnt[0]):
        // Fazer requisição GET para a URL do capítulo
        // Analisar o conteúdo da página

        // Criar nome do arquivo HTML do capítulo
        // Escrever o conteúdo no arquivo HTML
        // Aguardar 5 segundos

conteudo = ""
para cada path em createdPaths:
    para cada nome_arquivo em arquivos do path:
         // Ler conteúdo de cada arquivo no path e adicionar a
'conteudo'

    // Criar nome do arquivo de conclusão do path
    // Escrever conteúdo no arquivo HTML de conclusão

    para cada nome_arquivo em arquivos do path:
        // Remover cada arquivo no path

               converter_html_para_pdf(nome_arquivo_conclusao,
nome_arquivo_pdf)

exibir tempo de execução
```

Em relação às mudanças de decisão que foram feitas no projeto, pode-se citar a conversão para PDF em vez de ePUB, pois será melhor para visualização na apresentação do trabalho. Além disso, optou-se por implementar multiprocessing em vez de multithreading, pois no momento do desenvolvimento do relatório parcial ainda não se tinha conhecimento a respeito de como funcionava a concorrência em Python, além de outros fatores já explicitados sobre as vantagens do multiprocessing em relação ao multithreading. Por fim, foi decidido também salvar os capítulos em HTML em vez de transformá-los em arquivos de texto para evitar um processamento desnecessário.

3. Testes de corretude

Para verificar a corretude do código, irá criptografar em sha256 ambas saídas .html: concorrente e sequencial. Após isso, será verificado se todos os itens da lista estão idênticos. Caso sim, foi verificado que o código foi executado corretamente, uma vez que qualquer mudança realizada geraria um sha256 diferente.

O teste de corretude do programa consiste em verificar se a execução ocorreu de forma precisa e se as saídas geradas são corretas. Para isso, é adotada uma abordagem que envolve a geração de arquivos HTML em paralelo e a comparação de seus hashes com os hashes dos arquivos HTML gerados pelo programa principal.

A etapa de geração de arquivos HTML ocorre de forma concorrente, onde múltiplas instâncias do programa são executadas simultaneamente. Cada instância é responsável por gerar um arquivo HTML específico. Esses arquivos contêm informações e estruturas HTML que são produzidas pelo programa.

Para verificar a corretude do programa, é necessário salvar os arquivos HTML gerados pelas instâncias concorrentes. Isso é feito através da obtenção do código sequencial desses arquivos, ou seja, a serialização do conteúdo dos arquivos em uma sequência de bytes.

Em seguida, é aplicada uma função de hash, como por exemplo, a função SHA-256, nos arquivos HTML gerados pelas instâncias concorrentes, bem como nos arquivos HTML gerados pelo programa principal. A função de hash é responsável por produzir uma sequência de caracteres única que representa o conteúdo do arquivo.

Comparando os hashes gerados pelas instâncias concorrentes com os hashes dos arquivos gerados pelo programa principal, é possível verificar se as saídas são consistentes. Se todos os hashes coincidirem, isso indica que o programa está executando perfeitamente e gerando saídas corretas.

Esse teste de corretude é fundamental para garantir a confiabilidade do programa, uma vez que assegura que todas as instâncias concorrentes estão produzindo resultados consistentes e em conformidade com o programa principal.

4. Avaliação de desempenho

Para atestar a melhoria de desempenho na solução concorrente, é feita a contagem do tempo de processamento na parte paralela e o total gasto ao fim, possibilitando a comparação com o tempo de execução total gasto com o algoritmo sequencial.

Será feita variação na dimensão dos dados de entrada, de modo a testar diferentes tamanhos de listas de URLs, representando diferentes quantidades de livros ou novels com variados números de capítulos a serem processados. Isso ajudará a garantir que a solução lide corretamente com diferentes volumes de trabalho, além de permitir analisar o impacto do tamanho da entrada no tempo de processamento. Também será feita variação no número de processos para que sejam testadas diferentes configurações de quantidade de processos para avaliar a performance da solução concorrente em diferentes cenários de concorrência, o que ajudará a identificar o ponto ótimo em termos de número de processos para obter o melhor desempenho.

O objetivo do teste de desempenho é verificar a eficiência da solução concorrente em termos de tempo de processamento e ganho de desempenho em comparação com a solução sequencial. Os resultados esperados são tempos de execução menores e um aumento no desempenho à medida que mais processos são utilizados, pelo menos até a quantidade de processos considerada ótima, na qual obtém-se a melhor performance.

Foram feitos os seguintes casos de testes:

- 1 novel e 2 processo
- 1 novel e 3 processos
- 1 novel e 5 processos
- 1 novel e 6 processos

- 1 novel e 10 processos

- 3 novels e 1 processo
- 3 novels e 2 processos
- 3 novels e 6 processos
- 3 novels e 10 processos

Com isso, testa-se a eficiência do programa para uma quantidade de novels pequena, média e grande, além de ser possível verificar e comparar a performance usando diferentes quantidades de processos. A máquina onde os testes foram realizados tem como processador com 4 núcleos de execução, 4GB RAM e usa o sistema operacional Manjaro XSCE baseado em ArchLinux.

Cada caso de teste foi executado 5 vezes com o tempo medido em segundos. Para analisar o desempenho, foi utilizado o menor tempo obtido dos testes.

Para analisar o ganho da implementação de concorrência do algoritmo, pode-se utilizar a Lei de Amdahl para calcular o ganho teórico e compará-lo com o que realmente foi obtido na execução do programa paralelo.

Dessa maneira, tem-se que o **ganho previsto** corresponde a:

$$\frac{Tempo\ total\ de\ execução\ do\ programa\ sequencial}{Tempo\ total\ de\ execução\ do\ programa\ concorrente}$$

Antes, deve-se calcular o **tempo total de execução do programa concorrente**, obtido com:

$$tempo\ da\ parte\ sequencial\ + \frac{tempo\ da\ parte\ concorrente}{quantidade\ de\ processadores}$$

Execução do algoritmo sequencial

1 novel de 145 capítulos

Passo executado	Tempo de execução
1	1.3565669159753418
2	**852.1488165855408**
3	0.44127964973449707
4	84.90272617340088

Considerando a execução acima, tem-se pela lei de Amdahl:

$$tempo\ da\ parte\ sequencial\ + \frac{tempo\ da\ parte\ concorrente}{quantidade\ de\ processadores}$$

Tempo da parte sequencial

$$Ts = 1.3565669159753418 + 0.44127964973449707 + 84.90272617340088$$

$$Ts = 86.7005727367\ segundos$$

Tempo total de execução do programa concorrente

Com 2 processos:

$$86.7005727367 + \frac{852.1488165855408}{2} = 512.774981029\ segundos$$

Com 3 processos:

$$86.7005727367 + \frac{852.1488165855408}{3} = 370.750178265\ segundos$$

Com 5 processos:

$$86.7005727367 + \frac{852.1488165855408}{6} = 257.130336054\ segundos$$

Com 6 processos:

$$86.7005727367 + \frac{852.1488165855408}{6} = 228.725375501\ segundos$$

Com 10 processos:

$$86.7005727367 + \frac{852.1488165855408}{10} = 171.915454395\ segundos$$

Ganho teórico estimado

Com 2 processos:

$$\frac{938.849389325}{512.774981029} = 1.83091887096$$

Com 3 processos:

$$\frac{938.849389325}{370.750178265} = 2.5322965284$$

Com 5 processos:

$$\frac{938.849389325}{257.130336054} = 3.65125874968$$

Com 6 processos:

$$\frac{938.849389325}{228.725375501} = 4.10470148871$$

Com 10 processos:

$$\frac{938.849389325}{171.915454395} = 5.46111105966$$

Agora, resta descobrir o ganho real obtido com a implementação da concorrência.

Abaixo, segue o tempo médio total de execução do programa sequencial e do programa concorrente.

Programa Sequencial

Quantidade de novels/URLs	Total de capítulos	Tempo médio total de execução do programa sequencial
1	98	669.6921696662903
1	145	875.6444160938263
1	181	1070.268844127655
2	279	1763.5171763896942

Programa Concorrente

1 novel de 145 capítulos

Quantidade de processos	Tempo de execução
2	456.3934259414673
3	315.52629232406616

5	203.54458045959473
6	226.01588916778564
10	118.60294938087463

3 novels totalizando 399 capítulos

Quantidade de processos	Tempo de execução
2	1307.6703660488127
3	1144.5494606494904
6	747.7752003669739

Com esses valores obtidos na saída do programa, é possível calcular o ganho real com a concorrência.

Com 2 processos:

$$\frac{938.849389325}{456.3934259414673} = 2.05710541818$$

Com 3 processos:

$$\frac{938.849389325}{315.52629232406616} = 2.97550286438$$

Com 5 processos:

$$\frac{938.849389325}{203.54458045959473} = 4.61250005873$$

Com 6 processos:

$$\frac{938.849389325}{226.01588916778564} = 4.15390879279$$

Com 10 processos:

$$\frac{938.849389325}{118.60294938087463} = 7.91590254902$$

Quantidade de processos	Ganho teórico estimado	Ganho real
2	1.83091887096	2.05710541818
3	2.5322965284	2.97550286438
5	3.65125874968	4.61250005873
6	4.10470148871	4.15390879279
10	5.46111105966	7.91590254902

Nota-se que, em geral, o ganho real foi maior que o ganho teórico estimado e isso se deve à grande otimização na seção de I/O bound implementada no código concorrente, que gerou uma melhoria maior que a esperada.

Vale pontuar que apesar de haver esse suposto grande ganho com 10 processos, o consumo passa a não valer a pena a partir de 7 processos porque a máquina passa a não suportar todo o alcance necessário, apresentando travamentos.

5. Discussão

Com todos os fatos expostos, é possível concluir que há ganho de desempenho na implementação do programa concorrente em relação ao sequencial. Esse ganho é esperado, pois, como já justificado anteriormente, um grande volume de requisições é feito ao servidor de uma só vez enquanto se aguarda a resposta do site, atingindo mais rapidamente o limite de requisições que o servidor não considera malicioso, e, com isso, torna o algoritmo mais veloz por conta do uso de concorrência com multiprocessing, que evoca processos filhos que dividem entre si a tarefa principal da leitura e escrita dos capítulos das novels e podem executar simultaneamente nos diferentes processadores da máquina.

Além disso, por se tratar de um programa de I/O bound, que envolve um grande volume de operações de entrada e saída, e executar tarefas que possuem certo grau de independência entre si, como é o caso de ler e escrever cada capítulo separadamente, que viabiliza a implementação paralela de certas seções do algoritmo, a possibilidade de realizar novas requisições concorrentemente enquanto se espera pela resposta do servidor faz com que se tenha um melhor aproveitamento e desempenho ao maximizar a realização de solicitações em determinado período e minimizar o tempo de espera por novas solicitações, já que não será preciso aguardar a resposta do servidor para realizar outras requisições. Portanto, conclui-se que a implementação da concorrência do PDF Novel Converter é extremamente vantajosa e útil na prática.

Apesar da melhoria de desempenho em relação ao código sequencial, o programa concorrente ainda pode ser mais aperfeiçoado. Algumas das melhorias possíveis seriam verificar alternativas para aumentar a quantidade de requisições feitas a servidores e expandir o número de servidores permitidos. Embora não seja recomendável, é possível baixar mais que duas novels com o código atual, porém, com essa sofisticação, possibilitaria realizar essa tarefa mais adequadamente.

Vale destacar que diversos problemas surgiram durante o desenvolvimento do PDF Novel Converter.

Primeiramente, como não se tinha um grande conhecimento inicial de como aplicar concorrência em Python, foi testado o uso de multithreading na aplicação. Nesse teste, foi observável que havia problema com velocidade e com o desempenho da máquina, visto que era feito não só requisições web, mas também web scraping. Assim, descobriu-se que a implementação de multithreading não faria o melhor rendimento possível com concorrência no caso deste algoritmo.

Inicialmente, o sistema operacional Windows da máquina de teste não suportou a execução do código, então foi necessário usar outro ambiente alternativo — a máquina virtual Manjaro

— para realizar os testes de execução e averiguar a performance do algoritmo.

Houve dificuldade no teste de corretude, no qual foi utilizado hash para realizar a comparação entre a saída e o texto original e não estava obtendo hashs iguais, porém isso foi facilmente corrigido com uma linha de código que foi esquecida.

Em busca de melhorar ainda mais a performance do PDF Novel Converter, testou-se a aplicação de multiprocessing na parte de conversão de HTML para PDF para converter todas as novels passadas paralelamente. No entanto, o overhead e o tempo gasto não foram suficientes para manter essa ideia, pois tornou o programa ainda mais custoso e lento.

5. Referências Bibliográficas

Repositório – PDF Novel Converter

Documentação do Python referente a multiprocessing

I/O bound

Multiprocessing em Python 1

Multiprocessing em Python 2

Multiprocessing em Python 3

Hash em Python 1

Hash em Python 2

Weasyprint

Web Scraping — Beautiful Soup