Trabalho Prático 0

Luiz Felipe de Sousa Faria – 2020027148

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lutilipe@ufmg.br

1. Introdução

O problema proposto foi implementar um Tipo Abstrato de Dado (TAD) de uma matriz. O objetivo da documentação é analisar o desempenho, a robustez, a localidade de referência e distância de pilha dessa estrutura diante de operações comuns que elas realizam. No trabalho, foram implementadas e analisadas as operações de soma, multiplicação e transposição da matriz.

2. Implementação

O programa principal se encontra no diretório ***TP***. Esse diretório é dividido em:

- ***include***: contém o arquivo *header* das funções da matriz e do *memlog*. Contém também as macros de *asserts*;
- ***src***: contém a implementação das funções do include;
- ***obj***: contém os .o gerados pelo programa (vazio inicialmente)
- ***bin***: contém o binário do programa (vazio inicialmente)
- ***Makefile***: gera os arquivos binário e *.o* do programa e executa os testes de desempenho e análise de localidade.

O programa foi escrito em *C* e compilado utilizando o *GCC*.

2.1. Estrutura de dados

A estrutura de dado principal do projeto é a matriz, uma forma de vetor em duas dimensões. Sua *struct* possui os atributos:

- ***m:*** correspondente a matriz em si. Esse atributo é um ponteiro para um ponteiro do tipo double;
- M e N: correspondem a quantidade de linhas e colunas, respectivamente;
- id: identificador da matriz para fins de *logs*.

O atributo *m* da estrutura é alocado dinamicamente. Isso permite a atribuição da dimensão da matriz, seja por um *input* externo ou alguma função interna, durante a execução do programa.

2.2. Funcionamento

O programa recebe como parâmetro o caminho para um arquivo contendo a primeira matriz (parâmetro *-1*) e outro caminho para outro arquivo contendo a segunda matriz (*-2*). Recebe também o caminho do arquivo de saída (*-o*). Esses arquivos possuem a primeira linha contendo as dimensões da matriz. O restante das linhas contém a matriz em si.

Pelos parâmetros, também são recebidos a operação a ser realizada (*-s* para soma; *-t* para transpor; e *-m* para multiplicação); o parâmetro *-l* para ativação do log de memória e o parâmetro *-p* indicando o caminho do output desse log.

As principais funções do programa são:

- ***initMatrixFromFile:*** essa função irá receber como parâmetro o nome do arquivo que contém a matriz que queremos usar; o ponteiro de uma *Matriz*; e o id da matriz. Ela irá ler a primeira linha do arquivo contendo a quantidade de linhas e colunas da matriz e criar uma matriz com essas dimensões. Após isso, inicializará a matriz com os valores do arquivo;
- ***writeMatrixToFile***: irá receber o nome do arquivo de saída e a *Matrix* contendo o resultado final de uma operação. A forma como a matriz é escrita no arquivo segue a mesma estrutura dos arquivos de entrada;
- ***sumMatrix***: realiza a soma de duas matrizes de dimensões iguais. Para cada posição das matrizes A e B, é obtido o valor resultante da soma de cada uma dessas posições. Esse valor é então armazenado em uma terceira matriz C. O ponteiro dessas matrizes são recebidos no parâmetro da função;
- ***multiplyMatrix***: realiza a multiplicação da matriz A pela matriz B, armazenando o resultado em C. Para cada linha da matriz A, obtém-se o valor da célula e é realizada a multiplicação escalar pela célula correspondente (mesmo *index*) na coluna B. O resultado é armazenado na posição (linha A)x(coluna B) na *Matrix* C.
- ***transposeMatrix***: realiza a transposição de uma matriz. Recebe como parâmetro a matriz a ser transposta e a matriz de destino na qual será armazenada o resultado da operação. A leitura é feita por linha na matriz A (origem) e armazenada por coluna na matriz B (destino). Essa função, ao contrário das outras duas relacionadas a operações (*sumMatrix* e *multiplyMatrix*), necessita apenas de um arquivo de entrada contendo uma matriz. O segundo arquivo (caso seja passado por parâmetro), é ignorado.

3. Análise de complexidade

Para a análise da complexidade de tempo das funções e do programa em si, consideremos uma matriz de dimensões M x N (número de linhas e número de colunas, respectivamente).

Para as funções de soma, transposição, leitura e escrita em arquivos, a análise é bem similar, por isso será feita em conjunto. Nessas funções, todos os elementos da matriz vão ser acessados uma vez. No caso da função soma, duas matrizes são acessadas na mesma operação para que seja possível obter o valor delas e inseri-las na matriz resultante. No entanto, isso não interfere na análise. Para tais funções, a complexidade dá-se por:

$$O(M) \, . \, O(N) \; = \; O(N . M)$$

Para a função de multiplicação, a complexidade é diferente. Nessa função, cada elemento de uma linha da matriz A será multiplicado pelo seu correspondente na coluna da matriz B. O resultado de cada multiplicação será somado e adicionado em uma posição da matriz C. O processo se repete até que a matriz C tenha a quantidade de linhas iguais às linhas da matriz A, e colunas iguais às linhas da matriz B. Sendo assim, a complexidade desse processo se dá por:

$$O(M) \, . \, O(N) \, . \, O(N) \; = \; O(N^2 . M)$$

Caso ambas as matrizes sejam quadradas, a complexidade será $O(N^3)$.

A complexidade geral do programa dependerá da operação escolhida.

4. Estratégias de robustez

As operações com matrizes, em sua maioria, exigem que certas condições sejam satisfeitas previamente. Caso contrário, essas operações não devem ocorrer. O programa apresenta estratégias para lidar com essas condições.

Na inicialização da matriz pelo arquivo, é validado o tipo e valor (maior que zero:) das dimensões a serem definidas para a estrutura, bem como os valores a serem inseridos nela. Nesse último caso, é checado se o valor a ser inserido é um número. É checado também se o arquivo de entrada existe. Caso algum deles seja inválido, o programa para e gera uma mensagem de erro.

Para as funções que realizam operações sobre a estrutura, há uma checagem em relação às dimensões das matrizes. No caso da soma, é checado se a quantidade

de linhas e colunas da matriz A é igual a da matriz B. Na multiplicação, verifica-se se a quantidade de colunas em A é igual a quantidade de linhas em B. Caso o contrário, também é gerado um erro. Na operação de transposição, é checado apenas a dimensão da matriz, para ver se ela é nula ou não.

Por fim, a função que escreve o resultado da operação no arquivo checa se o arquivo de destino existe. Caso não, cria no lugar. Se essa criação falhar, é gerado um erro com uma mensagem.

5. Testes

5.1. Plano experimental

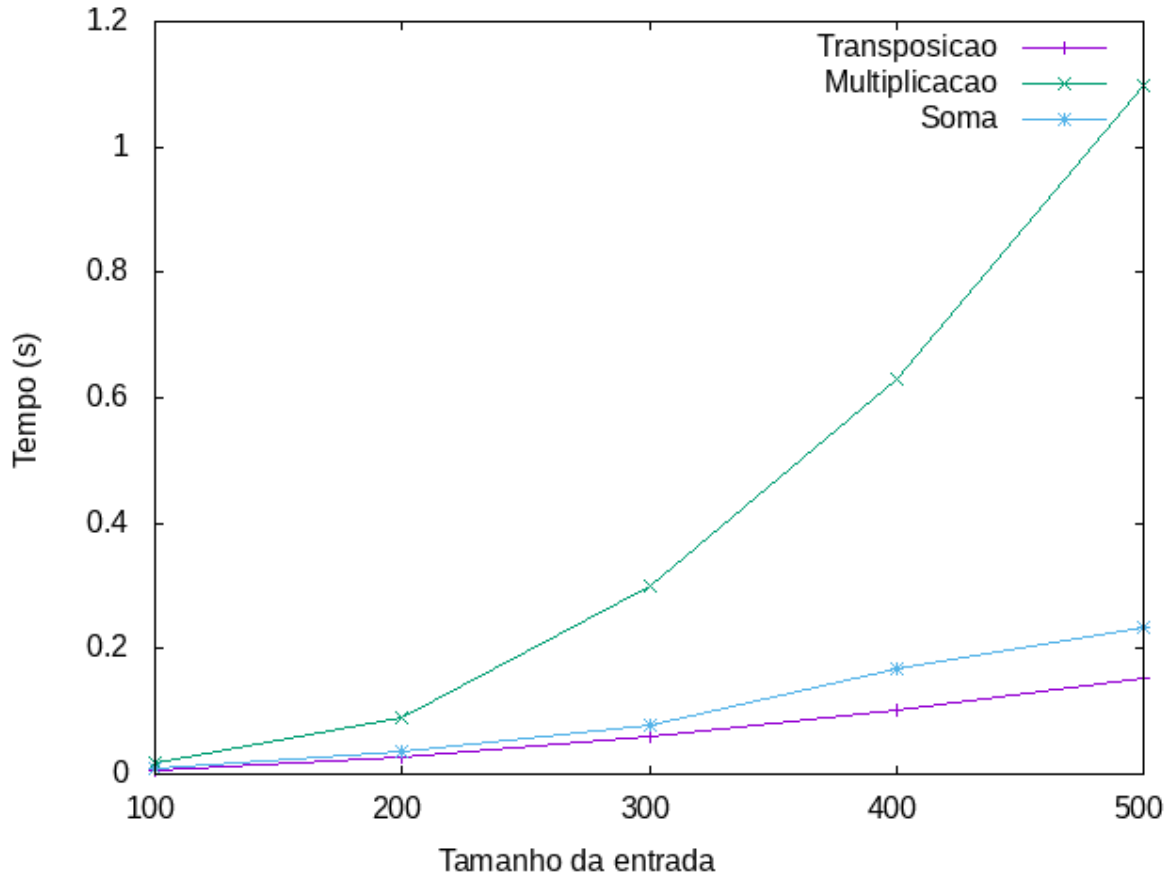
Para analisar a fundo o algoritmo e as operações (soma, multiplicação e transposição), foram realizados testes de desempenho computacional e análise de localidade de referência, juntamente com o padrão de acesso em memória. Os testes foram feitos com diferentes tamanhos de matrizes.

Para o teste de memória, foram utilizadas matrizes de dimensões 5x5. Já o teste de análise de desempenho foi dividido em duas etapas: a primeira analisando a evolução do tempo de execução de cada operação, na qual foram utilizadas matrizes quadradas de dimensões entre [100-500]; já a segunda etapa, analisou-se o tempo de execução de cada função de cada operação. Para tal, foi utilizado uma matriz de dimensão 1000x1000.

Esses experimentos foram realizados em um ambiente sem interferência de programas rodando em *background* que utilizam CPU ou memória RAM do computador. Os resultados podem ser vistos nas seções a seguir.

5.2. Resultados

5.2.1. Desempenho computacional



5.2.1.1. Soma

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
50.07	0.02	0.02	4	05.01	05.01	accessMatrix
25.04	0.03	0.01	2	05.01	05.01	initMatrixFromFile
25.04	0.04	0.01	1	10.01	10.01	sumMatrix
0.00	0.04	0.00	4	0.00	0.00	createMatrix
0.00	0.04	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.04	0.00	3	0.00	0.00	destroyMatrix
0.00	0.04	0.00	3	0.00	0.00	initNullMatrix
0.00	0.04	0.00	1	0.00	0.00	clkDifMemLog
0.00	0.04	0.00	1	0.00	0.00	desativaMemLog
0.00	0.04	0.00	1	0.00	0.00	finalizaMemLog
0.00	0.04	0.00	1	0.00	0.00	iniciaMemLog
0.00	0.04	0.00	1	0.00	0.00	parse_args
0.00	0.04	0.00	1	0.00	0.00	writeMatrixToFile

5.2.1.2. Multiplicação

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.97	11.46	11.46	1	11.46	11.46	multiplyMatrix
0.09	11.47	0.01	4	0.00	0.00	accessMatrix
0.09	11.48	0.01	1	0.01	0.01	writeMatrixToFile
0.00	11.48	0.00	4	0.00	0.00	createMatrix
0.00	11.48	0.00	3	0.00	0.00	defineFaseMemLog
0.00	11.48	0.00	3	0.00	0.00	destroyMatrix
0.00	11.48	0.00	3	0.00	0.00	initNullMatrix
0.00	11.48	0.00	2	0.00	0.00	initMatrixFromFile
0.00	11.48	0.00	1	0.00	0.00	clkDifMemLog
0.00	11.48	0.00	1	0.00	0.00	desativaMemLog
0.00	11.48	0.00	1	0.00	0.00	finalizaMemLog
0.00	11.48	0.00	1	0.00	0.00	iniciaMemLog
0.00	11.48	0.00	1	0.00	0.00	parse_args

5.2.1.3. Transposição

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
50.07	0.02	0.02	3	6.68	6.68	accessMatrix
25.04	0.03	0.01	1	10.01	10.01	transposeMatrix
25.04	0.04	0.01	1	10.01	10.01	writeMatrixToFile
0.00	0.04	0.00	3	0.00	0.00	createMatrix
0.00	0.04	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.04	0.00	2	0.00	0.00	destroyMatrix
0.00	0.04	0.00	1	0.00	0.00	clkDifMemLog
0.00	0.04	0.00	1	0.00	0.00	desativaMemLog
0.00	0.04	0.00	1	0.00	0.00	finalizaMemLog
0.00	0.04	0.00	1	0.00	0.00	iniciaMemLog
0.00	0.04	0.00	1	0.00	0.00	initMatrixFromFile
0.00	0.04	0.00	1	0.00	0.00	initNullMatrix
0.00	0.04	0.00	1	0.00	0.00	parse_args

5.2.2. Localidade de referência e acesso à memória

5.2.2.1. Soma

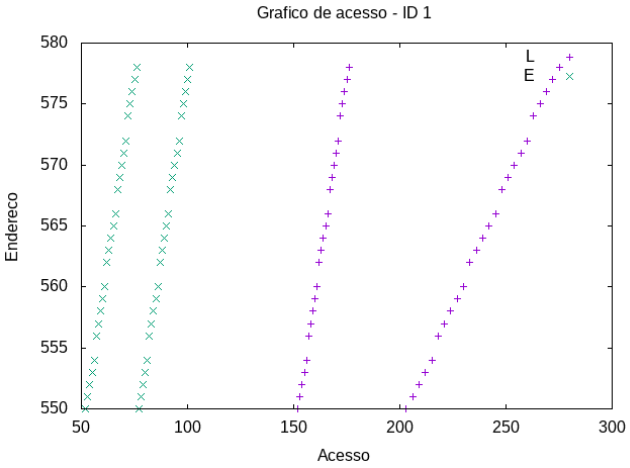


Grafico de acesso - ID 1
Endereco
Acesso
L
E

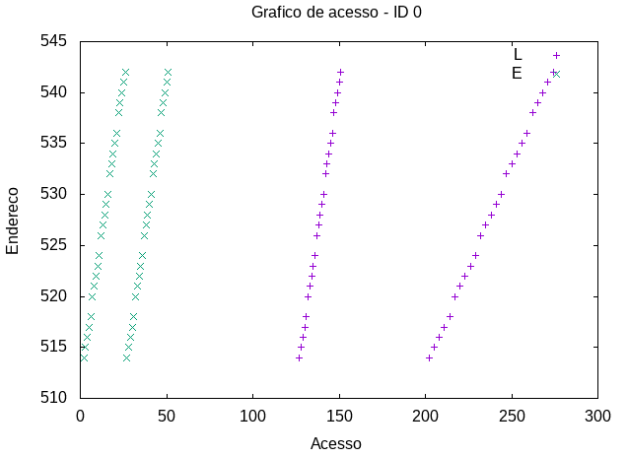


Grafico de acesso - ID 0
Endereco
Acesso
L
E

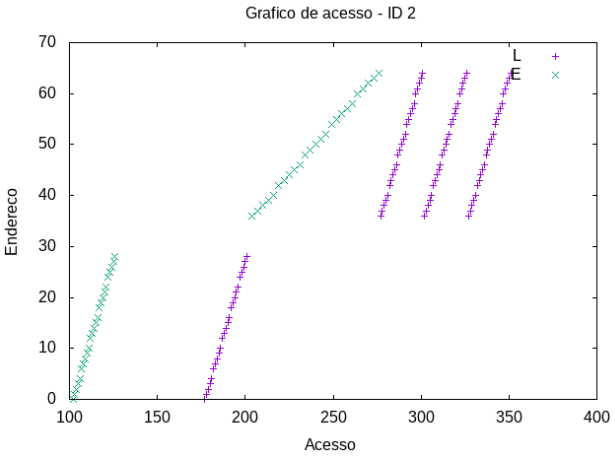
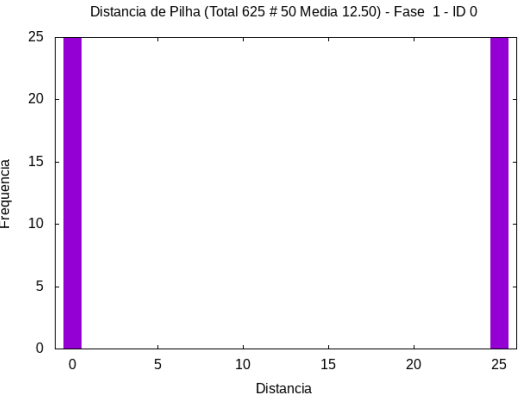
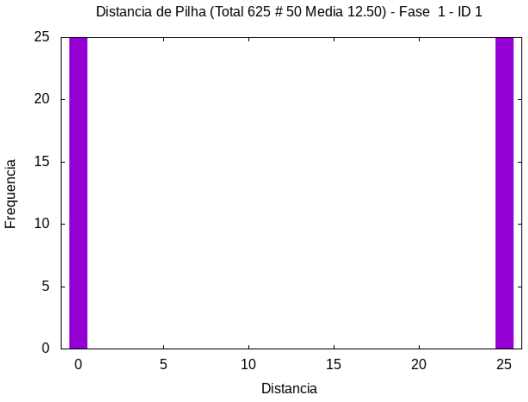
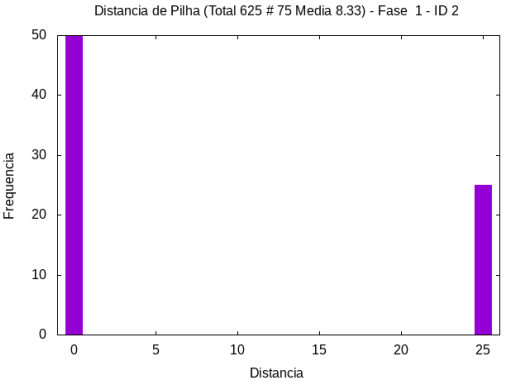
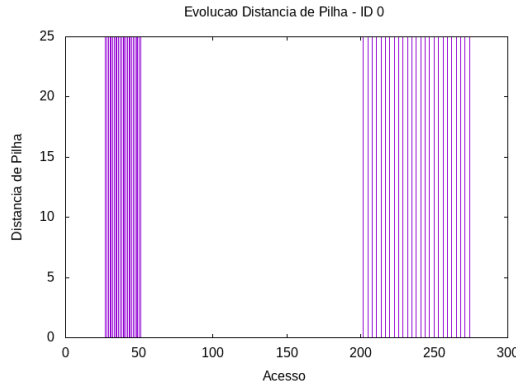


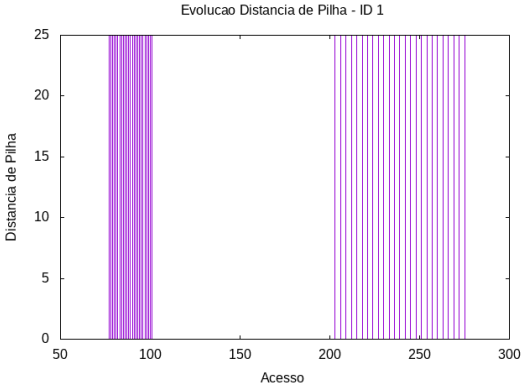
Grafico de acesso - ID 2
Endereco
Acesso
L
E

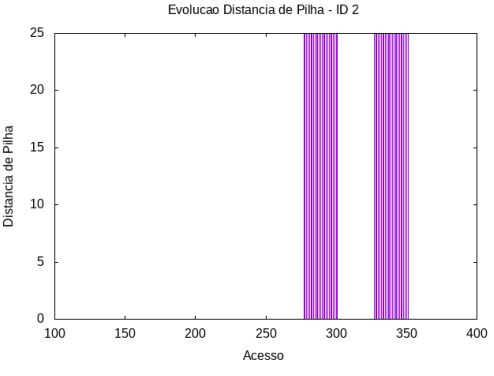

Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 0
Frequencia
Distancia


Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 1
Frequencia
Distancia

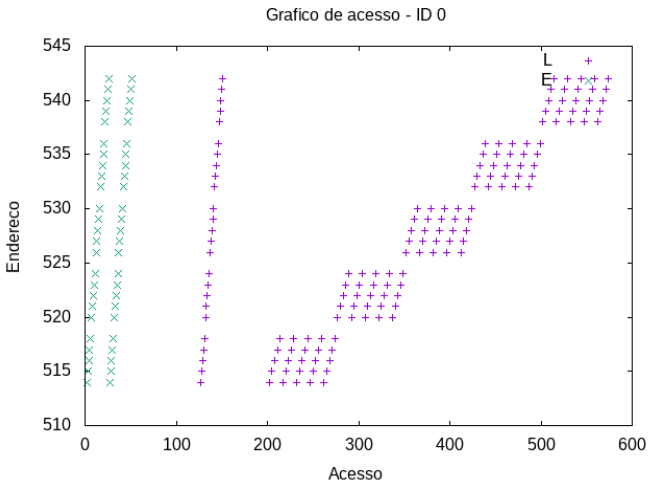

Distancia de Pilha (Total 625 # 75 Media 8.33) - Fase 1 - ID 2
Frequencia
Distancia

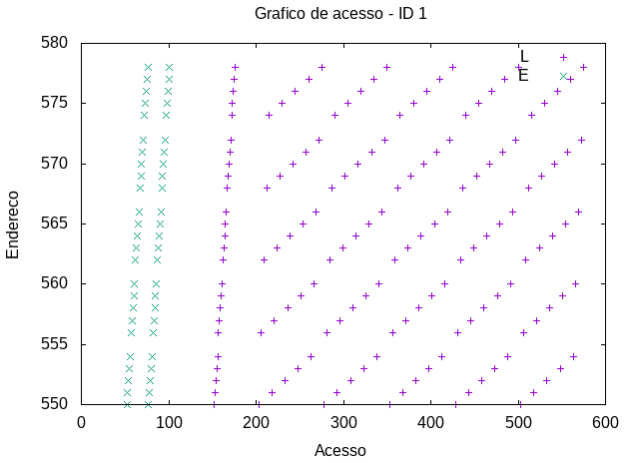

Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso

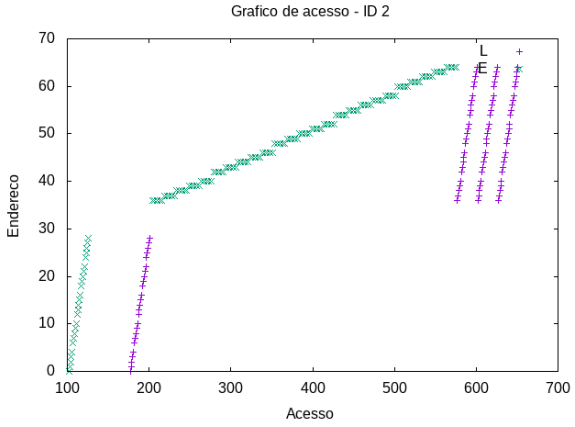

Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso

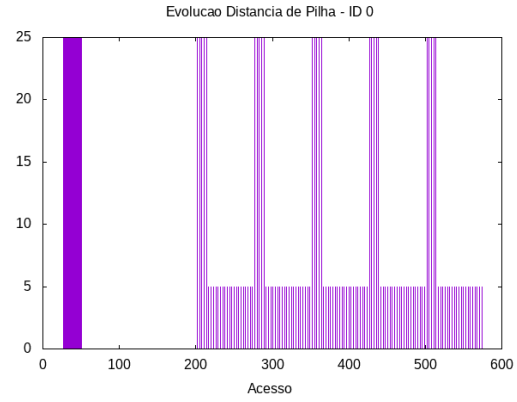

Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso

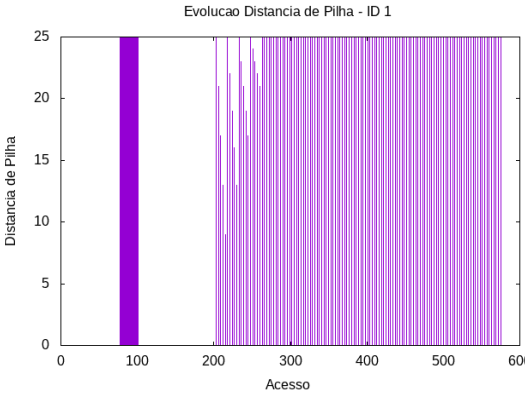
5.2.2.1. Multiplicação

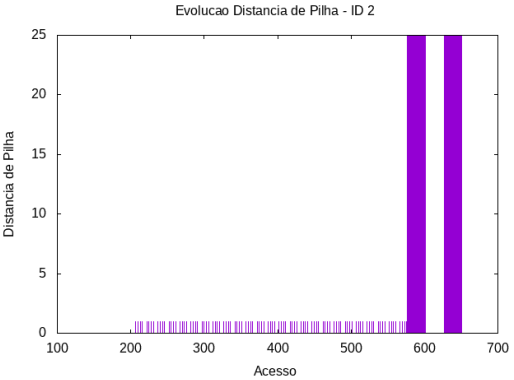


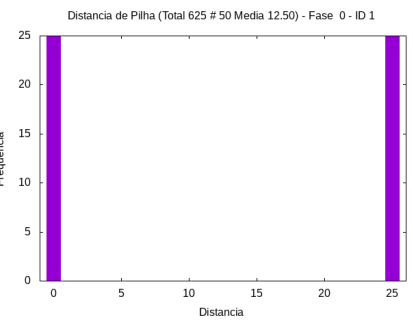


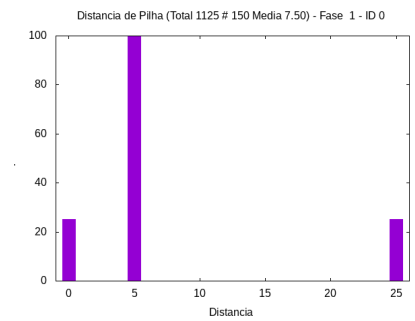


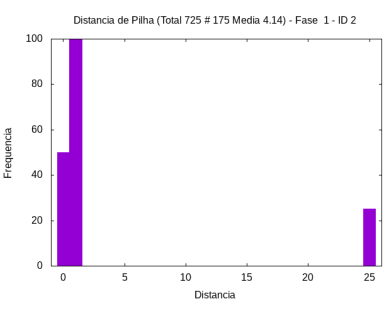


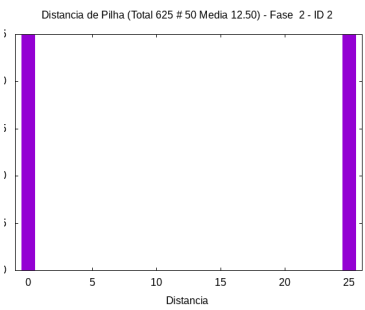




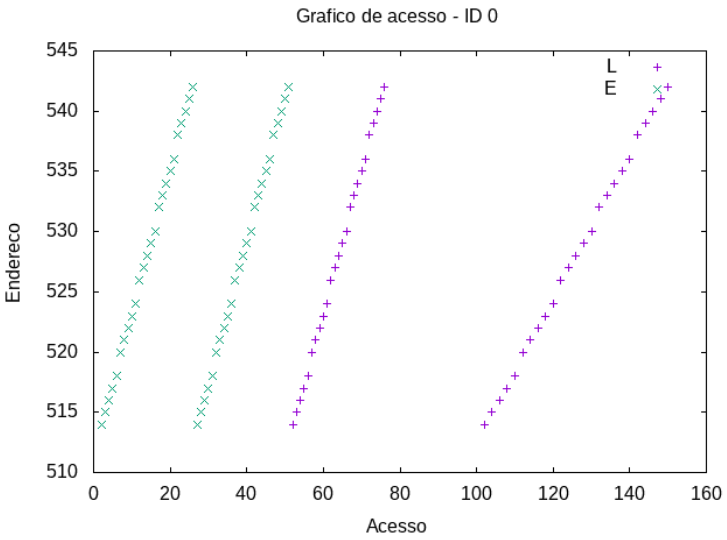


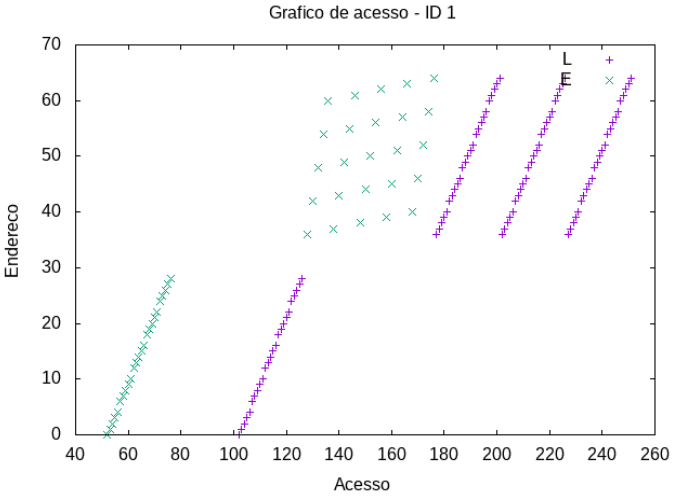


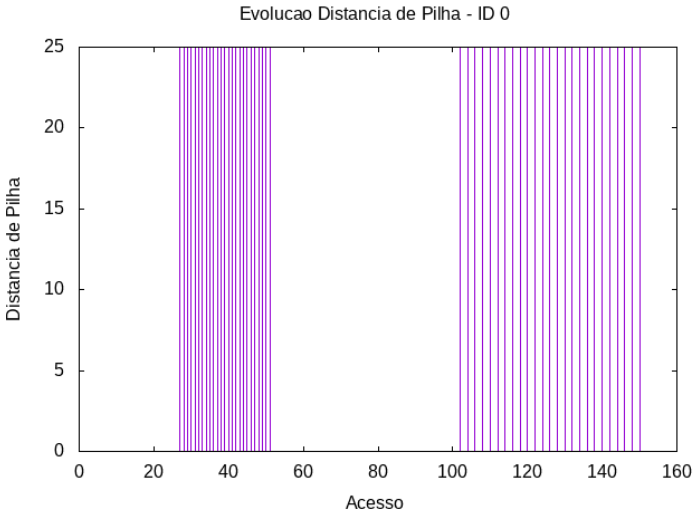


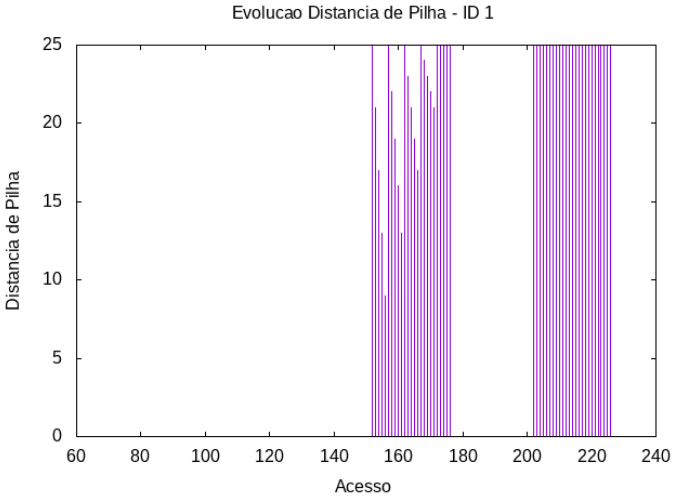


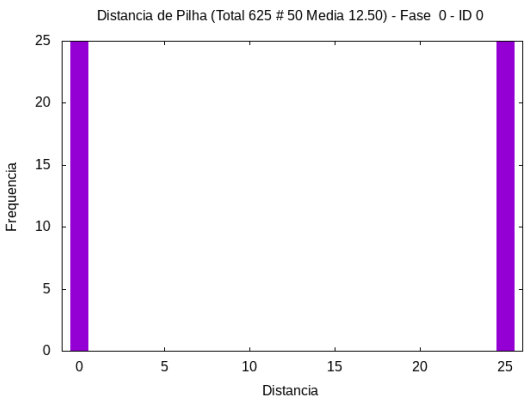
5.2.2.1. Transposição

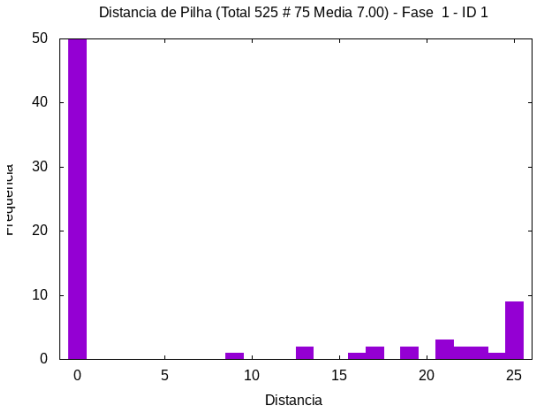


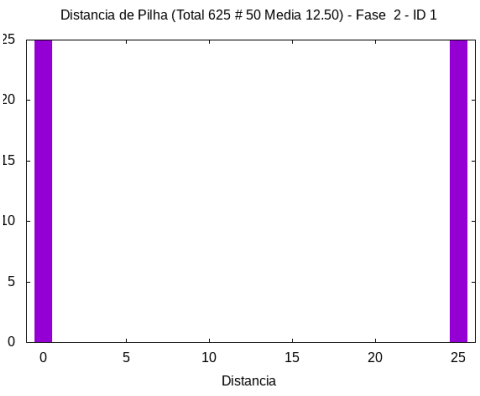












6. Análise experimental

6.1. Desempenho computacional

Na seção **5.2.1,** são apresentados os resultados de desempenho computacional para cada operação. O primeiro gráfico foi gerado com *outputs* do ***gprof,*** no qual é possível visualizar a complexidade de cada função.

Como descrito na seção **3**, tanto o algoritmo de transposição quanto o algoritmo de soma de matrizes são quadrados ($\Theta(n^2)$). No gráfico em questão, ambas as curvas dessas funções estão relativamente próximas umas das outras. Nas tabelas de tempo de execução, se desconsiderássemos as funções de "accessMatrix", que no geral, servem para a análise de acesso na memória, ambas detém um tempo considerável na execução total do programa.

Já para a operação de multiplicar matrizes, é possível perceber que ela sobressai perante as outras. Isso ocorre devido a sua complexidade ser $\Theta(n^3)$, como também descrito anteriormente. Com isso, à medida que aumentamos a quantidade da entrada, seu tempo de execução torna-se extremamente maior quando comparado às outras duas operações. O tempo de execução da função que realiza a multiplicação é responsável por quase 100% da duração do programa.

Uma possível melhoria na função de multiplicação, seria utilizar o algoritmo de ***Strassen***, que utiliza o paradigma de dividir-para-conquistar e possui complexidade de $\Theta(n^{2.8074})$, ou o algoritmo de ***Coppersmith-Winograd***, que possui complexidade $\Theta(n^{2.375})$. Em matrizes pequenas, essa melhoria seria imperceptível. Mas à medida que aumentamos as dimensões da entrada, o algoritmo se torna mais eficiente.

6.2. Localidade de referência e acesso à memória

6.2.1. Soma

Para a operação de soma, os IDs 0 e 1 apresentados nas imagens da seção **5.2.2.1** referem-se às matrizes que vão ser somadas. No mapa de acesso deles, é possível perceber uma fase inicial na qual são escritas, em cada posição da matriz por linha, o valor nulo e o valor vindo do arquivo de entrada. Depois, é feita uma leitura da matriz por linha e, por fim, a leitura dos seus valores para a soma. Nota-se que a leitura final é mais espaçada, pois é feita a leitura das duas matrizes ao mesmo tempo. O histograma e a evolução do acesso à memória e distância de pilha de ambas matrizes também é idêntico. Neles, é possível perceber o primeiro acesso a matriz no início, gerando a distância de pilha 0 e os acessos posteriores para leitura, gerando a distância de pilha 25, referente ao número total de posições na matriz.

Já para o ID 2, referente a matriz em que será escrito a soma, é possível ver um comportamento diferente. Primeiro, é possível ver no mapa que a matriz é inicializada duas vezes, sendo a primeira (parte “inferior” do mapa) ela é inicializada como nula, e na segunda (parte “superior”), como uma nova matriz na qual serão escritos os resultados das somas, por linha. No histograma, é registrado apenas a fase de soma e escrita na matriz. Na função soma, por criarmos uma nova matriz, é feita a leitura da matriz “antiga”, com distância de pilha 0 nessa fase; a escrita na nova matriz, com distância 0 também, e a leitura para escrita no arquivo, tendo distância 25.

6.2.1. Multiplicação

Para o algoritmo de multiplicação, os IDs 0 e 1 representam as matrizes a serem multiplicadas. É evidente no mapa de acesso a forma como a multiplicação ocorre. Para cada linha da matriz 0, uma posição da memória é lida e seu valor é multiplicado com seu correspondente em todas as coluna da matriz 1. Tal forma de acesso afeta a distância de pilha de ambas as matrizes. No caso de 0, a distância acaba de pilha durante a multiplicação acaba sendo 25 para o primeiro acesso e 5 para os acessos sequentes na linha, já que cada elemento da linha tinha sido acessado uma vez. No caso de 1, o acesso ocorre por cada elemento de uma coluna a cada iteração de uma linha de 0, o que faz a evolução da distância de pilha ter “saltos” durante a execução do programa, e a distância de pilha dessa matriz ser menor do que a distância no ID 0.

Já para o ID 2, que representa a matriz em que será escrito o resultado da multiplicação, o acesso à memória se torna diferente. Essa matriz é inicializada primeiro como nula e depois é sobrescrita com outra matriz (por isso existe uma fase de acesso “em cima” e “em baixo” no mapa). Durante o processo de multiplicação, cada posição da matriz é acessada 5 vezes, para que ela possa guardar a soma total da multiplicação da linha de 0 pela coluna de 1. O acesso se dar por linha, por isso é possível perceber no mapa a evolução do acesso com escritas de 5 a 5 em cada posição. Nesse caso, o primeiro acesso em uma posição da pilha terá distância de 25 e o restante de 5. O mesmo se repete para as outras linhas.

6.2.1. Transposição

Para a operação de transposição, existem duas matrizes a serem analisadas. A matriz de ID 0 possui um padrão de acesso semelhante aos IDs 0 e 1 da operação de soma. Primeiro é feita a escrita e leitura dessa matriz, e durante a operação de transposição, é feita a leitura do valor dessa matriz para a escrita em outra matriz (por isso essa parte está mais espaçada no mapa). A distância de pilha é 0 para o primeiro acesso e 25 para a leitura (nessa fase de transposição).

Já para o ID 1, que refere-se a matriz em que será escrita a matriz transposta de 0, a análise muda. O acesso nessa matriz ocorre por coluna, o que é evidenciado pelo mapa de acesso e pela evolução da distância de pilha, que apresenta saltos, representado o acesso em diferentes colunas. Isso afeta a distância de pilha total do algoritmo, que acaba sendo menor que o algoritmo de soma, que possui a mesma complexidade de tempo.

7. Conclusões

O presente trabalho propôs-se a analisar os algoritmos de operações básicas em uma das estruturas de dados mais importantes, as matrizes. Para tal, foram realizadas as análises de desempenho computacional e uso da memória, evidenciando o comportamento dessas matrizes diante de diferentes entradas. Com isso, foi possível comparar a análise experimental com a teórica feita previamente, em relação a complexidade de tempo dos algoritmos e a forma que eles acessam a memória.

A análise feita sobre a matriz acaba se tornando mais complexa, devido a sua estrutura em duas dimensões. No entanto, com a geração de gráficos e tabelas feitas durante as análises, é possível visualizar o comportamento da estrutura com diferentes tamanhos de entrada ao longo da execução do programa, tornando a análise mais simples.

8. Bibliografia

CHORMEN, Thomasb H. *et al*. **Algoritmos - Teoria e Prática**. 3ª edição.
Capítulo 3: Crescimento de funções. Editora Elsevier, 10 de abril de 2012.

APÊNDICE A - INSTRUÇÕES PARA COMPILAÇÃO E EXECUÇÃO

A compilação do programa pode acontecer de duas maneiras distintas:

- Dentro da root do projeto (TP), executar: **make bin/matop;**
- ou gerar cada arquivo objeto e binário separadamente:
    1) gcc -pg -Wall -c -I include -o obj/matop.o src/matop.c
    2) gcc -pg -Wall -c -I include -o obj/mat.o src/mat.c
    3) gcc -pg -Wall -c -I include -o obj/memlog.o src/memlog.c
    4) gcc -pg -o bin/matop obj/matop.o obj/mat.o obj/memlog.o -lm .

Assim, basta executar o programa com o comando:

bin/matop -l -t -p ${output log} **-1** ${input matriz 1} **-2** ${input matriz 2} **-o** ${output da operação}

O parâmetro ***-t*** indica que a operação a ser realizada é a de transposição. Para operação de soma é o parâmetro **-s** e multiplicação **-m**. Os *inputs* e *outputs* representam o caminho para os arquivos de origem ou destino. Esse caminho pode ser relativo ou absoluto. Para o caso da operação de transpor matriz, o *input matriz 2* é desnecessário e será ignorado. Existe o parâmetro **-l** para habilitar a análise de memória do programa.