Universidade Federal de Santa Catarina Centro Tecnológico - CTC
Departamento de Informática e Estatística - INE

Relatório

Programação Concorrente

Profs.
Giovani Gracioli e
Márcio Castro

Alunos
Augusto de Hollanda Vieira Guerner (22102192),
Eduardo Gwoszdz De Lazari (22100612) e
Micael Angelo Sabadin Presotto (22104063)

Problema

O problema consiste em ver a validade de sudokus implementando métodos de concorrência na linguagem Python. Para saber se eles são válidos, todas suas linhas, colunas e regiões (3x3) devem conter números de 1-9, sem poder repeti-los.

Código

Para resolver o problema, foram implementados processos e threads para cuidar da parte da concorrência, em que os processos seriam responsáveis por um ou mais sudokus, e cada processo teria uma ou mais threads para validar o sudoku.

A separação dos sudokus foi feita utilizando um método que consiste em combinar a divisão inteira com o resto da divisão. Dessa forma, todos os processos receberão um intervalo de tamanho igual a divisão inteira, e o resto da divisão será repartido entre as threads, fazendo com que algumas tenham um intervalo de no máximo 1 sudoku a mais. Por exemplo, se houvesse 10 sudokus para 3 processos, a divisão seria: 1º processo [0,4), 2º processo [4,7) e 3º processo [7,11). Percebe-se que o resto da divisão é igual a 1, logo foi incrementado apenas o trecho do 1º em 1.

Após isso, foram implementadas três funções, onde o papel de cada uma era retornar todas linhas, colunas e regiões do sudoku, respectivamente. Esses retornos, que consistiam em listas de 9 inteiros, apelidamos de “blocos”, e todos os blocos deveriam ser repartidos igualmente entre as threads. Para isso, foi utilizado o mesmo método implementado para os processos.

```python
def get_lines(sudoku):
    return [[f"L{i + 1}", *line] for i, line in enumerate(
sudoku)]

def get_columns(sudoku):
    t = [[sudoku[l][c] for l in range(9)] for c in range(9)]
    return [[f"C{i + 1}", *col] for i, col in enumerate(t)]

def get_regions(sudoku):
    regions = [[f"R{i + 1}", ] for i in range(9)]
    for r in range(9):
        for l in range((r // 3) * 3, (r // 3) * 3 + 3):
            for c in range((r % 3) * 3, (r % 3) * 3 + 3):
                regions[r].append(sudoku[l][c])
    return regions[:]
```

Trecho de código responsável por retornar todos os possíveis blocos da matriz (linhas, colunas e regiões)

Assim, cada thread era responsável por N blocos do sudoku, verificando assim, se estavam corretos ou não, e retornando seus respectivos erros. Para isso, foi utilizada uma técnica de transformar cada bloco em um conjunto, pois, um conjunto não repete elementos e a ordem dos mesmos é irrelevante. Assim, comparamos cada bloco com o conjunto {1,2,3,4,5,6,7,8,9}, para validar a repetição de elementos, desconsiderando sua ordem. Por exemplo, se pegarmos uma linha de um sudoku que tenha a seguinte configuração: [1,4,5,9,8,2,3,7,4], a função set irá tirar as repetições dessa lista, tornando-a {1,2,3,4,5,7,8,9}. Dessa maneira, a linha será diferente de {1,2,3,4,5,6,7,8,9}, resultando em erro.

```
exemplo_block_correto = [4,3,6,1,5,2,9,8,7]
exemplo_block_incorreto = [4,3,6,1,5,2,2,8,7]

set_exemplo_block_correto = set(exemplo_block_correto) ⇒ {4,3,6,1,5,2,9,8,7}
set_exemplo_block_incorreto = set(exemplo_block_incorreto) ⇒ {4,3,6,1,5,2,8,7} #2 foi deletado

set_exemplo_block_correto == {1,2,3,4,5,6,7,8,9} ⇒ True
set_exemplo_block_incorreto == {1,2,3,4,5,6,7,8,9} ⇒ False
```

Exemplo de código que representa a validação de uma linha coluna e/ou região

Diferentemente da linguagem C, as threads em python não possuem um retorno. Portanto, tivemos que criar uma lista com um índice único para cada thread, onde nela serão adicionados os erros computados. Após todos os blocos serem calculados, o processo exibe os erros captados por cada thread, na ordem de Linha, Coluna e Região.

Para fins comparativos, foi implementado um código sequencial, sem usar métodos de threads e processos. Seu tempo será usado para calcular o speedup relativo ao concorrente.

Resultados

Os seguintes gráficos foram gerados em um computador com um processador Intel I3-12100F(3300 Mhz, 4 núcleos e 8 processadores) e 16gb ram. Foram feitas combinações de 1 até 12 processos com 1 até 8 threads.

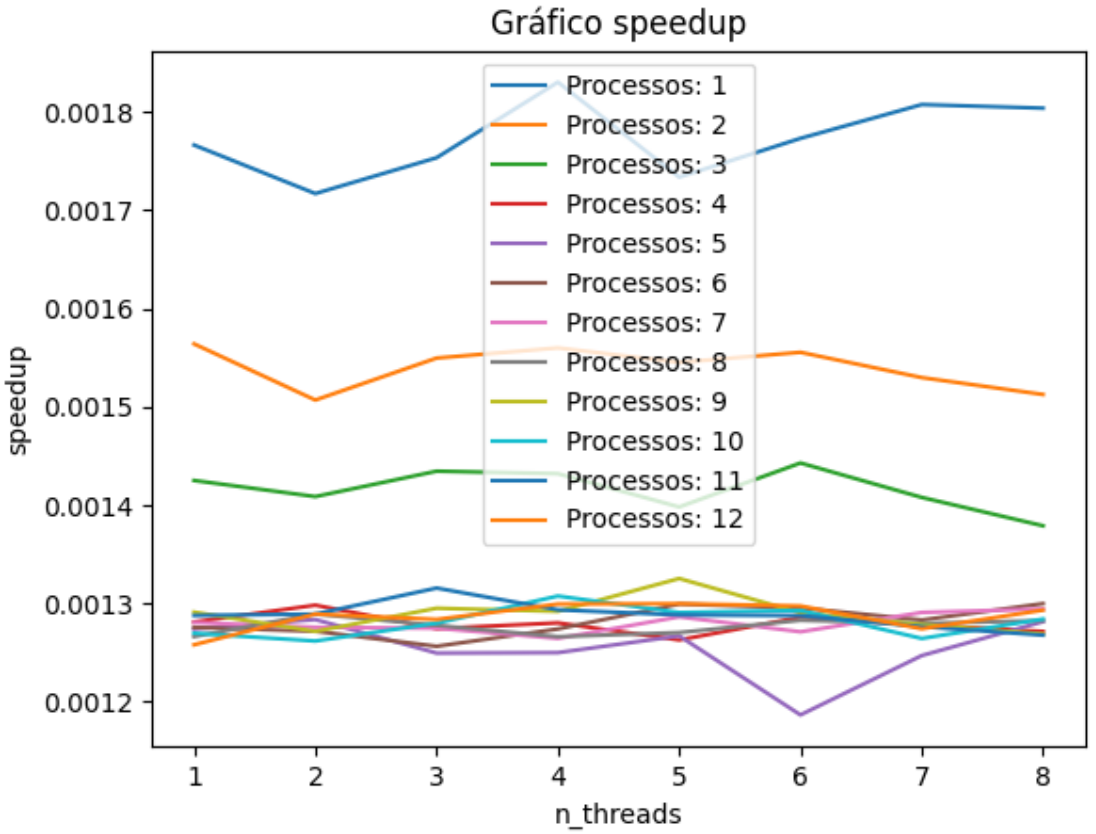

Gráfico Speedup utilizando input-sample, combinando 12 processos e 8 threads.

Podemos perceber que mesmo com um aumento no número de processos, e também no número de threads, a solução ideal continua sendo a sequencial, **mas por quê?**

O gráfico foi gerado se baseando no input fornecido, onde o número de sudokus é igual a 4. Para pouco processamento (poucos sudokus), na maioria das vezes, não se torna necessário a paralelização da execução. Portanto, somente o tempo de criação e destruição dos processos e suas respectivas threads, já torna o tempo do algoritmo maior. Podemos ver isso no gráfico, onde quanto mais processos, maior era o número de threads gerais, e assim, o tempo de criação e destruição influenciam no speedup, como comentado acima.

Porém, criamos um input com uma quantia muito maior de dados: 450.000 sudokus! E dessa maneira, iremos explorar um input com tamanho considerável, para assim, tomarmos outras conclusões acima da utilização ou não de threads e processos.

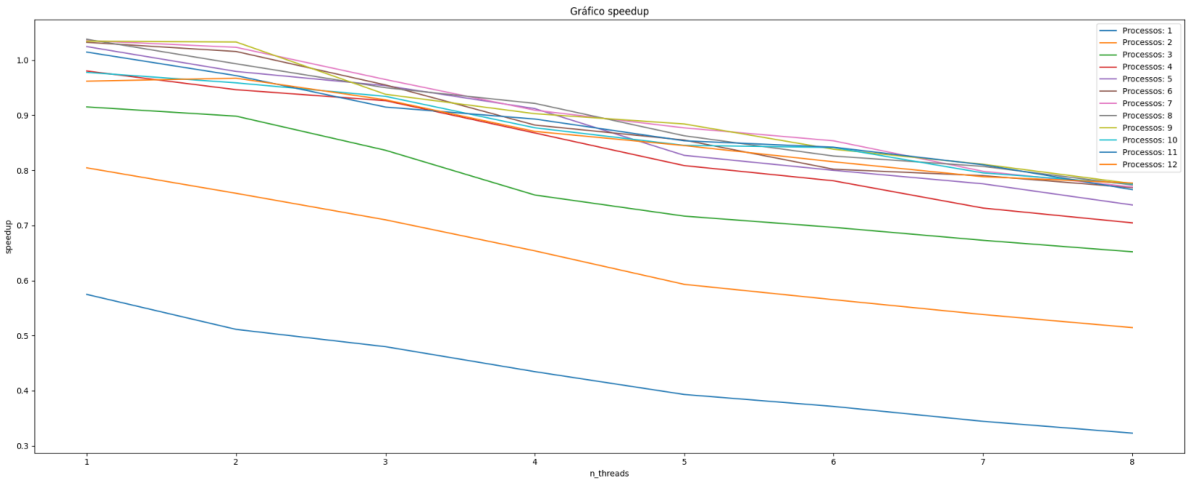

Gráfico Speedup utilizando input-big, combinando 12 processos e 8 threads.

Com uma quantidade maior de sudokus, podemos ver que acima de 3 processos, praticamente todas as execuções tendem a superar o tempo da sequencial, porém com poucas threads. Assim que o número de threads aumenta, todas as linhas tendem a descer, mas por quê?

As threads no programa foram responsáveis por separar o trabalho recebido pelos processos. A cada sudoku, elas deveriam dividir a quantidade de blocos entre elas, verificar se estavam corretos ou errados, e retornar ou não seus erros . Porém, como as threads em python trabalham de forma concorrente, e não paralela, não se torna viável utilizá-las no cálculo de validação dos sudokus. Além do mais, o gerenciamento de dados entre elas, combinado com seus tempos de criação e destruição, pioram o desempenho do programa.

Conclusão

Dessa maneira, podemos observar o comportamento de um programa utilizando métodos de concorrência para verificar a solução de um sudoku. A principal observação consiste em ponderar a necessidade de utilizar a concorrência em relação à quantidade de dados de entrada. Além disso, a quantidade de processos utilizados deve ser de forma balanceada, pois não se torna tão custoso utilizar muitos processos, quanto usar poucos. Importante ressaltar que a taxa ótima de processos varia de computador para computador.