Universidade Federal de Santa Catarina
Relatório do Projeto Prático 2

Participantes:
Thiago Comelli(19100546)
André Luiz Souza Santos(19150871)

Dezembro de 2020

Lógica de Divisão de Tarefas entre processos trabalhadores e threads

A lógica de divisão de trabalhos ficou simples. Primeiramente é realizada a leitura de todos os sudokus dentro do arquivo '.txt' solicitado, e armazenado dentro de uma array. Esta leitura inicial é realizada de forma sequencial antes de todo o processo de verificação começar, quando toda a leitura termina, o programa segue para o início das verificações.

Posteriormente cada processo criado ganha um sudoku para fazer a verificação, dentro de cada processo é criado as threads desejadas onde recebem as tarefas que devem realizar com o ThreadPoolExecutor.

Limites do número e processos trabalhadores e threads de correção

Para os processos não colocamos limites, porque vendo alguns artigos relacionados ao multiprocessing, cada núcleo de processamento da sua cpu recebe um processo para ser executado, conseguindo assim desviar do GIL que não permite o paralelismo. Sendo assim, se colocássemos limites de processos baseado na quantidade de núcleos que a sua máquina possuísse, talvez o usuário pudesse não encontrar o desempenho máximo para a situação (quer dizer, a quantidade de processos rodando em paralelo seria igual a quantidade de núcleos da sua máquina. Mas de forma concorrente podendo ser uma valor maior).

Realizando alguns testes em um máquina Quad-core, encontramos o melhor desempenho quando executado com um valor entre 5 e 7 processos. Por final acabamos não colocando limites para a quantidade de processos.

Para as threads de correção, colocamos o limite de 27 threads, porque cada thread realiza uma operação de verificação por inteira, ou seja, para cada sudoku é necessário realizar 27 operações de verificações (9 linhas, 9 colunas e 9 áreas).

Com isso não faria sentido gastar tempo de processamento criando mais de 27 threads sendo que o excedentes delas não seriam nem executadas.

Métodos de Sincronização Utilizados

Para os processos foram utilizados apenas um método de sincronização, possui um Lock que serve para dar print no final da sua execução. Sem esse Lock, era possível observar múltiplos processos dando print e as saídas ficando juntas. Com o Lock, a saída sai como o esperado.

Para as threads possui um lock que serve para trancar as seções críticas quando a função vai modificar algum valor da classe Task (essa classe

armazena todas as informações necessárias para cada processo funcionar e fazer as verificações). Fazendo muitas tentativas para quebrar essa parte do código, vimos que esse Lock é meio 'inútil' nessa situação, pois, como as threads não executam de forma paralela e nem realizam tarefas muito pesadas que forcem elas a ceder seu espaço, o Lock se torna desnecessário, mas colocamos para garantir que o resultado fique certo.

Ganho de performance Mensurado

Houve um ganho de performance bem significativo quando executado de forma paralela. Executando a correção de 3584 sudokus, a versão sequencial demorou 5.01 milissegundos para efetuar todas as correções, enquanto a versão paralela com 6 processos e 1 thread demorou 1.54 milissegundos. Conseguindo nesse caso um speedup de 3.253.

Com as execuções de teste que fizemos, percebemos que conforme aumentamos a quantidade de threads, pior fica o tempo de execução, isso ocorre porque como as threads não realizam nenhuma tarefa pesada que faça com que elas entre em um modo que permita ceder espaço e tempo de execução para outras threads, isso faz com que elas passam mais tempo escalonando do que realizando a tarefa em si.

Uma coisa que é perceptível quando é executado o programa com mais de uma thread, é possível observar que uma mesma thread realiza muitos trabalhos enquanto outras quase não chegam a ser executadas, isso ocorre pelo motivo do qual descrevemos acima, optamos por não forçar o uso das threads porque isso consumiria absurdamente muito tempo de processamento e deixar para que o próprio escalonador decida, por isso usamos um ThreadPoolExecutor para tal tarefa. Mas ainda é possível observar que existe a concorrência de threads, no arquivo ‘tasks.py’ linha 23, possui um time.sleep comentado, se ativar ele é possível ver múltiplas threads realmente trabalhando, contudo, o tempo de execução aumenta (o time.sleep força com que a thread de espaço e tempo de execução para outras threads).

Os testes foram realizados em uma máquina com um i5-3330 3.5ghz Quad-core, 12gb ram ddr3 1660mhz, GTX 750ti 2gb.