Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Algoritmos e Estruturas de Dados III

Trabalho Prático 3

Aluna Ana Luiza de Avelar Cabral

1. Introdução

Este trabalho prático visa trabalhar as dificuldades e conceitos de grandes volumes de dados e uso da memória secundária no processamento.

O problema dado consiste em encontrar a média aritmética de todos os elementos de uma matriz, a média aritmética dos elementos de cada linha e a mediana dos elementos de cada linha. Como são grandes elementos e o uso da memória é muito restrito o processamento precisa ser feito usando a memória secundária.

É amplamente sabido que o custo de tempo para acesso à memória secundária é altíssimo. Sendo assim, o uso de memória secundária que é imprescindível para o processar o problema deve ser feito de forma otimizada ou terá um custo altíssimo a solução.

Para o processamento, são lidos da linha de comando: o arquivo de entrada, qual será o arquivo para saída, e o limite máximo de memória que o programa pode consumir durante seu processamento. Tudo é passado somente na hora da execução, dinamicamente.

./tp3 arquivo_entrada.in arquivo_saida.out LIM_M

2. Modelagem do problema

De acordo com o enunciado do problema os valores da matriz são valores inteiros positivos até no máximo 4,294,967,296, que é o maior valor que um inteiro pode assumir.

Para calcular a média aritmética dos valores, seja por linha ou em toda a matriz, esses valores precisam ser somados e então a soma é dividida pela quantidade de elementos somados. Assim, optei, para economizar memória alocada, ler apenas um elemento por vez sem salvá-los em estruturas de dados do programa, para diminuir a quantidade de memória alocada.

Ler cada elemento sem salvar implica em então à medida que for lendo já ir calculando as saídas pedidas.

Cálculo da média aritmética da matriz

Para calcular a média aritmética entre todos os elementos da matriz sem salvá-los vou lendo cada valor e somando-o ao valor total. Isso é possível porque o valor máximo de cada valor é 4,294,967,296, aproximadamente $4{,}2 \times 10^9$. São no máximo $(3{,}0 \times 10^3) \times (3{,}0 \times 10^3) = 9{,}0 \times 10^6$ elementos na matriz (máximo de linhas x máximo de colunas). Assim, no pior caso, o valor máximo da soma acontece quando temos o máximo de linhas, máximo de

colunas e valor máximo em cada elemento: 3,0 x 10^3 x 3,0 x 10^3 x 4,2 x 10^9 = 37,8 x 10^{15} = 3,78 x 10^{16}. De acordo com [1] e de acordo com a biblioteca *limits.h* [2], o tipo de dados long long int tem valor máximo 9 x 10^{18}, e unsigned long long int tem valor máximo 1,8 x 10^{19}. Testei no meu computador este tipo de dados junto com o operador sizeof(), para imprimir na tela a quantidade de bytes do tipo e verificar. E de fato o tipo tem suporte.

Assim, como a maior soma possível é 3,78 x 10^{16} e long long int suporta até 9 x 10^{18}, decidi ler cada valor do arquivo como um long long int e logo após ler o somo na variável *mediaGeral*, que armazenará a soma de todos os valores lidos. Por fim a variável *mediaGeral* não pode ser long long int porque quando eu dividir a soma nela pela quantidade de inteiros lidos o valor não será necessariamente um inteiro, pode ser ponto flutuante. Assim eu escolhi o tipo float para mediaGeral, o que permite após ler todos os valores do programa e somá-los nela, dividi-la pela quantidade de valores lidos e já salvar nela própria o resultado. De acordo com a biblioteca float.h [3] e com testes locais para confirmar, o valor máximo que um float pode armazenar é 3,4 x 10^{38} (340282346638528859811704183484516925440.000000), o que permite armazenar a mediaGeral calculada.

Cálculo da média aritmética por linha

Para calcular a média aritmética por linha sem salvar os elementos, a cada linha faço a soma da linha e divido pelo número de elementos na linha. O máximo que a soma por linha pode alcançar é, no pior caso, 3,0 x 10^3 x 4,2 x 10^9 = 12,6 x 10^{12} = 1,26 x 10^{13}. Essa soma vai ser dividida pelo número de elementos lidos para encontrar a média e pode retornar um valor não inteiro. Como o valor máximo para a soma no pior caso é menor do que um float, foi decidido armazenar essa média em uma variável do tipo float, que cabe a soma e suporta valores não inteiros para após a divisão.

Assim, a cada linha a função *LeLinhaMatriz* lê uma nova linha da matriz e reseta a variável *mediaLinha* para calcular a média daquela linha sendo lida. Ao final da leitura de uma linha já imprime no arquivo de saída a média aritmética daquela linha. Já a média total só imprime após ler a última linha, que aí vai ter lido todos os elementos e vai poder então calcular.

Cálculo da mediana por linha

Para o cálculo da mediana dos valores de uma linha, é necessário:

- Ordenar os elementos lidos
- Encontrar a mediana entre os elementos lidos:
    - Se foi lido um número ímpar de elementos, a mediana é exatamente o elemento do meio.
    - Se foi lido um número par de elementos, divide o conjunto exatamente no meio pelo centro dele. A mediana é e média aritmética dos dois elementos centrais, os do meio do conjunto.

Assim, o cálculo da mediana é o grande desafio do trabalho. Isso porque dependendo do limite de memória passado para o programa essa ordenação não pode ser feita no programa porque não há memória disponível, a ordenação precisa ser feita usando a memória secundária.

Estudamos diversos algoritmos de ordenação para o uso eficiente da memória secundária. Entre eles eu escolhi usar o QuickSort Externo, pelo baixo consumo de memória primária e por permitir variar a área que usa da memória principal dinamicamente de forma direta. De acordo com o limite de memória informado na linha de comando a área está sendo colocada.

Assim, a função *LeLinhaMatriz* lê os valores de uma linha da matriz. A cada valor lido já incluo este na média aritmética por linha e na média aritmética total. Para a mediana preciso ler todos os valores e armazenar para depois processar com eles salvos separados e não na mesma variável como no caso das médias. Para conseguir fazer isso sem consumir memória primária, à medida que cada valor é lido ele é escrito no arquivo *auxlinha*, que ao final vai ter todos os valores de uma linha da matriz. Isto facilita a ordenação também por já colocar em arquivo separado os valores que serão ordenados.

Então após a leitura da linha e escrita dela em *auxlinha*, os valores em *auxlinha* são ordenados *in situ* usando o QuickSort Externo. Após a ordenação é então calculada a mediana:

- Se a linha tiver número ímpar de elementos, a mediana será o elemento no meio do arquivo exato;
- Se a linha tiver número par de elementos, a mediana será a média aritmética entre os dois elementos centrais do conjunto.

Impressão dos valores calculados no arquivo de Saída

Após ler cada linha e calcular a média por linha e a mediana por linha, esses valores já são impressos no arquivo de saída para não precisar armazená-los em uma estrutura de dados. Os valores lidos na linha também foram somados em uma única variável de média Geral. Após a impressão da última linha (impressão da média e mediana daquela linha), o programa calcula a média total com base na soma e na quantidade de elementos lidos e a imprime.

Dinâmica Geral do programa e modularização do código

Dinâmica Geral

No main é feita a leitura dos parâmetros de execução do programa (arquivo de entrada, arquivo para saída e valor limite de memória para a execução) passados na linha

de comando e a conferência se são válidos pela função InicializaPrograma. Se não forem válidos aborta a execução e retorna na saída padrão uma mensagem informando o erro.

Depois são lidas pela função *LeInfoMatriz* as informações do arquivo sobre a matriz: quantidade de linhas e colunas dela.

Logo após, cada linha da matriz é lida separadamente pela função *LeLinhaMatriz*. Essa função lê cada valor da linha e:

- o soma na variável de cálculo da média geral,
- o soma na variável de cálculo da média da linha,
- o escreve em arquivo auxiliar externo que vai ser ordenado para cálculo da mediana.

Então dentro da função *LeLinhaMatriz* a média da linha é calculada (isto é: divide a soma de todos os valores lidos pela quantidade de elementos lidos na linha).

Logo depois a mediana é calculada em duas etapas: primeiro o arquivo auxiliar com os valores é ordenado *in situ* pelo QuickSort Externo. Depois, com os valores ordenados, é feita a leitura do arquivo até o centro dos elementos para calcular, com base nesses elementos centrais, a mediana da linha.

Por último imprime os valores referentes a cada linha no arquivo de saída (média aritmética da linha e mediana dela). Caso esteja na última linha do arquivo, calcula a média aritmética geral com base na soma total (isto é: divide a soma de todos os elementos lidos pela quantidade de elementos lidos) e imprime também, logo após a impressão dos valores da última linha lida.

Modularização do Código

Para maior clareza e menor repetitividade, procurei modularizar em funções vários trechos do código.

Além disso, separei as funções por escopo em bibliotecas separadas.

A biblioteca **area.h** (arquivos area.h e area.c) tem definições sobre tipos de dados e funções somente sobre o TipoArea que é usado pelo QuickSort Externo. Poderiam estar na biblioteca do QuickSort Externo mas para maior clareza e separação de focos coloquei em bibliotecas separadas.

A biblioteca **qse.h** (arquivos qse.h e qse.c) tem as funções, bibliotecas, constantes e tipos de dados (estes últimos importados da biblioteca area.h) para fazer o **Q**uick**S**ort **E**xterno sobre os elementos da linha para o cálculo da mediana da linha.

A biblioteca **funcoes.h** (arquivos funcoes.h e funcoes.c) não é muito reutilizável como biblioteca por ser muito específica para o programa, mas não é esta a finalidade de separá-la em uma biblioteca: é uma biblioteca que existe para declarar e usar o que é preciso no programa principal (constantes, funções, bibliotecas, …). É a biblioteca que contém a modularização do programa principal, as funções do programa principal.

3. ANÁLISE DE COMPLEXIDADE

Complexidade de Tempo

A função InicializaPrograma chamada no início da execução tem custo constante O(1). Logo após esta, a função LeInfoMatriz também tem custo constante O(1). A última função do programa, EncerraPrograma, também tem custo O(1). Então analisamos a função LeLinhaMatriz.

A função LeLinhaMatriz começa lendo os elementos do arquivo. Ela realiza n leituras/escritas, onde n é o número de elementos da matriz. Assim, esta operação tem custo O(n). O Cálculo da média aritmética da linha tem custo constante O(1). A Ordenação pelo QuickSort Externo tem complexidade O(n*log n). Logo depois, o cálculo da mediana percorre o arquivo até a metade, tem custo (n/2) logo O(n). A impressão de resultados por linha tem custo constante. Assim, vemos que cada chamada da função LeLinhaMatriz tem custo O(n*log n).

A função LeLinhaMatriz é chamada M vezes, onde M é o número de linhas da matriz.

Assim, o programa tem complexidade de tempo de **O(M * n * log n)**, onde M é o número de linhas da matriz e n o número de elementos dela.

Complexidade Espacial

A função InicializaPrograma chamada no início da execução tem custo espacial que varia linearmente com LIM_M passado na execução do programa: O(LIM_M). A função LeInfoMatriz tem custo constante espacial e a função EncerraPrograma também.

A função LeLinhaMatriz é chamada M vezes, onde M é o número de linhas na Matriz. No início, todas as operações até a ordenação tem custo espacial constante. A ordenação usa a área linearmente proporcional a LIM_M alocada no início de memória primária apenas, o resto é no próprio arquivo logo o custo é constante. A busca pela mediana não armazena valores: tem custo espacial contante. E também tem custo espacial constante a impressão.

Assim, a complexidade espacial do programa é **O(LIM_M)**, onde LIM_M é o limite passado para a execução do programa na linha de comando.

4. ANÁLISE EXPERIMENTAL

Para analisar o comportamento do programa, realizei experimentos em um notebook com Sistema Operacional Linux - Ubuntu 16.04 com arquitetura de 64bits, 8GB de memória RAM, processador Intel Core i5 2.30GHz.

Para medir o consumo de memória usei a ferramenta *timeout* [4], que me informou o consumo máximo de memória em KB durante a execução. Em alguns casos usei também na linha de comando o comando

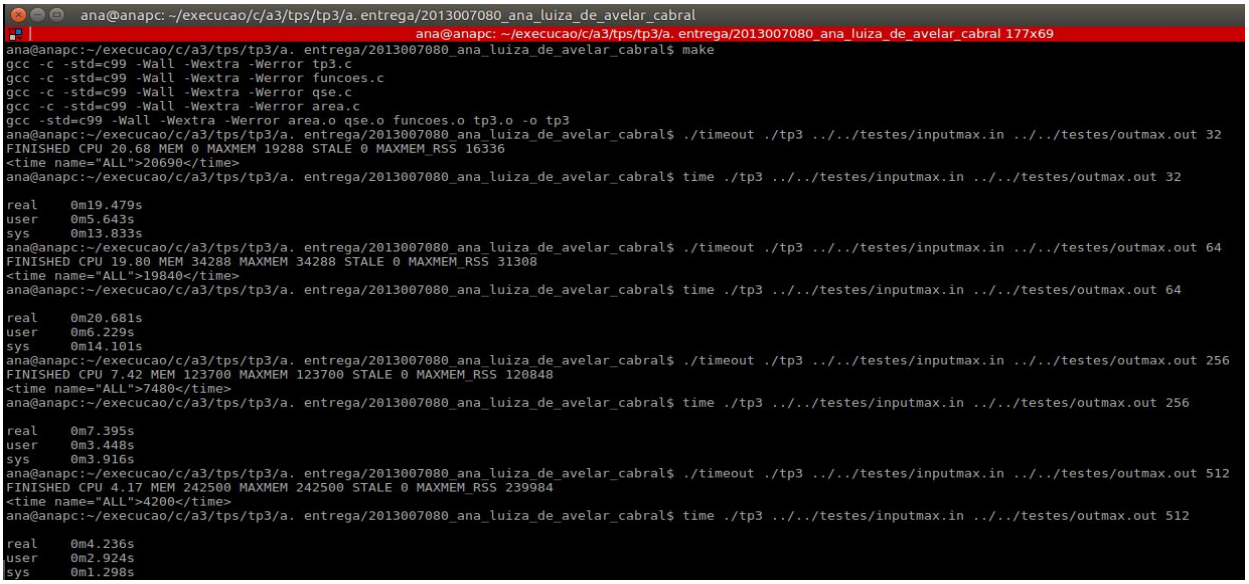
```
alias time="$(which time) -f '\t%E real,\t%U user,\t%S sys,\t%K amem,\t%M mmem'"
```

que faz com que *time ./tp3 [arquivo_entrada.in] [arquivo_saida.out] [LIM_M]* imprima o consumo máximo de memória em KB no trecho %M mmem. [5] [6].

O Caso toy fornecido começou confirmando que o cálculo da média aritmética por linha, mediana por linha e média aritmética total está certo. Porém não consegui medir consumo de memória com o caso toy usando LIM_M = 32 por exemplo por esse caso ser bem pequeno.

Para análise então gerei casos de teste maiores, e inclusive o caso de teste com o máximo possível: o máximo de linhas e colunas e o valor máximo em todos os elementos. Além disso gerei outros testes e também o máximo com apenas 1 linha na matriz para analisar um grande parcial.

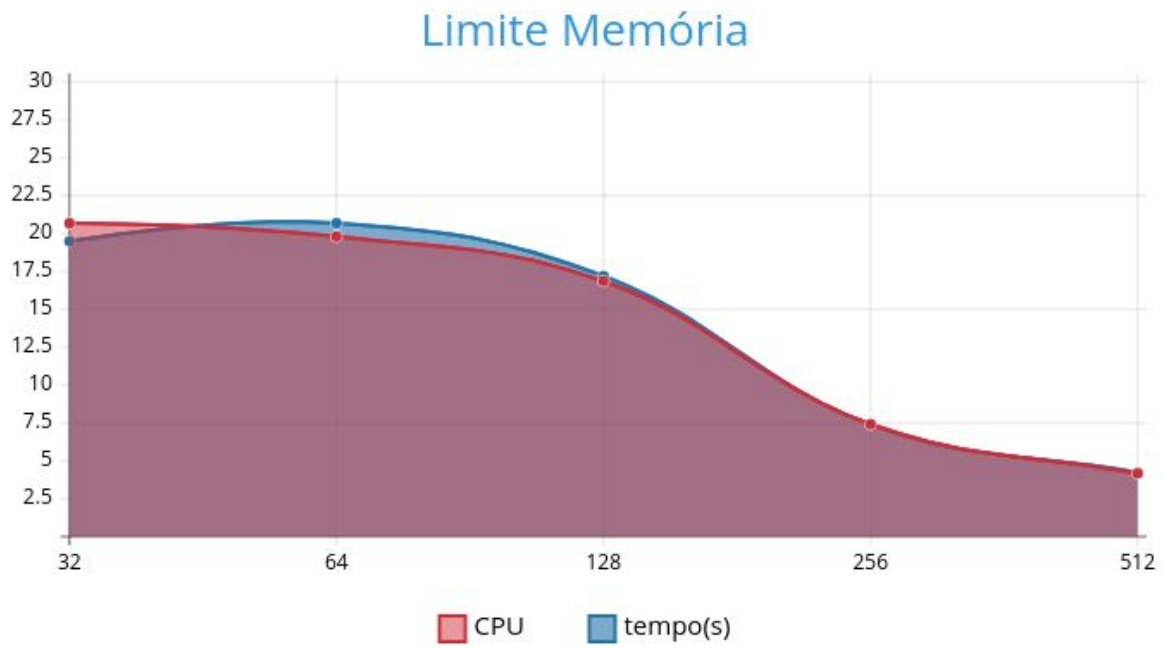
No print aqui abaixo tem a execução do caso de teste max.in, que tem o máximo de linhas, de colunas e em todos os valores da matriz. Por ser bem grande ele pode trazer análises mais claras na execução.



Na execução conseguimos ver claramente o tradeoff entre aumento do limite de memória e menor tempo de execução e menos processamento (refletido em CPU).

Isto é claro porque o limite de memória está ligado diretamente à área usada pelo QuickSort Externo. Quanto maior o limite maior coloco a área para o QuickSort, é uma alocação dinâmica e dependente do primeiro. Com uma área de memória primária maior para usar, tem menor processamento logo menor consumo de CPU e assim também termina mais rápido como vemos na função time.

Assim, podemos ver que existe um tradeoff entre limite de memória e maior trabalho (refletido no processamento pela CPU e no tempo).



Pelos testes vemos também que o programa em todos os testes executa dentro do limite de memória passado para ele.

5. Conclusão

Foi bastante interessante implementar o trabalho e ver diretamente na execução os resultados estudados e esperados. Foi muito interessante também ver a parte de medição do consumo de recursos: aprender as ferramentas que fazem esta análise. Em uma das ferramentas testadas para possível uso no trabalho [7], [8] foi possível ver até a quantidade de page faults na execução do programa:

```
ana@anapc:~/execucao/c/a3/tps/tp3/a. entrega/2013007080_ana_luiza_de_avelar_cabral$ sudo perf stat ./tp3 ../../testes/inputmax.in ../../testes/outmax.out 512.5
[sudo] password for ana:

 Performance counter stats for './tp3 ../../testes/inputmax.in ../../testes/outmax.out 512.5':

       4098,421344      task-clock (msec)         #    0,992 CPUs utilized
             3.018      context-switches          #    0,736 K/sec
                 1      cpu-migrations            #    0,000 K/sec
            60.064      page-faults               #    0,015 M/sec
    10.801.473.585      cycles                    #    2,636 GHz
    20.865.000.593      instructions              #    1,93  insn per cycle
     4.574.964.183      branches                  # 1116,275 M/sec
        16.089.832      branch-misses             #    0,35% of all branches

       4,130455762 seconds time elapsed

ana@anapc:~/execucao/c/a3/tps/tp3/a. entrega/2013007080_ana_luiza_de_avelar_cabral$
```

É possível concluir que foi interessante ver o comportamento do programa e aplicar as estratégias aprendidas na disciplina, e ver na prática funcionando.

Referências Citadas

[1] - https://en.wikipedia.org/wiki/C_data_types

[2] - http://pubs.opengroup.org/onlinepubs/009604499/basedefs/limits.h.html

[3] - http://pubs.opengroup.org/onlinepubs/007904875/basedefs/float.h.html

[4] - https://github.com/pshved/timeout

[5] - https://superuser.com/questions/480928/is-there-any-command-like-time-but-for-memory-usage, último post da discussão.

[6] - https://linux.die.net/man/1/time

[7] - https://superuser.com/questions/480928/is-there-any-command-like-time-but-for-memory-usage, post de pegunta.

[8] - https://linux.die.net/man/1/perf-stat