Trabalho Prático 0: Operações com matrizes alocadas dinamicamente

Gabriel Franco Jallais

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais(UFMG)

Belo Horizonte – MG – Brasil

gfjallais@ufmg.br

1. Introdução

O problema proposto foi realizar operações básicas com matrizes alocadas dinamicamente, no caso, soma, multiplicação e transposição de matrizes, o problema, em si, é simples matematicamente, mas o interessante é analisar o custo dessas operações tanto de memória quanto de tempo conforme o tamanho da entrada aumenta, como a memória é acessada e qual a distância de pilha quando são feitas as operações, quando elas são feitas em matrizes dinamicamente alocadas, em suma, esse foi o objetivo do trabalho.

Para tal, foi criado um TAD (Tipo Abstrato de Dados) para a matriz, contendo um ponteiro de ponteiros para alocar a matriz dinamicamente, além de variáveis inteiras para as suas respectivas dimensões e um id, para melhor reconhecimento de cada matriz no processo de análise de desempenho. Além disso, foram criadas funções para realizar as operações necessárias e desalocar as matrizes ao final.

Importante ressaltar que as entradas são feitas por meio de arquivos .txt, em que, na primeira linha, há as dimensões ***nxm*** da matriz, e as próximas n linhas contém os valores para cada elemento da matriz.

Nas próximas seções serão dados mais detalhes quanto a implementação, a análise de complexidade das operações, além de quais foram as estratégias de robustez adotadas, quais testes foram feitos, junto a análise experimental, por fim, é apresenta a conclusão, a bibliografia utilizada e instruções para compilação e execução.

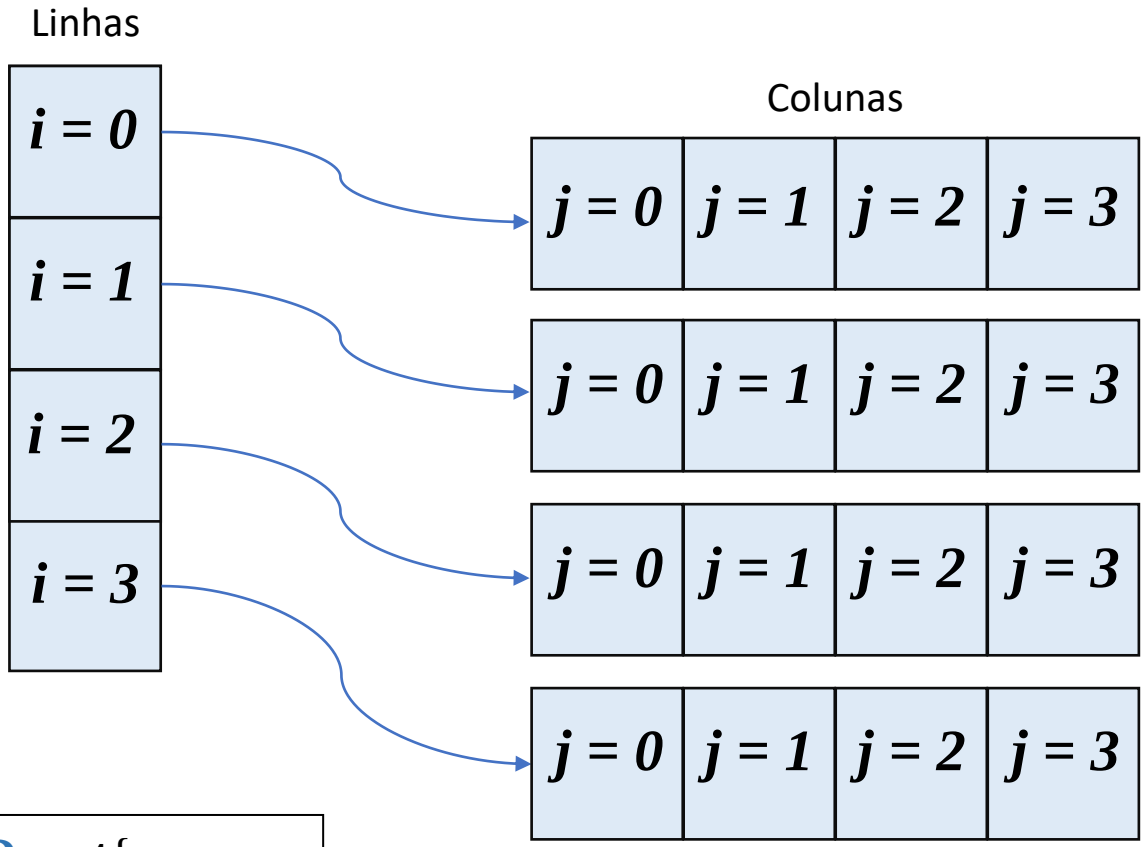
2. Implementação

O programa foi desenvolvido na linguagem C, compilada pelo compilador GCC da GNU Compiler Collection.

Foi compilado e executado no sistema operacional Ubuntu 20.04 dentro do WSL2 no Windows 11, com um processador i7-1165G7 com 16 GB de memória RAM.

2.1 Estrutura de Dados

A implementação do programa foi feito com base na estrutura de dados vetor, no caso um vetor de duas dimensões, ou seja, uma matriz, dinamicamente alocado, isto é, não há um tamanho pré-estabelecido, estático para a matriz, de forma que seja limitado ou que desperdice espaço não utilizado, o tamanho da matriz e a sua alocação são definidos conforme a entrada do programa. Para aplicá-la foi utilizado um TAD, com um ponteiro de ponteiros para alocar a matriz, variáveis inteiras para guardar suas respectivas dimensões, além de um id para facilitar sua identificação. Foi feita, primeiramente, a alocação das linhas da matriz, e em seguida de suas colunas, segue um diagrama da estrutura.




```
TAD mat{
        double ** m;
        int tamx, tamy;
        int id;
}
```

2.2 Funções

void criaMatriz(*mat_tipo* * *mat*, int *tx*, int *ty*, int *id*) :

Descricao: cria matriz com dimensoes tx X ty
Entrada: mat, tx, ty, id

Saida: mat

Na função é feita a alocação da matriz utilizando-se a função malloc, e são atribuídos os valores passados como argumentos aos atributos do TAD

double acessaMatriz(*mat_tipo * mat*) :

Descricao: cria matriz com dimensoes tx X ty
Entrada: mat, tx, ty, id
Saida: mat

São feitos dois loops para acessar todos os elementos da matriz, função útil para o registro de acesso e análise de localidade de memória.

void somaMatrizes(*mat_tipo *a, mat_tipo *b, mat_tipo *c*) :

Descricao: soma as matrizes a e b e armazena o resultado em c
Entrada: a, b
Saida: c

Apenas são feitas checagens de integridade das entradas e dos requisitos para a soma, no caso a compatibilidade das dimensões, em seguida são usados dois for para efetuar a soma das matrizes a e b e seu resultado é armazenado na matriz c.

void multiplicaMatrizes(*mat_tipo *a, mat_tipo *b, mat_tipo *c*) :

Descricao: multiplica as matrizes a e b e armazena o resultado em c
Entrada: a,b
Saida: c

Da mesma forma que na função somaMatriz(), são feitas checagens para que a operação possa ser efetuada, no caso que o número de colunas de a seja igual ao número de linhas de b, e que a matriz c não seja nula, após isso, para ser feito o produto seguido da atribuição aos elemento da matriz c, são feitos três loops.

void transpoeMatriz(*mat_tipo *a*) :

Descricao: transpoe a matriz a
Entrada: a
Saida: a

São feitas realocações de memória de acordo com a maior dimensão da matriz, para que assim possa ser feita a transposição, que é efetuada com dois for e o uso da função ELEMSWAP() - **ELEMSWAP(*x,y*) (x+=y,y=x-y,x-=y)** – após, isso são feitas realocações para que não sejam desperdiçados espaços na memória, além de ajustes nas variáveis tamx e tamy da matriz a. As realocações não são necessárias para matrizes quadradas(***nxn***).

void destroiMatriz(*mat_tipo *a*) :

Descricao: destroi a matriz a, que se torna inacessível
Entrada: a
Saida: a

Primeiramente, é feita uma checagem para verificar se a matriz já foi destruída anteriormente, se não, é feito um for para desalocar cada coluna e a todos os atributos do TAD é atribuído -1.

void leMatriz(*mat_tipo *a*, char **arq*, int *id*) :

Descricao: cria uma matriz a partir de um arquivo .txt
Entrada: a, arq, id
Saida: nenhuma

Como a entrada do programa é dada por meio de arquivos, a função apenas os abre, checa se abriram corretamente, cria uma matriz por meio da função criaMatriz(), e atribui os valores lidos do arquivo, cujo nome foi passado como parâmetro (arq), utilizando dois loops.

void geraOutput(*mat_tipo *c*, char **arq*) :

Descricao: cria uma matriz de saída a partir do resultado das operacoes efetuadas
Entrada: c, arq
Saida: nenhuma

Para visualizar os resultados das operações de maneira melhor, a função cria arquivos .txt, no mesmo formato dos de entrada. A função possui a mesma estrutura que a função leMatriz(), só que, ao invés de ler os elementos do arquivos, os elementos do TAD são escritos no arquivo cujo nome foi passado como parâmetro.

3. Análise de Complexidade

Para a análise de complexidade do programa, serão levadas em conta apenas as funções referentes às operações feitas com matrizes, no caso, soma, multiplicação e transposição. Será feita apenas a análise de complexidade em relação ao tempo de execução, visto que até então, a análise de complexidade de espaço não foi tão bem abordada em sala.

- somaMatrizes()

A começar pela operação de soma, a operação relevante para a sua análise será a soma de um elemento da matriz a com um elemento da matriz b e a sua respectiva atribuição ao elemento de mesmos index da matriz c, como descrito na seção anterior,

na função somaMatrizes(), são feitos dois loops for, considerando uma matriz quadrada de entrada n, cada operação de soma e atribuição, ***O(1)***, será realizada n*n vezes, ou seja, a operação de soma de matrizes possui complexidade $O(n^2)$, caso a entrada fosse ***mxn***, sua complexidade seria ***O(mn)***.

- **multiplicaMatrizes()**

Partindo para a próxima operação em matrizes, a operação que será considerada relevante para a análise de complexidade, será o produto entre um elemento da matriz a com um elemento da matriz b e a sua respectiva atribuição ao elemento de mesmos index da matriz c, ***O(1)***, para tal, conforme a definição matemática para a multiplicação de matrizes, é necessário varrer todas as colunas de cada linha da primeira matriz e somar o produto de cada um desses elementos com os elementos de cada linha de todas as colunas da matriz b. Por exemplo, para uma matriz A(mxn) e uma matriz B(nxm), será necessário multiplicar os n elementos de cada linha de A com os n elementos de cada coluna de B, assim serão feitas n*n*m operações consideradas relevantes. Portanto, considerando uma matriz quadrada de entrada n, a ordem de complexidade da função multiplicaMatrizes() é $O(n^3)$.

- transpoeMatriz()

Por último, analisemos a complexidade da transposição de uma matriz, será considerada a operação de "troca" de elementos como a operação relevante para análise, essa troca teria complexidade, ***O(1)***. Como para essa operação é necessário apenas varrer toda a matriz e trocar de posição os valores presentes nas linhas pelos valores presentes nas colunas da matriz, para varrer todo o vetor serão necessários dois loops, indo de 0 até as respectivas dimensões de cada lado da matriz, logo, considerando uma matriz quadrada de ordem n, teremos que a complexidade da função **transpoeMatriz()**, será $O(n^2)$, da mesma forma que **somaMatriz()**, caso a entrada fosse ***mxn***, sua complexidade seria ***O(mn)***.

4. Estratégias de Robustez

Para garantir a boa operação do programa, isto é, que ele seja robusto para tolerar erros do usuário e se defender de tentativas de "ataque" que visam achar falhas no programa para explorá-las, foram tomadas as seguintes estratégias.

A começar pela compilação e execução do programa, a função parseArgs() é responsável por verificar a passagem de argumentos na linha de comando, no apêndice serão apresentados mais detalhes quanto a compilação e execução do programa, mas a entrada é basicamente:

(EXE) (OPERAÇÃO) (COMANDO OPICIONAL 1) (SAÍDA DO COMANDO OPCIONAL) (COMANDO OPICIONAL 2) -1 (ARQUIVO DE ENTRADA DA PRIMEIRA MATRIZ) -2 (ARQUIVO DE ENTRADA DA SEGUNDA MATRIZ) -o (ARQUIVODE DE SAÍDA COM OS RESULTADOS DA OPERAÇÃO)

EXE = matop.exe
OPERAÇÃO = -s, -m ou -t
COMANDO OPICIONAL 1 = -p, para a geração de arquivo .out com dados de desempenho computacional
COMANDO OPICIONAL 2 = -l, para a geração de mapas de acesso a memória
ARQUIVOS DE ENTREDA E SAÍDA - todos em .txt

Logo, a função parseArgs() verifica essa entrada para que ela siga este padrão, sem que mais de uma operação seja passada, que os comandos sejam validos, ou seja, os comandos previamente estabelecidos, e que os arquivos passados sejam válidos, que eles
abram sem erro.

Ademais, para cada função/operação descrita na seção 2, são feitas checagens para que as entradas cumpram os requisitos para as operações matemáticas, isto é, no caso de matrizes, que as dimensões sejam válidas na criação da matriz, na soma e na multiplicação, além disso, nas funções que envolvem alocação, realocação e desalocação, são feitas asserções para garantir que a alocação ou realocação funcionaram corretamente, isto é, que elas não tenham retornado um valor nulo, e na desalocação, é verificado se ela já foi feita previamente.

5. Testes

Para analisar desempenho computacional do programa, além de fazer a análise do padrão de acesso à memória e localidade de referência, foram feitos os seguintes testes:

Para a análise de desempenho, a análise de memória não foi ativada, para melhor fidelidade e representação do desempenho do programa, e foram dadas entradas com matrizes quadradas geradas aleatoriamente, de ordem 10, 20, 30, 40 e 50.

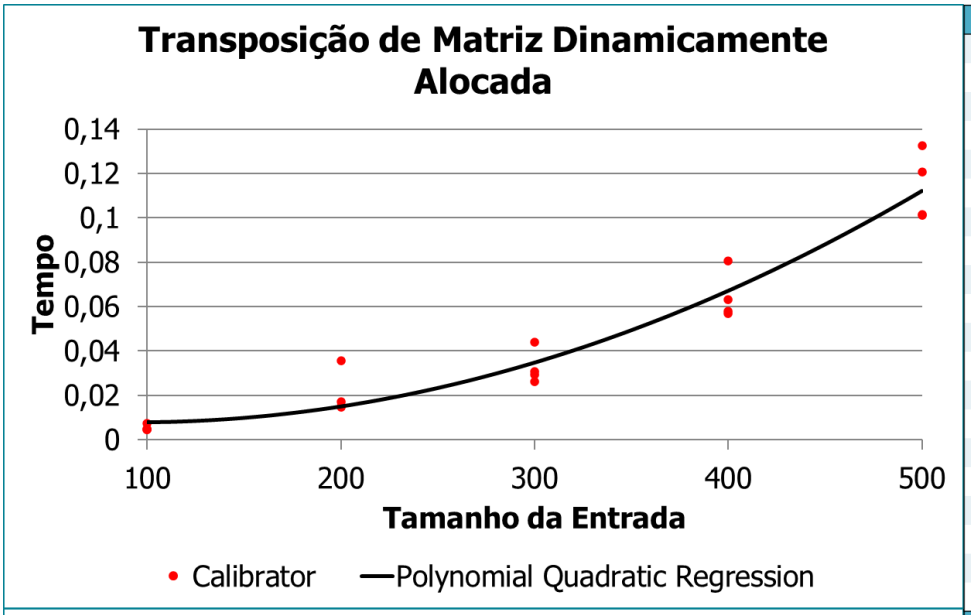
Já para a análise de acesso e localidade de memória, foi utilizada uma matriz também quadrada de ordem 10, dessa forma fica mais fácil visualizar os seus padrões quando observados os gráficos e histogramas gerados.

6. Análise Experimental

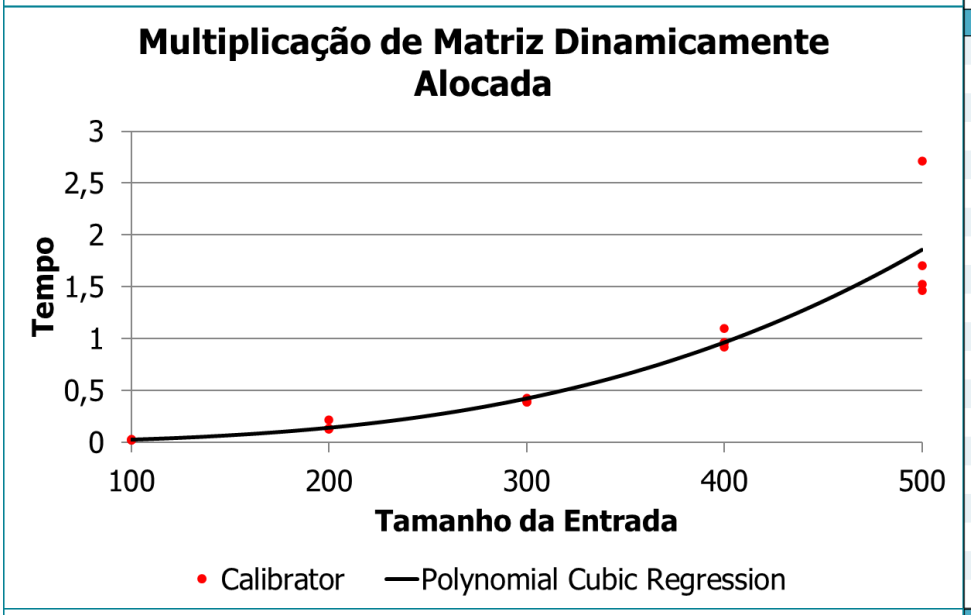
Para efetuar a análise do programa em questão foram utilizados os softwares memlog e analisamem, que foram responsáveis por mapear o tempo de execução e o acesso à memória. As operações consideradas relevantes para a avaliação foram as mesmas descritas na seção 3.

6.1 Desempenho Computacional

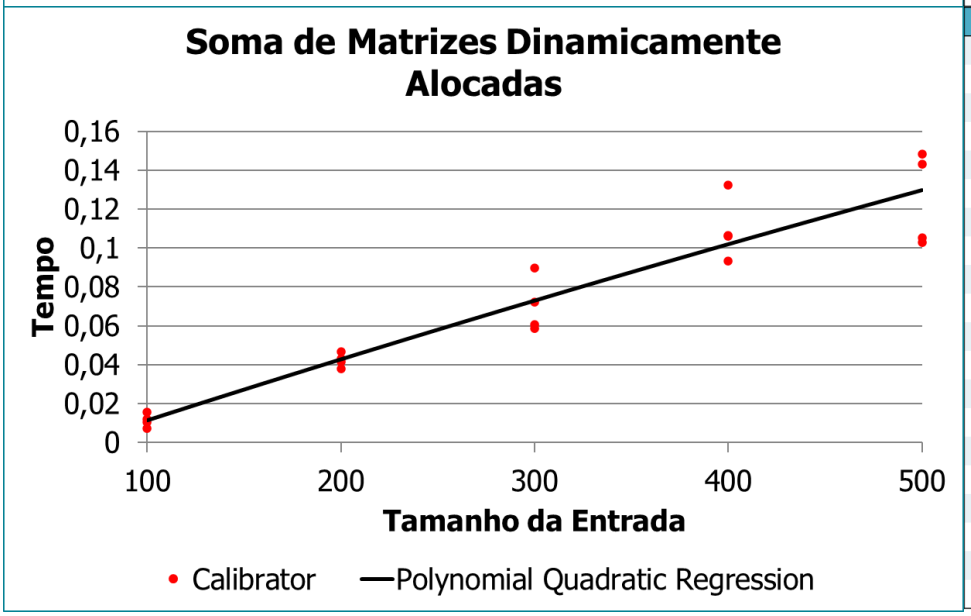
Para verificar a análise de complexidade feita previamente, foi feita uma sequência de testes e traçada uma curva para capturar o comportamento das operações em função do aumento da entrada, seguem os gráficos:



Tamanho da Entrada	Tempo
100	0.005061502
100	0.007362518
100	0.004478378
100	0.004653143
200	0.01704223
200	0.035533407
200	0.014550247
200	0.01494548
300	0.030676495
300	0.043943903
300	0.026040443
300	0.029306893
400	0.056959418
400	0.080656813
400	0.05783121
400	0.06320587
500	0.100990504
500	0.101370083
500	0.132640239
500	0.120752466



Tamanho da Entrada	Tempo
100	0.028418576
100	0.026418252
100	0.025858239
100	0.025707498
200	0.21489869
200	0.137412506
200	0.135589046
200	0.126904715
300	0.38569239
300	0.391778988
300	0.395401992
300	0.429944494
400	0.967425059
400	0.922367437
400	0.959620293
400	1.097918841
500	1.465768806
500	1.528380258
500	1.700548808
500	2.711960591



Tamanho da Entrada	Tempo
100	0.010435455
100	0.015554948
100	0.012055198
100	0.007372273
200	0.043170319
200	0.046506811
200	0.037890367
200	0.041406386
300	0.07222306
300	0.060555651
300	0.089552474
300	0.058776163
400	0.106329324
400	0.106057751
400	0.093333341
400	0.1325215
500	0.105354115
500	0.148260651
500	0.102669986
500	0.143029749

Os resultados para cada entrada são oscilantes devido ao hardware e às outras atividades que estavam sendo executadas em paralelo no computador, levando isso em consideração foram feitos alguns testes para tentar melhorar a precisão da análise.

6.1.1 Resultados do gprof

Para analisar os resultados do gprof, utilizaremos matrizes quadrada de ordem 1000 e 2000, para haver uma visualização mais direta do funcionamento do programa e para que o tempo de execução seja grande o suficiente para gerar resultados visíveis, para matrizes de ordem maior o tempo de execução ficaram muito grandes e por isso não as utilizaremos.

Matriz de ordem 1000:

Multiplicação de Matrizes

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.81	9.53	9.53	1	9.53	9.53	multiplicaMatrizes
0.21	9.55	0.02	4	0.01	0.01	acessaMatriz
0.10	9.56	0.01	2	0.01	0.01	leMatriz
0.00	9.56	0.00	3	0.00	0.00	criaMatriz
0.00	9.56	0.00	3	0.00	0.00	defineFaseMemLog
0.00	9.56	0.00	3	0.00	0.00	destroiMatriz
0.00	9.56	0.00	1	0.00	0.00	clkDifMemLog
0.00	9.56	0.00	1	0.00	0.00	desativaMemLog
0.00	9.56	0.00	1	0.00	0.00	finalizaMemLog
0.00	9.56	0.00	1	0.00	0.00	iniciaMemLog
0.00	9.56	0.00	1	0.00	0.00	parseArgs

Soma de Matrizes:

Flat profile

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.38	0.01	0.01	4	2.50	2.50	acessaMatriz
33.38	0.02	0.01	2	5.01	5.01	leMatriz
33.38	0.03	0.01	1	10.01	10.01	somaMatrizes
0.00	0.03	0.00	3	0.00	0.00	criaMatriz
0.00	0.03	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.03	0.00	3	0.00	0.00	destroiMatriz
0.00	0.03	0.00	1	0.00	0.00	clkDifMemLog

0.00	0.03	0.00	1	0.00	0.00	desativaMemLog
0.00	0.03	0.00	1	0.00	0.00	finalizaMemLog
0.00	0.03	0.00	1	0.00	0.00	iniciaMemLog
0.00	0.03	0.00	1	0.00	0.00	parseArgs

Transposição de Matrizes:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
50.06	0.01	0.01	1	5.01	5.01	transpoeMatriz
0.00	0.01	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.01	0.00	2	0.00	0.00	acessaMatriz
0.00	0.01	0.00	1	0.00	0.00	clkDifMemLog
0.00	0.01	0.00	1	0.00	0.00	criaMatriz
0.00	0.01	0.00	1	0.00	0.00	desativaMemLog
0.00	0.01	0.00	1	0.00	0.00	destroiMatriz
0.00	0.01	0.00	1	0.00	0.00	finalizaMemLog
0.00	0.01	0.00	1	0.00	0.00	iniciaMemLog
0.00	0.01	0.00	1	0.00	0.00	leMatriz
0.00	0.01	0.00	1	0.00	0.00	parseArgs

Matriz de Ordem 2000:

Multiplicação de Matrizes

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.97	125.49	125.49	1	125.49	125.49	multiplicaMatrizes
0.08	125.59	0.10	4	0.03	0.03	acessaMatriz
0.08	125.69	0.10	2	0.05	0.05	leMatriz
0.00	125.69	0.00	3	0.00	0.00	criaMatriz
0.00	125.69	0.00	3	0.00	0.00	defineFaseMemLog
0.00	125.69	0.00	3	0.00	0.00	destroiMatriz
0.00	125.69	0.00	1	0.00	0.00	clkDifMemLog
0.00	125.69	0.00	1	0.00	0.00	desativaMemLog
0.00	125.69	0.00	1	0.00	0.00	finalizaMemLog
0.00	125.69	0.00	1	0.00	0.00	iniciaMemLog
0.00	125.69	0.00	1	0.00	0.00	parseArgs

Soma de Matrizes:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
44.50	0.08	0.08	4	20.03	20.03	acessaMatriz
44.50	0.16	0.08	1	80.10	80.10	somaMatrizes
11.13	0.18	0.02	2	10.01	10.01	leMatriz
0.00	0.18	0.00	3	0.00	0.00	criaMatriz
0.00	0.18	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.18	0.00	3	0.00	0.00	destroiMatriz
0.00	0.18	0.00	1	0.00	0.00	clkDifMemLog
0.00	0.18	0.00	1	0.00	0.00	desativaMemLog
0.00	0.18	0.00	1	0.00	0.00	finalizaMemLog
0.00	0.18	0.00	1	0.00	0.00	iniciaMemLog
0.00	0.18	0.00	1	0.00	0.00	parseArgs

Transposição de Matrizes:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
36.41	0.04	0.04	2	20.03	20.03	acessaMatriz
36.41	0.08	0.04	1	40.05	40.05	leMatriz
27.31	0.11	0.03	1	30.04	30.04	transpoeMatriz
0.00	0.11	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.11	0.00	1	0.00	0.00	clkDifMemLog
0.00	0.11	0.00	1	0.00	0.00	criaMatriz
0.00	0.11	0.00	1	0.00	0.00	desativaMemLog
0.00	0.11	0.00	1	0.00	0.00	destroiMatriz
0.00	0.11	0.00	1	0.00	0.00	finalizaMemLog
0.00	0.11	0.00	1	0.00	0.00	iniciaMemLog
0.00	0.11	0.00	1	0.00	0.00	parseArgs

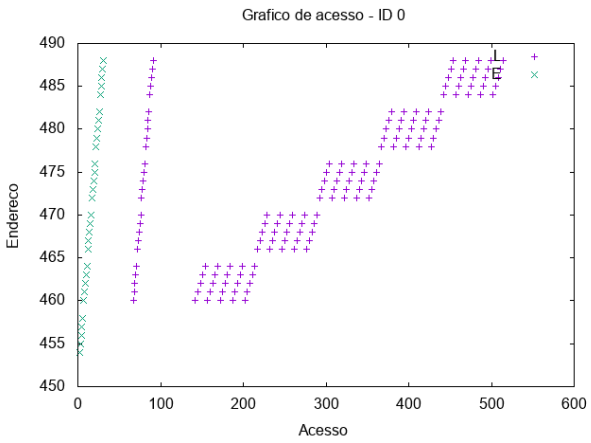
Pode-se observar que o comportamento das funções é bem similar nas duas matrizes, a função multiplicaMatrizes() por ser muito “custosa” acaba por dominar a tabela, utilizando quase 100% do tempo da execução do programa, vê-se também o tempo que a operação demora para ser executada com uma entrada deste tamanho em comparação com as outras operações, demorando um minuto e meio na matriz de ordem 1000 e pouco mais que dois minutos na matriz de ordem 2000, enquanto as outras operações nem chegam perto de um segundo. Em ambos os casos, as funções somaMatrizes() e transpoeMatriz() ocupam parte razoável da execução mas não se comparam a multiplicaMatrizes(), chegando a competir com as funções de leitura e acesso de matrizes.

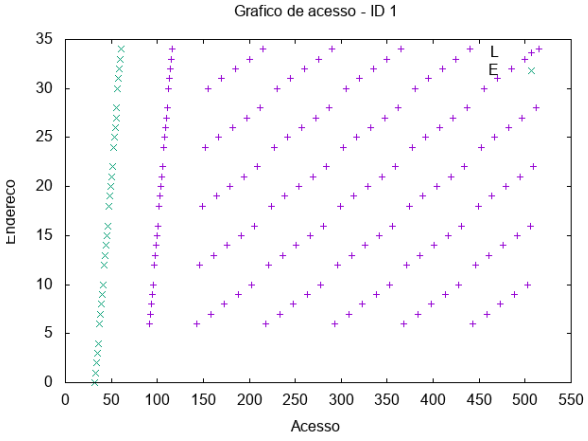
6.2 Análise de Padrão de Acesso à Memória e Localidade de Referência

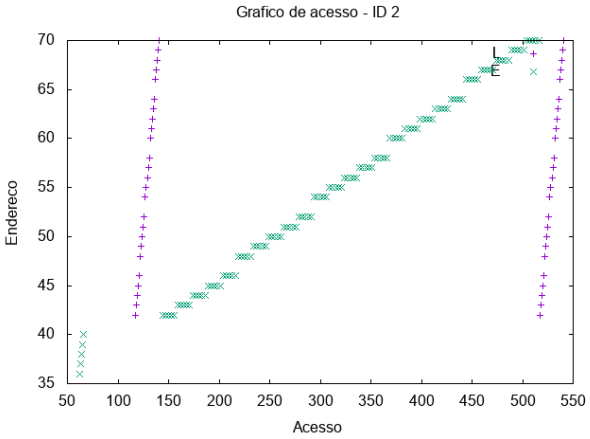
Agora, para entender melhor como o programa acessa memória e a sua localidade de referência, observaremos os gráficos gerados por meio do software gnuplot, e cujos dados foram fornecidos pelo programa analisamem, utilizando uma matriz quadrada de ordem 5, para uma visualização menos poluída.

Primeiramente, cabe dizer que, o id 0 é referente à matriz passada no primeiro arquivo de entrada, o id 1 à segunda, e o id 2 ao arquivo de saída com os resultados da operação. Os gráficos mostram como foi feito o acesso à memória ao decorrer do tempo, pontos (+) próximos no eixo x indicam endereços que foram acessados em intervalos próximos de tempo.

multiplicaMatrizes() :

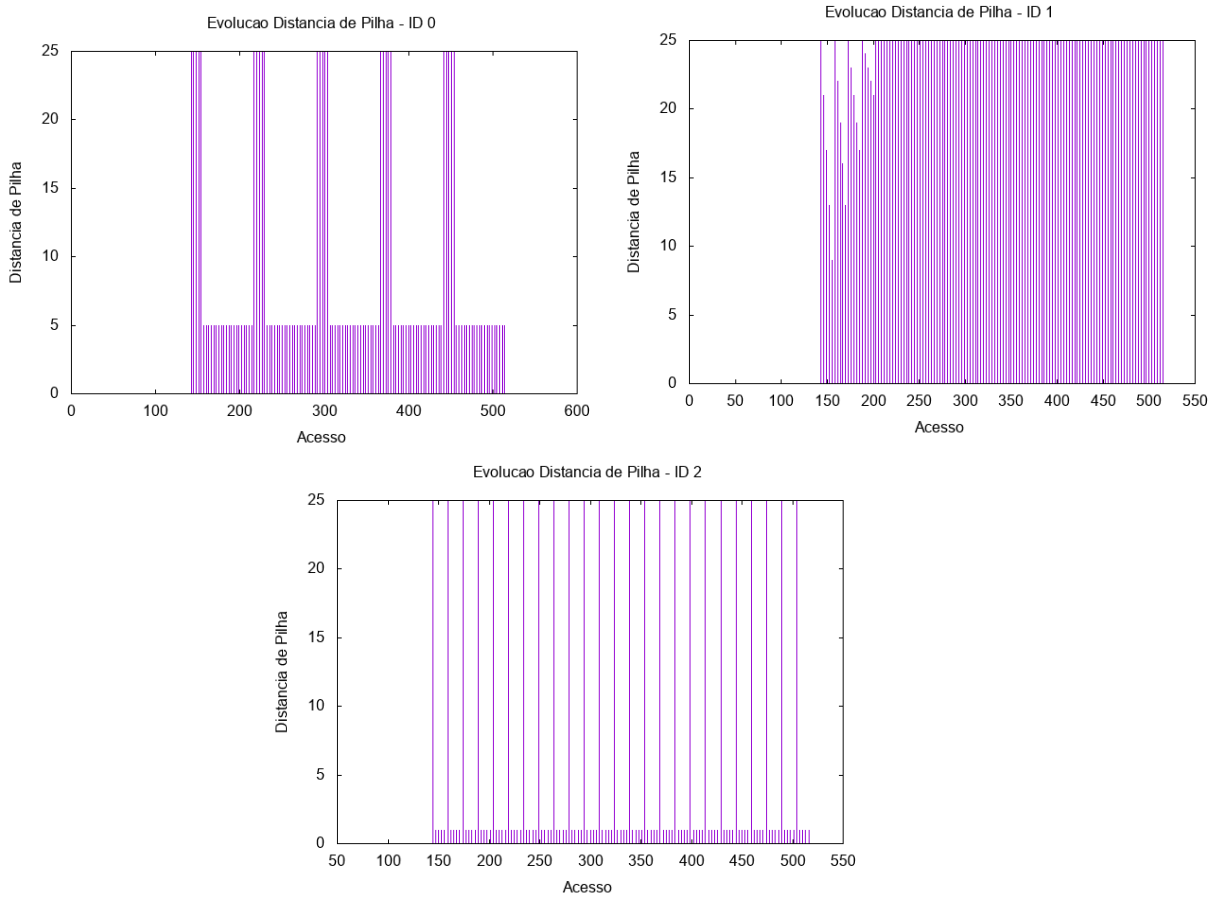




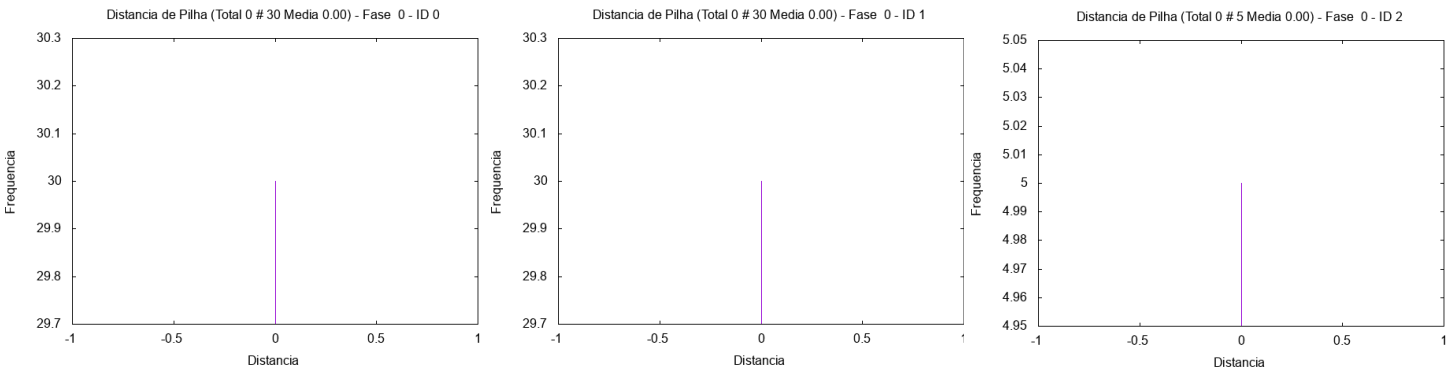


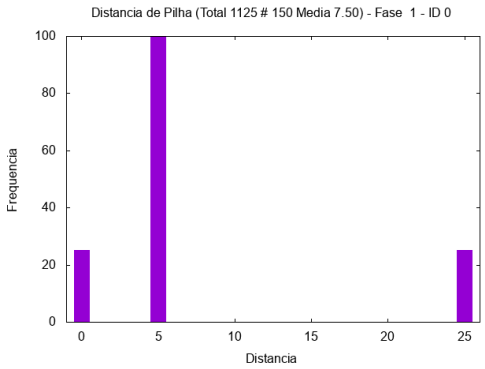
Pode-se observar, que a primeira linha verde de cada gráfico representa, no caso dos Ids 1 e 2 a leitura do arquivo passado como argumento, enquanto no caso do Id 2 é feita apenas a alocação da matriz c para que essa receba o resultado da operação. Em seguida, todas as posições de cada matriz são acessadas, esse acesso é

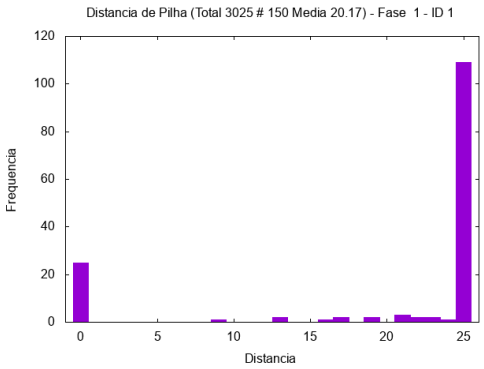
representado pela primeira linha rosa. Na mesma fase, é feita a multiplicação dos elementos das matrizes de Id 0 e 1 e a atribuição às referentes posições na matriz de id 2, vê-se que em cada gráfico o acesso é feito de forma diferente, isso se dá, pois o acesso da matriz 0 é feito por linha, enquanto o da matriz 1 é feito por coluna, cada bloco no gráfico da matriz 0 representa uma linha, e como, não é possível um acesso sequencial às diferentes linhas da matriz o gráfico da matriz 1 é mais espaçado. Na matriz 2, cada bloco representa uma linha, semelhante ao gráfico da matriz 0, visto que a forma de acesso é a mesma e é feita apenas a atribuição aos elementos. Por fim a matriz 2 é acessada e todas são deletadas.

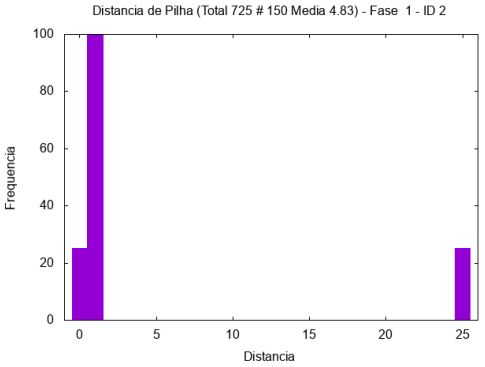


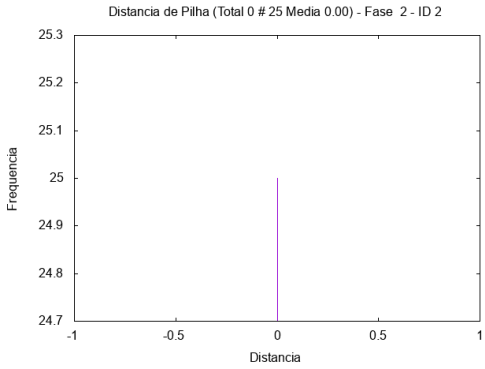
É possível observar a relação entre a evolução da distância de pilha e os gráficos apresentados anteriormente, vê-se como a distância aumenta nos momentos de acesso para as operações e atribuições, e como a distância de pilha é afetada conforme é feito o acesso à matriz 1.



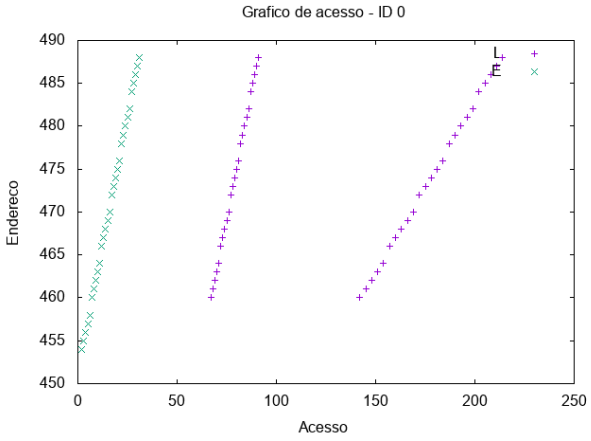


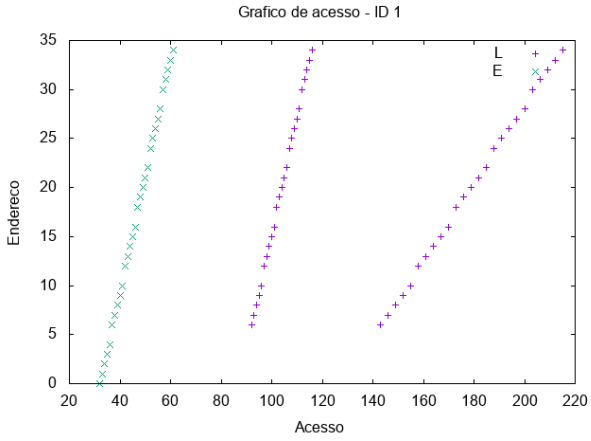


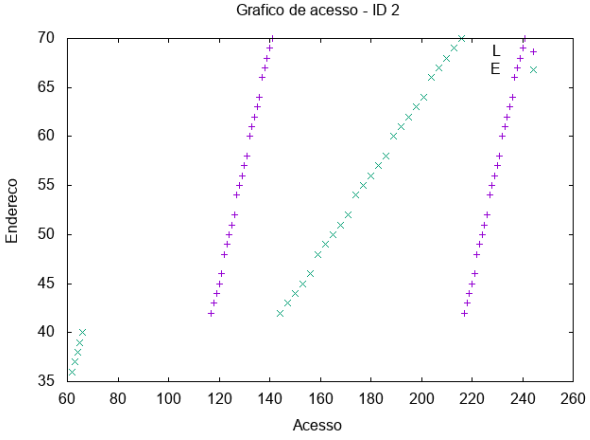




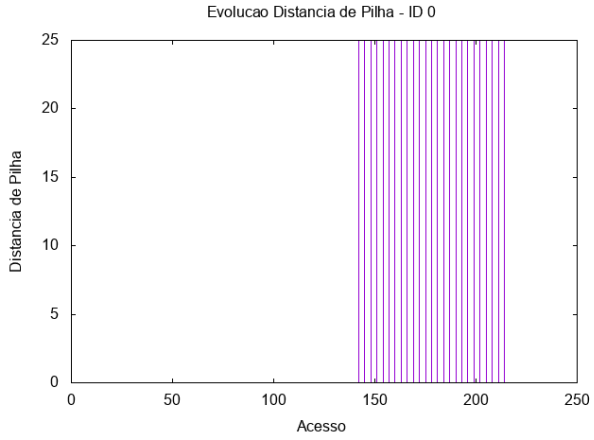
somaMatrizes() :

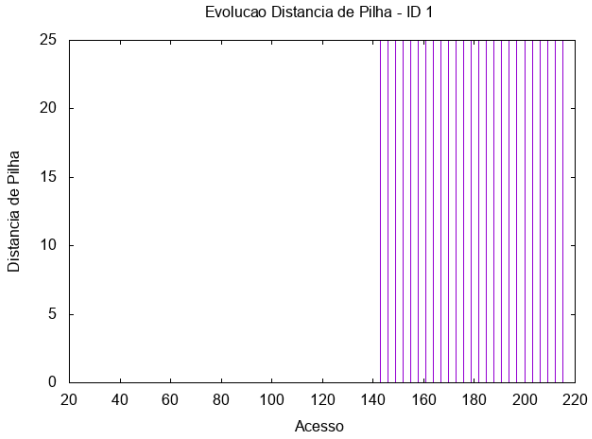


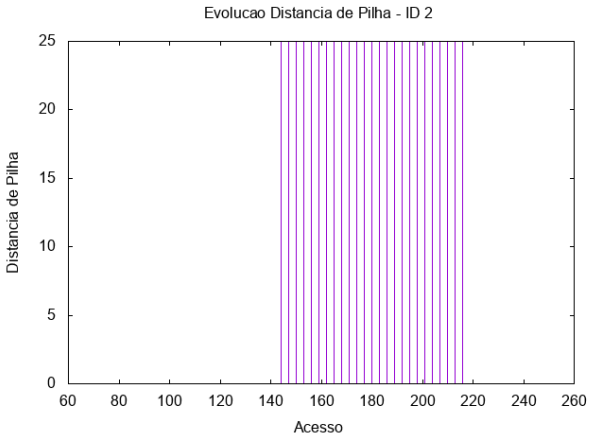




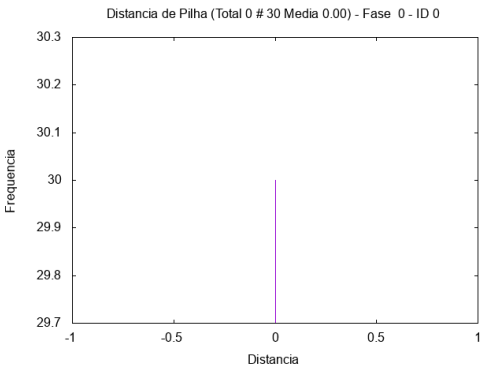
Assim como na função analisada anteriormente, a primeira linha verde representa a leitura e a criação das matrizes, a primeira linha rosa o acesso a cada posição e em seguida é feita a operação, nesse caso o acesso em ambas as matrizes é similar, e é um acesso mais linear assim como o feito na primeira linha rosa. E assim como na operação anterior a atribuição é representada pela linha diagonal no mapa de id 2, e é feito um último acesso à matriz 2 antes de todas serem deletadas.

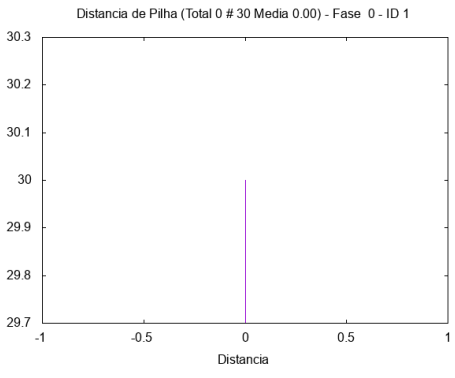


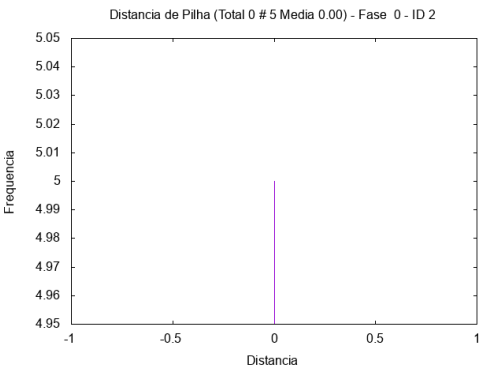


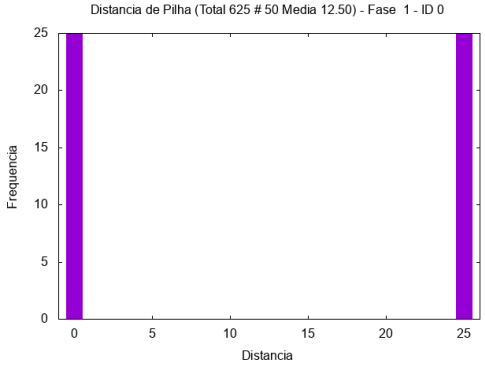


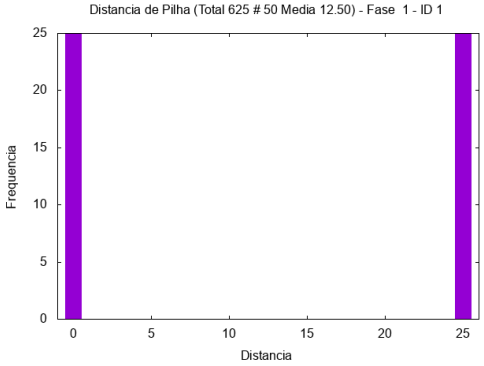
É possível observar a relação entre a evolução da distância de pilha e os gráficos apresentados anteriormente, vê-se como a distância aumenta nos momentos de acesso para as operações e atribuições, e como a distância de pilha é mais homogênea em comparação com a operação multiplicaMatrizes(), visto que na soma o acesso é feito de forma igual em todas as matrizes.

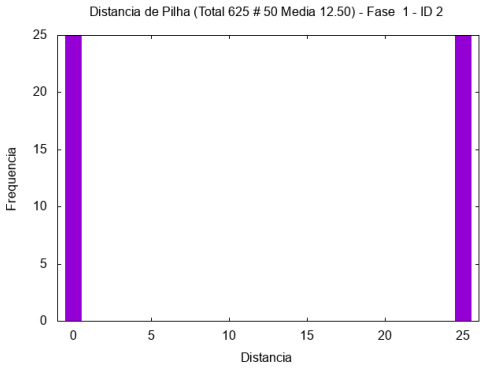


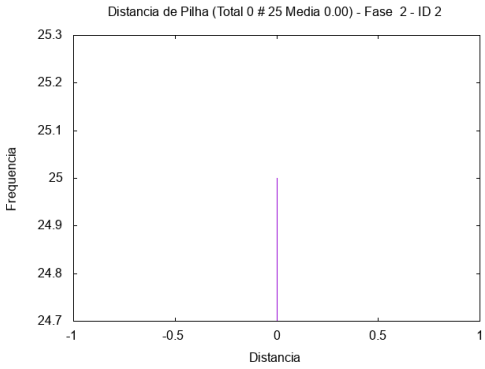




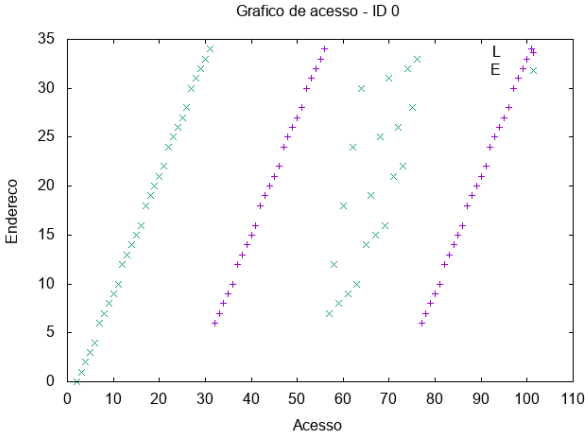




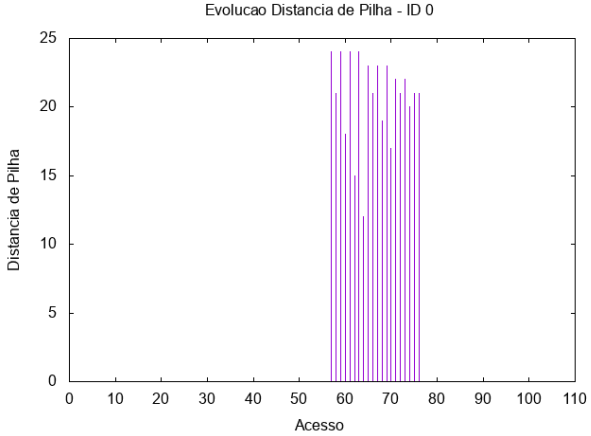




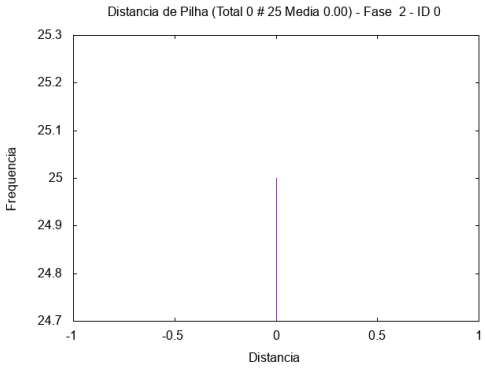
transpoeMatriz() :

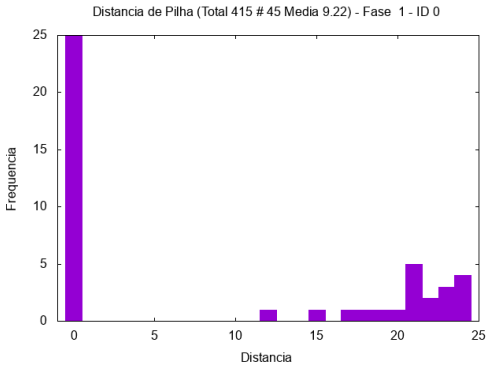


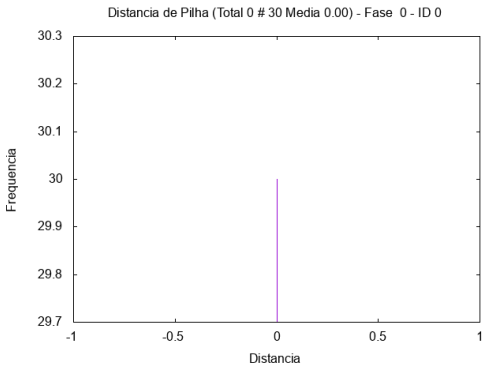
Da mesma forma que nas operações anteriores, a primeira linha verde representa a leitura e criação da matriz, a primeira linha roxa, um primeiro acesso, e a segunda um último acesso antes da desalocação de memória. O pontos dispersos entre as linhas de acesso, representam a operação, o acesso é feito por coluna por coluna em cada linha, mas seus valores são trocados, com um elemento de outra posição, por isso um mapa de acesso mais bagunçado.



É possível observar a relação entre a evolução da distância de pilha e os gráficos apresentados anteriormente, vê-se como a distância aumenta nos momentos de acesso para as operações e atribuições.







Assim como foi visto no gráfico na primeira e última fase, é feito apenas o acesso, enquanto na fase intermediária, há um acesso um pouco mais caótico, visto a maneira como os elementos são acessados para realizar a transposição.

7. Conclusões

O trabalho consistiu em analisar o desempenho em relação ao tempo de execução e o acesso à memória conforme o tamanho da entrada cresce, nas operações básicas envolvendo matrizes, soma, multiplicação e transposição. Por se tratar de uma tarefa relativamente simples, não houve grandes dificuldades para a realização do tal, sendo a tarefa de análise a mais laboriosa.

Pôde-se observar como as funções seguem a análise assintótica feita na seção 2, e como o acesso a memória e a distância de pilha foram coerentes com a implementação feita para as operações e às definições matemáticas dessas.

8. Bibliografia

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. 3a edição. Elsevier, 2012. ISBN 9788535236996

Nivio Ziviani. Projeto de Algoritmos com implementações em Pascal e C. 3ª edição. Cengage Learning, 2011. ISBN 9788522110506

Apêndice

Instruções para compilação e execução

Para facilitar a compilação e execução do programa matop.c, foi disponibilizado um arquivo Makefile com comandos mais simples.

Para compilar o programa, basta possuir o arquivo Makefile no mesmo diretório que os outros programas, estruturados de forma:

|- src
|- bin
|- obj
|- include
Makefile
Onde src deve possuir os arquivos .c e include deve possuir os arquivos .h

e executar o comando "*make comp*".

Para executar o programa basta executar o comando na linha de comando no seguinte formato:

(EXE) (OPERAÇÃO) (COMANDO OPICIONAL 1) (SAÍDA DO COMANDO OPCIONAL) (COMANDO OPICIONAL 2) -1 (ARQUIVO DE ENTRADA DA PRIMEIRA MATRIZ) -2 (ARQUIVO DE ENTRADA DA SEGUNDA MATRIZ) -o (ARQUIVODE DE SAÍDA COM OS RESULTADOS DA OPERAÇÃO)

EXE = matop.exe
OPERAÇÃO = -s, -m ou -t
COMANDO OPICIONAL 1 = -p, para a geração de arquivo .out com dados de desempenho computacional
COMANDO OPICIONAL 2 = -l, para a geração de mapas de acesso a memória
ARQUIVOS DE ENTREDA E SAÍDA - todos em .txt

Para realizar as experimentações vistas na documentação basta executar o comando "*make mem*", para gerar os mapas de acesso e histogramas, o comando "*make perf*", para gerar arquivos .out para a análise de desempenho, e o comando "*make gprof*", para gerar arquivos .gprof para a análise de desempenho.

Caso queira executar todos de uma vez basta dar o comando "*make*" ou "*make all*"

Caso queira mudar os arquivos de entrada e saída e criar seus próprias matrizes basta alterar os arquivos após -1, -2 e -o

Por fim, o comando “*make clean”*, pode ser utilizado para apagar os arquivos .o e gmon.out gerados