Documentação - Trabalho Prático 3

Esthefanie Pessoa Lanza - esthefanielanza@gmail.com

Janeiro, 2017

1 Introdução

O objetivo principal deste trabalho é aplicar os conceitos de programação dinâmica visto na matéria de Algoritmos e Estruturas de Dados III e melhorar a performance do programa utilizando paralelismo através da biblioteca pthread da linguagem C.

O problema visa ajudar o professor WM com a dominação mundial controlando o maior número de pessoas possíveis com a sua nova maquina de manipulação mental. Entretanto, tal equipamento existe grandes sacrifícios pois, ao dominar uma cidade, explodimos todas as cidades próximas. O objetivo é escolher as cidades cuidadosamente para conseguir o maior de número de seguidores.

A entrada consiste em uma matriz N por M onde cada casa representará uma cidade e seu respectivo número de habitantes. Ao escolher determinada cidade para ser dominada, as cidades da coluna ao lado são destruídas assim como todas presentes nas linhas acima e abaixo.

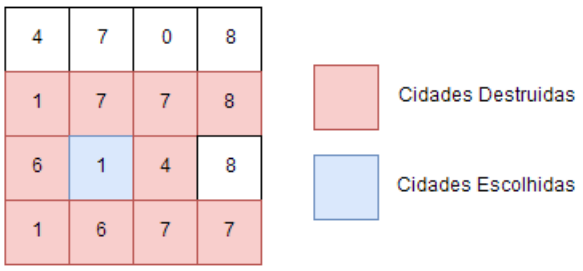

Figure 1: Exemplo de efeitos colaterais ao dominar uma cidade

A saída consiste em uma única linha que deverá informar o maior número possível de sobreviventes para aquele mapa.

2 Solução do Problema

2.1 Modelagem

Para a solução do problema deveríamos usar programação dinâmica, ou seja, era preciso encontrar uma subestrutura ótima para evitar recálculos. Dessa forma, foi necessário analisar cada linha da matriz individualmente e assim, considerando as restrições do problema, foi possível encontrar o máximo de cada linha e, através de um processo similar o valor máximo de sobreviventes daquele mapa.

Considerando cada linha individualmente, ao escolhermos uma cidade devemos ponderar quais decisões farão com que a linha atinga seu máximo. Assim, devemos considerar duas possibilidades:

- Escolher uma cidade somada ao seu complemento, localizado a duas casas atrás;
- Escolher a cidade anterior;

O máximo entre essas duas condições deverá substituir o elemento no vetor. Assim, o número obtido na última casa será o máximo daquela linha. Basicamente, devemos percorrer todos os elementos da linha e realizar a seguinte operação matemática:

$$linha[i] = max(linha[i-1], linha[i] + linha[i-2]) \tag{1}$$

Para facilitar o cálculo, como devemos lidar com duas posições anteriores à atual, foram adicionadas duas colunas zeradas a matriz original. Um exemplo das operações realizadas pode ser visto na figura abaixo.

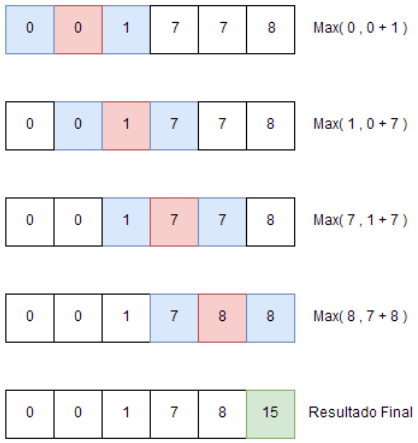

Figure 2: Exemplo calculo do valor máximo de uma linha

Após calcular o máximo de cada linha individualmente, devemos colocar todos esses valores em um vetor adjacente a matriz e realizar a mesma operação para calcular o valor máximo de sobreviventes possíveis do mapa.

Em relação a programação paralela é feito um paralelismo de dados, onde dividimos o número de linhas do mapa pelo o número de threads disponíveis processando os cálculos em grupos simultaneamente, já que estes são independentes.

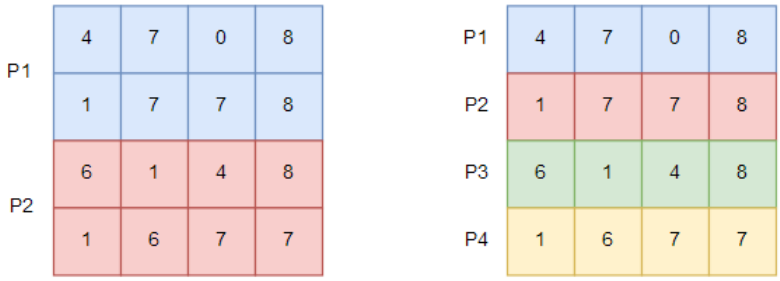

Figure 3: Processamento dos dados para 2 e 4 threads

2.2 Implementação

Para inicializar o programa é necessário informar, através do $arv[1]$ o número de threads que irão ser utilizados na execução. Dessa forma, o primeiro passo é alocar parte da memória disponível, utilizando o comando malloc, para uma variável do tipo pthread$_t$.

Com os elementos para a paralelização já alocados, iniciamos a leitura do mapa. A primeira linha deve informar o tamanho, N e M, da matriz. Em seguida, preenchemos as duas primeiras colunas com zeros para facilitar os cálculos como explicado na sessão anterior. Por fim, através de um for duplo lemos todos os elementos da matriz utilizando o comando scanf.

A partir do momento que o mapa já foi inicializado já podemos inicializar a resolução do problema. Considerando o número de threads iremos dividir a matriz em grupos de linhas realizando o cálculo explicitado na equação 2.

$$Númerodelinhasporthread = \frac{NúmerototaldeLinhas}{Númerodethreads} \tag{2}$$

Como os parâmetros devem ser passado por uma única variável, já que usaremos a função pthread_create, foi criado uma rotina responsável por definir a linha de inicio e término que deve ser processada por cada thread.Em seguida, executamos a função principal que deve calcular o valor máximo de cada linha. Através de um for duplo executamos a equação 1 descrita na sessão anterior, o pseudocódigo pode ser encontrado abaixo.

Algorithm 1: CalculaSoluçãoFila(void *arg)

```
Parametro p = *Parametro *)arg;
for i = p.start < p.end do
    for j = 2 < p.nColunas do
        p.mapa[i][j] = Max(p.mapa[i][j-1], p.mapa[i][j]+p.mapa[i][j-2];
    end
end
```

O primeiro for é referente as linhas que devem ser processadas. Por outro lado, o segundo deve percorrer todas as colunas, ignorando as duas primeiras pois são compostas apenas de zeros como dito anteriormente. Enquanto todas as linhas não são processados o código ficará travado por um $pthread\_join$, que esperará o término de todas as threads.

Em seguida, é criado um vetor solução, correspondente a ultima coluna da matriz, que representará os máximos de cada linha. Dessa forma, é feito um calculo idêntico ao descrito na equação 1 para calcular a solução final que é impressa no stdout.

3 Análise de complexidade

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço.

3.1 Análise Teórica do Custo Assintótico de Tempo

Para a análise assintótica foram observadas as funções principais do código de maneira geral, sendo elas as seguintes funções: leituraMapa, calculaLinha, criaParametros, criaVetorSolução e calculaSolução.

- A função leituraMapa é responsável por alocar a matriz, criar as colunas de zeros e a realizar a leitura do mapa. Para adicionar os zeros a matriz é feito um loop de custo $O(2 \times M)$, sendo M o número de linhas. Para a leitura das populações de cada cidade é feito um for duplo de custo $O(M \times N)$, sendo N o número de colunas. A complexidade total da função é $O(M \times N)$.

- A complexidade da função calculaLinha depende do número de threads utilizado naquela execução. Como para fazer o calculo do máximo de cada linha devemos percorrer todos os seus elementos, o custo, para cada processador, será equivalente a $O(\frac{M}{nThreads} \times N)$. É importante ressaltar que se o número de linhas não for divisível pelo número de threads o ultimo processador ficará encarregado de mais linhas. Entretanto, o custo geral da função, considerando todos os processadores, é $O(M \times N)$.

- A função criaParametros deve criar um parâmetro para cada thread existente, logo seu custo é $O(Nthreads)$.

- As funções criaVetorSolução e calculaSolução dependem apenas do número de linhas da matriz, ou seja, seu custo é $O(M)$.
- A complexidade temporal do programa em geral tem custo $O(M \times N)$

3.2 Análise Teórica do Custo Assintótico de Espaço

Para a análise assintótica foram observadas as funções principais do código de maneira geral, sendo elas a função responsável pela leituraMapa, criaParametros e criaVetorSolução.

- A função leituraMapa deve alocar espaço para matriz mais duas colunas zeradas, ou seja, sua complexidade espacial é $O(M \times (N + 2))$.
- A função criaParametros deve criar um vetor do tamanho do número de threads, assim seu custo é $O(nThreads)$.
- A função criaVetorSolução também deve ter o espaço para 2 colunas com zero e deve abranger o número total de linhas. Dessa forma, sua complexidade é $O(M + 2)$.
- A complexidade espacial do programa em geral é equivalente a $O(M \times (N + 2))$

4 Avaliação Experimental

4.1 Metodologia

A implementação da solução proposta foi compilada utilizando gcc 4.8.4. Os experimentos foram executados em um sistema com 4 GB de memória e um processador Intel Core i3 2310K @ 2.1 GHz. Cada teste foi realizado em torno de 30 vezes e foi realizada uma média entre os valores obtidos.

4.2 Avaliação Experimental do Tempo de Execução

4.2.1 Análise do tempo de execução em relação ao número de cidades

Para a análise da variação do tempo em relação ao número de cidades foram criados mapas com cidades até 10 habitantes e foram usados todos os processadores disponíveis da máquina teste, ou seja, 4.

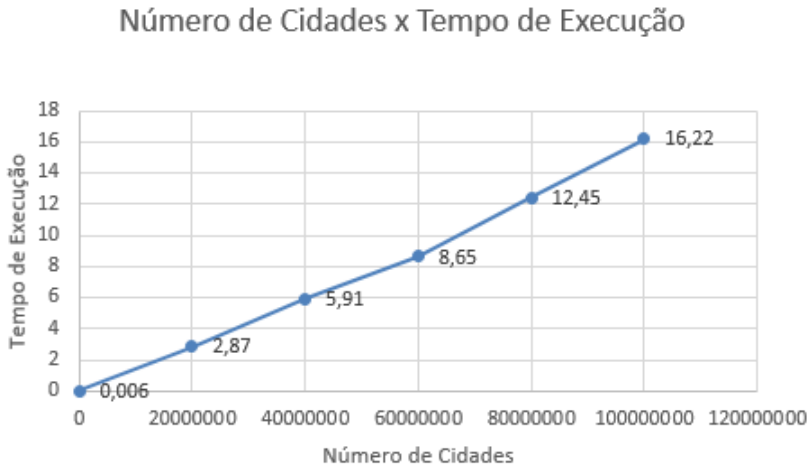

Figure 4: Análise da variação do tempo em relação ao número de cidades

Assim como o esperado, o gráfico apresentou comportamento linear. Para o número máximo de cidades explicitado pelo enunciado, 10^{12}, teríamos que o tempo de execução, através do cálculo da inclinação, seria de $1,9 \times 10^5$, em outras palavras, 2,2 dias.

4.2.2 Análise do tempo de execução em relação ao número de chaves

O teste foi realizado de maneira que veríamos somente o tempo do processamento da solução, ou seja, foi retirada a leitura da matriz e a resposta obtida foi adquirida através de números não inicializados. Isso foi feito para observar melhor as diferenças ao aumentar o número de threads disponíveis.

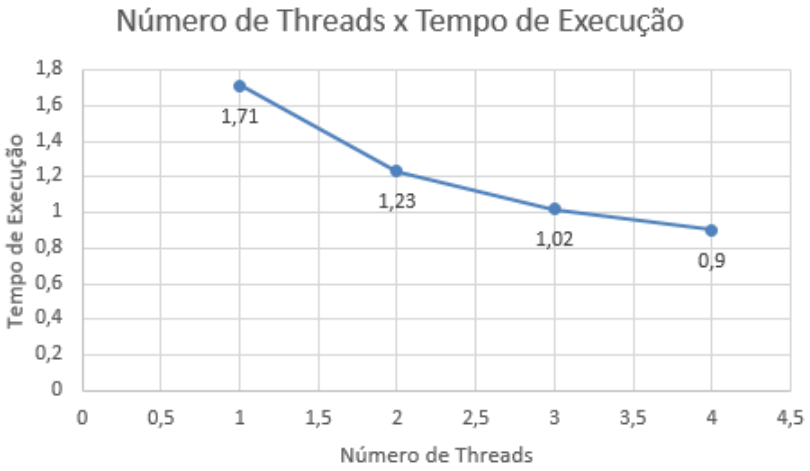

Figure 5: Análise da variação do tempo em relação ao número de threads

O número de threads disponíveis varia de acordo com a máquina de teste, como trata-se de uma máquina quadcore, aumentar para além desse limite não melhoraria os tempos de execução.

Assim como esperado, houve uma queda ao aumentar o número de threads. Entretanto, esta não foi tão linear como o esperado. Desta forma, podemos concluir que variáveis externas, como os processadores, podem influenciar nos tempos de execução.

4.3 Avaliação Experimental da Memória Utilizada

4.3.1 Análise da memória alocada em relação ao número de vértices

Neste teste aumentamos o número de cidades e deixamos o número de threads disponíveis constante em seu valor máximo.

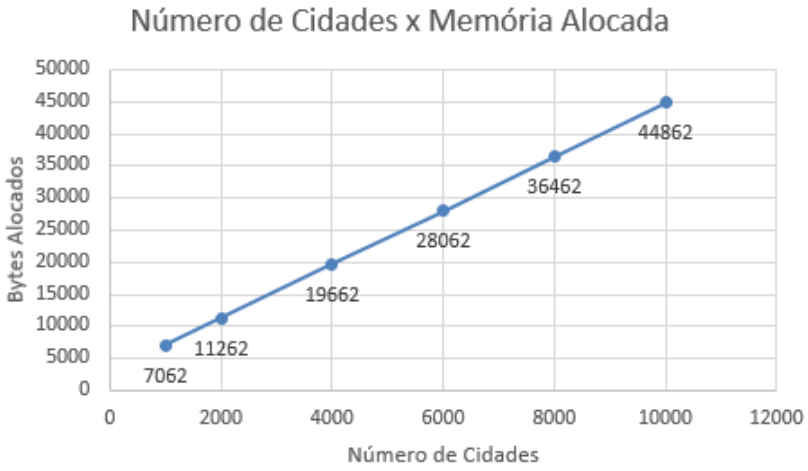

Figure 6: Análise da variação da memória alocada em relação ao número de cidades

Assim como o esperado, o gráfico cresceu linearmente ao aumentarmos o tamanho do mapa. Entretanto, para 10^{12} cidades, valor máximo estipulado para este trabalho, a memória alocada estaria na casa de 4,2 Terabytes.

5 Conclusão

Através das programação dinâmica foi possível identificar uma maneira de resolver um problema, que quando visto superficialmente nos leva a testar todas as combinações, de maneira simples e rápida. A implementação utilizada permitiu calcular o máximo de sobreviventes de mapas grandes em questão de segundos, sendo que a resolução por força bruta iria custar muito mais tempo.

O uso de programação paralela para calcular o máximo das linhas permitiu otimizar ainda mais o código e nos fez lidar com uma biblioteca até então inédita no curso, a pthread.h.

Em geral, a leitura da matriz com as populações de cada cidade mostrou-se muito mais lenta do que a própria execução das operações afetando muito o tempo de execução total do algoritmo. Entretanto, como a leitura dos valores não pode ser realizada em paralelo não foi possível otimizar esse trecho do código.