Trabalho 2 – Verificador de Sudoku Concorrente em Python

Grupo 14 - INE5410 – Programação Concorrente – UFSC
Profs. Márcio Castro e Giovani Gracioli
Alunos: Déborah Raquel B. Ferreira, Enzo Nicolás S. Bieger e Kuassi Dodji F. Kumako

Relatório do código

Validação de entrada:

Verificamos se a quantidade de elementos na linha de comando é o esperado, sendo eles nome do arquivo a ser rodado, nome do arquivo a ser avaliado, quantidade de processos e quantidade de threads para cada processo.

Quebra-cabeça:

A quantidade é obtida a partir do arquivo a ser validado. Temos a função que lê o arquivo txt e retorna o conteúdo em formato de dicionário com cada quebra-cabeças e um identificador único, de 0 até N-1 quebra-cabeças (usamos indexação começando no 0 para a lógica).

Processos:

Seu limite é avaliado a partir da função 'cpu_count()' que verifica quantos núcleos tem o computador que está rodando o código. Números abaixo de 1 não são considerados, em qualquer caso fora do limite é printado um erro.

Os processos são responsáveis por dividir os quebra-cabeças entre si, tendo 3 possíveis caminhos: quando o número de quebra-cabeças é maior que o de processos, quando o número de processos é maior que o de quebra-cabeças e a divisão de quebra-cabeças por processos tem resto igual a 0 ou não.

Sempre que quebra-cabeças%processos é maior do que zero, significa que sobraram quebra-cabeças que não foram distribuídos entre os processos, então pegamos todos os itens desses quebra-cabeças que restaram, e os dividimos igualmente entre todos os processos. Desta maneira, um processo pode pegar partes de quebra-cabeças para avaliar, fazendo com que a distribuição do trabalho entre cada processo, seja a mais igual possível.

Uma vez que o processo determinou quais itens ele vai avaliar, ele divide esses itens igualmente entre as threads.

Threads:

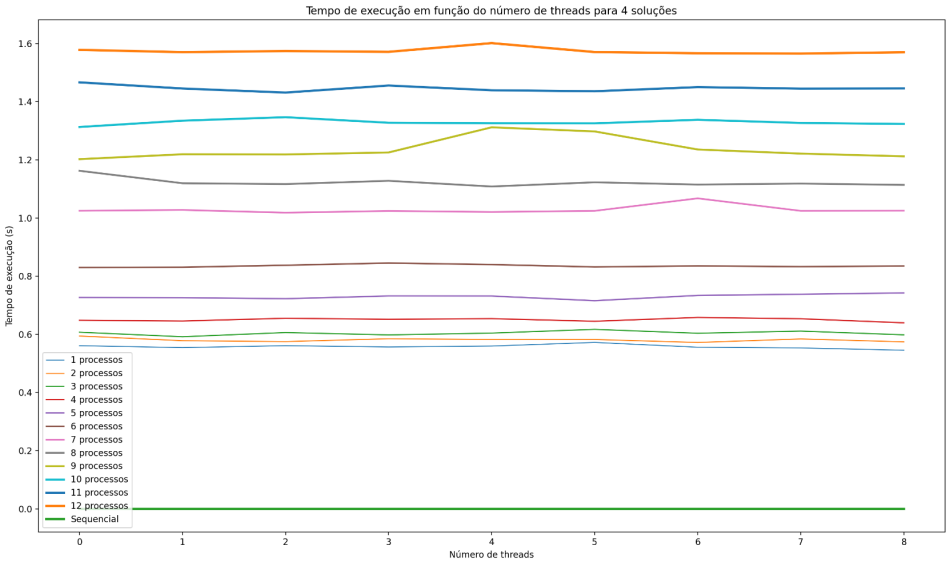
Seu limite depende do número de soluções dentro do arquivo. Em cada quebra-cabeça temos 27 itens, sendo 9 linhas, 9 colunas e 9 áreas. Multiplicamos o número de quebra-cabeças com o número de itens (27) e dividimos pelo número de processos escolhido na linha de comando, arredondamos para cima a resposta, para dessa maneira minimizar o número de threads ociosas em cada processo, nesse caso ficando no máximo com 1 thread ociosa em cada processo. Números abaixo de 1 não são considerados, em qualquer caso fora do limite é printado um erro.

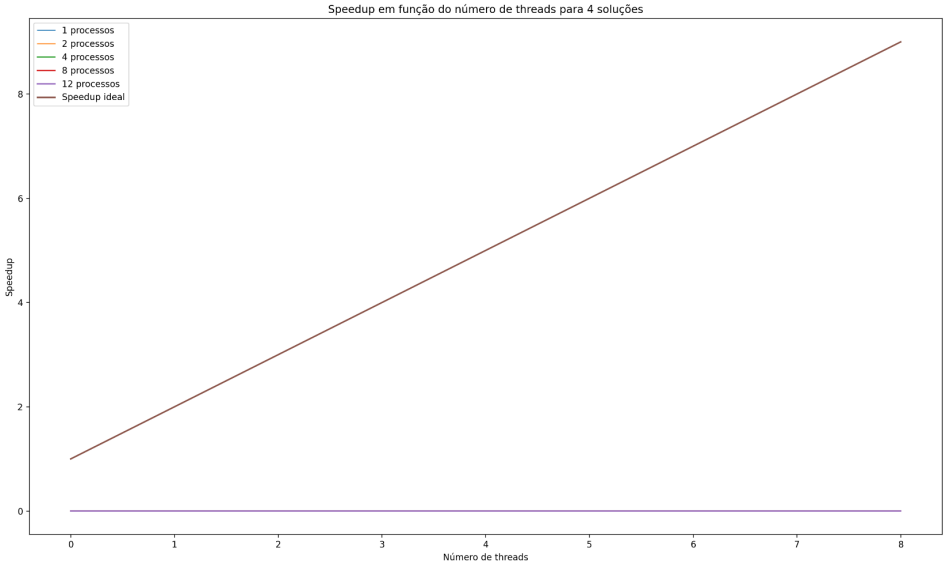
As threads são responsáveis por comparar se as linhas, colunas e regiões, estão corretas, a partir de um vetor de 1 a 9, comparando elemento por elemento.

Relatório de tempo de execução e speedup

Nos gráficos feitos analisando o tempo, percebemos que o modelo sequencial é mais rápido que o modelo paralelo independentemente do número de processos e threads escolhidos. Isso pode acontecer porque a validação de um item tem uma computação muito simples, então o programa demora mais tempo processando os itens e distribuindo eles para cada thread do que de fato calculando se um item é válido ou não. Então a criação de processos e threads não compensa. E por o tempo sequencial ser muito próximo de zero, os speedups também o são.

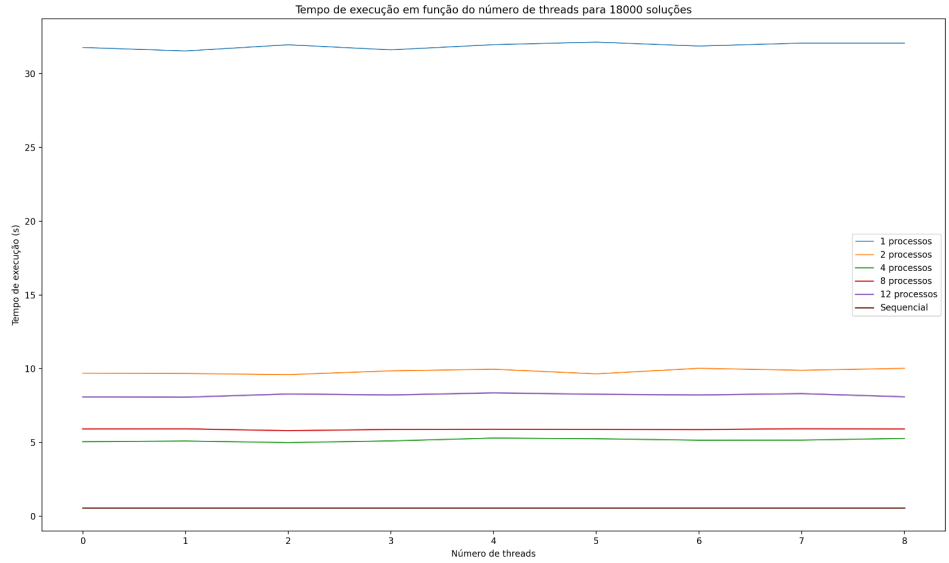
Nesses gráficos abaixo podemos ver os tempos de execução e os speedups para a avaliação de 4 e 18000 quebra-cabeças.

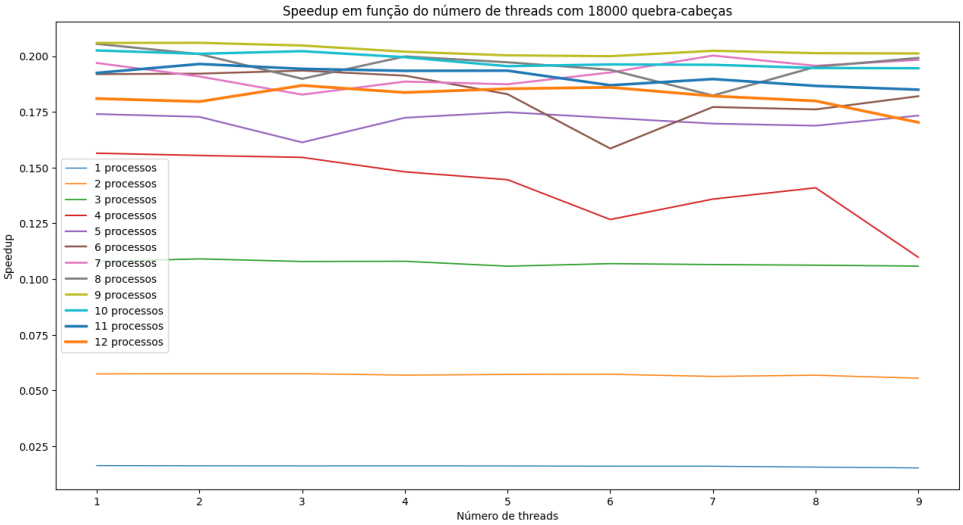




Nesses próximos gráficos, especialmente no de tempo de execução, por estarmos avaliando 18000 quebra-cabeças, podemos de fato ver que o aumento no número de processos diminui o tempo de execução, porém esse gráfico destaca ainda mais o problema da solução paralela, que é a divisão dos itens entre os processos e threads, isso se torna visível ao compararmos o tempo de execução com 1 processo e 1 thread (que praticamente é sequencial, porém teve que passar pelas divisões de itens) com o tempo sequencial que não teve que passar por essa divisão.

.





Como um adicional, achamos interessante simular que o programa teria muitas operações de entrada e saída, ou algo que gerasse uma certa ociosidade no programa. Fizemos isso colocando um sleep(1) na parte do código que avalia se um item está correto ou não. Nos gráficos abaixo, podemos visualizar como a concorrência dada pelas threads ajuda a reduzir a ociosidade e ganhar performance no programa, além é claro, de melhor visualizar o ganho de performance pelo paralelismo dos processos.