UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ALGORITMOS E ESTRUTURAS DE DADOS III

JOSÉ CARLOS DE OLIVEIRA JÚNIOR

DOCUMENTAÇÃO
TRABALHO PRÁTICO 3: PLANO DE DOMINAÇÃO MUNDIAL GLOBAL DO PROFESSOR WM. JR

Belo Horizonte – MG
31 de janeiro de 2017

1. INTRODUÇÃO

O problema apresentado neste trabalho consiste em obter a maior soma de inteiros em uma matriz com certas restrições a cada escolha. Escolhendo uma posição para incluir seu valor a soma, a posição a direita, a posição a esquerda, a coluna acima e a coluna abaixo são zeradas. Portanto, deve-se fazer as melhores escolhas entre as posições da matriz.

É possível sempre obter a soma das melhores escolhas utilizando Programação Dinâmica. Para cada linha, é possível obter sub-vetores, obter a melhor soma e então ir para um próximo super-vetor, para obter a soma novamente, fazendo com que nunca tenha-se que calcular um valor novamente.

Para isto, tomando cada linha da matriz, é verificado cada posição da segunte maneira:

- Se possui uma posição no total, a melhor soma é esta posição.

- Se possui duas posições no total, a melhor soma é a posição de maior valor.
- Se possui mais de duas posições no total, para cada posição é gravado como a melhor soma o maior valor entre a última soma obtida e a antipenúltima soma obtida mais o valor da soma atual.

Com isto, a próxima melhor soma do super-vetor é obtida usando a melhor soma dos sub-vetores, com é feita na Programação Dinâmica.

Depois que isto é feito, temos as melhores somas para cada linha da matriz. Ao escolher uma das somas, a soma da linha acima e abaixo são invalidadas conforme regras de escolha. Para resolver isto, os mesmos passos para encontrar a m*elhor soma das melhores somas*, uma vez que as regras de escolhas são as mesmas.

2. IMPLEMENTAÇÃO

2.1. Programa principal

O programa recebe o número de linhas e colunas, bem como os valores de cada posição diretamente pelo `stdin`. Além disto, é passado um inteiro como argumento da função principal que definirá o número de núcleos de processamento utilizados para a execução. São feitas verificações para que o programa somente execute quando é passado um inteiro como argumento da função principal e é feita a entrada correta da matriz.

Feita a entrada correta, a função `melhorSomaTotal` é chamada passada com a matriz, chamada aqui de mapa, e o número de núcleos utilizados. O resultado da função exibido, espaços dinâmicamente alocados são liberados e o programa é finalizado.

2.2. Programação Dinâmica para resolução do problema

Como foi mostrado, o problema pode ser resolvido por Programação Dinâmica e as duas funções responsáveis `melhorSomaTotal` e `melhorSomaLinha`. A ideia geral é que cada linha é processada, gerando um valor que é guardado em um vetor, sendo este vetor processado da mesma maneira gerando o resultado final enviado para a função principal. A seguir uma explicação mais detalhada das funções:

- **`melhorSomaLinha`:** esta função recebe como argumentos o vetor para ser processado e o tamanho do vetor. Na explicação da introdução é dito que as

duas primeiras posições são verificadas separadamente e é feito o aproveitamento das melhores somas anteriores a partir da terceira posição. Na implementação há duas variáveis a e b que representam as duas últimas melhores somas. Inicializando-as com `0`, os passos são os mesmos para todos as posições, funcionando normalmente quando o vetor possui apenas uma posição. Ao fim, a variável a possuirá a melhor soma e então, é retornada.

- **`melhorSomaTotal`:** está função recebe o mapa e o número de núcleos de processamento. Este último argumento será explicado a frente. Esta função é responsável por gerar um vetor final que será as melhores somas de cada linha. Para cada posição, será chamada a função `melhorSomaLinha` para armazenar os resultados no vetor. Por fim, a função é novamente chamada para processar o vetor das melhores somas, gerando a *melhor das melhores somas*, conforme explicado anteriormente. Este valor é o resultado do algoritmo.

2.3. Implementação de Paralelismo de Dados

Na implementação, a função `melhorSomaLinha` é executada uma vez para cada linha do mapa. A execução é normalmente feita sequencialmente. Com Programação Paralela, é possível que a execução se torne mais rápida, permitindo que cada núcleo de processamento receba um conjunto de linhas para ser processada. Não haverá conflitos porque os dados acessados ou escritos pelos núcleos nunca serão os mesmos, ainda que compartilhem o mesmo vetor ao armazenar as melhores somas.

Para implementar o paralelismo, foi feita modificações na função melhorSomaTotal e criada a função **`paralelismoMelhorSoma`**. Uma das formas de fazer a implementação é fazendo com que cada núcleo execute um bloco sequenciais e determinados de linhas. Esta implementação é diferente, uma vez que os núcleos não processam necessariamente blocos sequencias. Nesta implementação, os núcleos processarão a próxima linha ainda não processada assim que o seu processamento atual é finalizado.

A função `paralelismoMelhorSoma` é a função que será chamada para cada núcleo como também é a função que irá calcular a próxima linha disponível. Receberá como argumentos o mapa e o vetor de `melhoresSomas`. A função `melhorSomaTotal`

passa a chamar a função `paralelismoMelhorSoma` para cada núcleo passando os mesmos argumentos. Quando chamada, a função entra em uma estrutura de repetição que passará por todas as linhas e chamará a função `melhorSomaLinha` cada vez que encontrar uma linha não processada até que chegue ao fim da matriz.

O controle da linha atual é feito por um novo atributo adicionado a estrutura mapa, que contem toda a matriz. O atributo `linhaAtual` é inicializado com 0 quando o mapa é construído e incrementado cada vez que `paralelismoMelhorSoma` chama `melhorSomaLinha`, não importando qual núcleo fez isto.

3. ANÁLISE DE COMPLEXIDADE

3.1. Tempo

`melhorSomaLinha` irá passar pelo vetor apenas uma vez para armazenas as melhores somas nos sub-vetores. Possui complexidade $O(L)$, sendo L o número de linhas. melhorSomaTotal irá chamar a função melhorSomaLinha C vezes, sendo C o número de colunas. Ainda, irá chamar mais uma vez para processar o vetor de melhores somas. Possui complexidade $O(CL+L)=O(CL)$.

A complexidade do algoritmo é $O(CL)$, porém C e L juntos representam a quantidade de posições na matriz. No algoritmo, cada posição é avaliado individualmente para gerar a próxima melhor soma. Conforme a quantidade de posições na matriz cresce, o tempo cresce linearmente. Portanto é $O(n)$, crescimento linear de tempo para o número de posições na matriz.

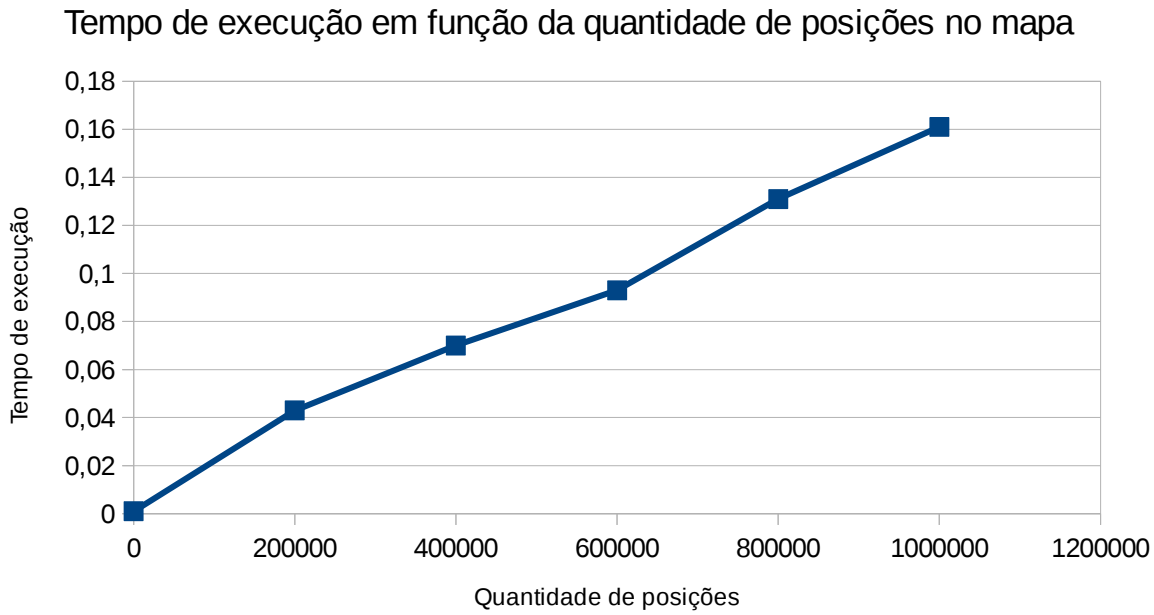
3.2. Espaço

Na função principal, uma estrutura é alocada com uma matriz $C \times L$ e na função melhorSomaTotal é alocada uma matriz de tamanho L. Possui então é $O(CL+L)=O(CL)$. Novamente, $C \times L$ é a quantidade de posições na matriz, portanto é $O(n)$, crescimento linear de espaço para o númeor de posições na matriz.

4. ANÁLISE EXPERIMENTAL

4.1. Tempo de execução em função da quantidade de posições no mapa

Conforme explicado anteriormente, a complexidade de tempo é linear em relação a quantidade de posições no mapa. Para mostrar isto, foram realizados testes para medir o tempo de execução do algoritmo para diferentes quantidades de posições no mapa.

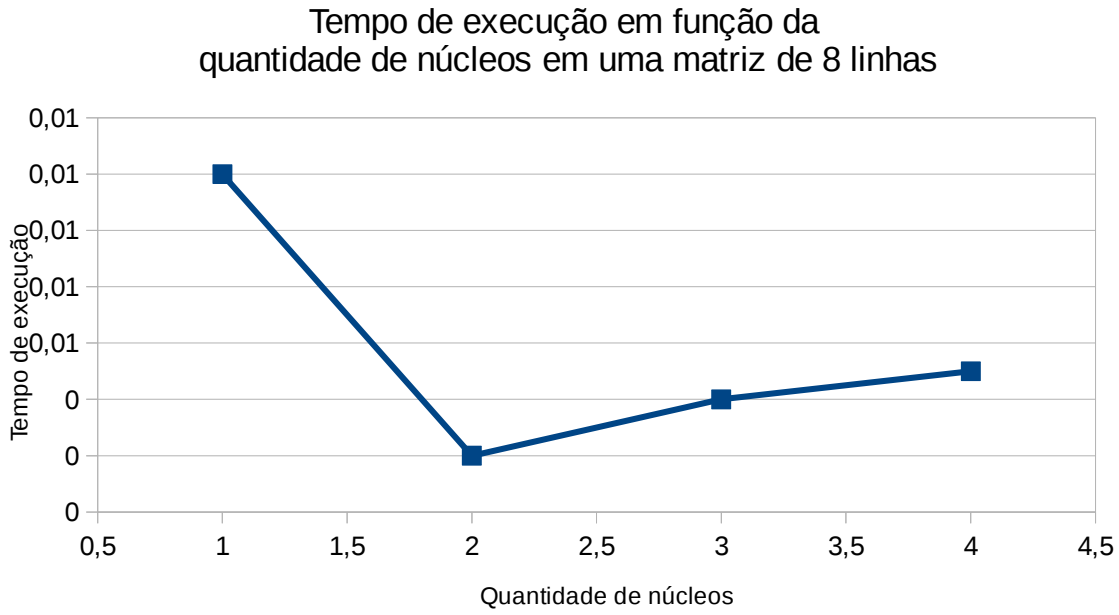


Quantidade de posições	Tempo de execução
1	0,001
200000	0,043
400000	0,07
600000	0,093
800000	0,131
1000000	0,161

Conforme dados mostrados, o tempo de execução apresenta um crescimento linear quando a quantidade de posições aumenta, comprovando a análise teórica.

4.2. Tempo de execução em relação ao número de núcleos

Com a implementação do paralelismo no algoritmo, para medir o tempo de execução conforme o número de núcleos aumenta, foi feito um teste com uma matriz e variando-se o número de núcleos.



Quantidade de núcleos	Tempo de execução
1	0,012
2	0,002
3	0,004
4	0,005

Neste teste, o tempo caiu drasticamente ao usar 2 núcleos, mas aumentou levemente ao usar 3 e 4 núcleos. Isto se deu, provalemente, devido a instrições não paralelizáveis. O tempo destas instruções acaba sendo maior que o tempo levado nas instruções paralelizáveis e o tempo passa a aumentar.

5. CONSIDERAÇÕES FINAIS

Neste último trabalho prático, a maior dificuldade foi entender como funcionam as funções da biblioteca pthread e como distribuir os dados a serem processados entre os núcleos. Com a possiblidade de pthreads melhorar o tempo de execução do código, deve-se percerber se o modo como foi implementado pode levar a conflitos de dados, como também se a paralelização realmente é benéfica levando-se em conta o modo de como foi implementada.