Trabalho Prático 0
Operações com matrizes alocadas dinamicamente

Carlos Leão Henrique Brito Malta Leão

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

chbmleao@ufmg.br

1. Introdução

Esta documentação lida com o problema proposto de implementação de um Tipo Abstrato de Dados (TAD) de matrizes, em que suas duas dimensões são alocadas dinamicamente. Além disso, esse TAD deve realizar três operações sobre matrizes: soma, multiplicação e transposição, desenvolvendo a abstração, desempenho e robustez do programa. Esta documentação tem como objetivo explicitar como foi realizada a implementação desse TAD, além de realizar uma análise explicativa de cada uma das três operações sobre matrizes de forma individual.

Para resolver o problema citado, foi criado um programa na linguagem C, que utiliza uma estrutura chamada *matriz* que implementar o TAD e operações solicitadas, que aloca os elementos da matriz de forma dinâmica para evitar gastos de memória desnecessários. Por fim, ao decorrer dessa documentação, alguns aspectos sobre o trabalho serão melhor explicados, como a implementação do código na seção 2 e as instruções de compilação e execução na seção 3. Já na seção 4, será feita uma análise de complexidade mais detalhada sobre as três operações principais do programa.

2. Implementação

- **Organização**

  Em relação à organização do código, foi dividido seguindo as instruções do trabalho prático, em que a pasta TP é a raiz do projeto, a pasta ‘bin’ está vazia, src armazena arquivos de código na linguagem c (‘*.c’), a pasta ‘include’ contém os cabeçalhos do projeto (‘*.h’) e, por fim, a pasta ‘obj’ está vazia. Além disso, o Makefile gera os códigos objeto (‘*.o’) no diretório ‘obj’ e o executável do TP no diretório ‘bin’.

  Ademais, para melhor organização do projeto, existem duas pastas, ‘assets’, que contém algumas matrizes de entrada em arquivos de texto (‘*.txt’), e a pasta ‘out’, que inicialmente fica vazia, mas com a execução do Makefile, recebe as matrizes de saída, também em arquivos de texto.

  O projeto apresenta três arquivos principais, primeiramente o cabeçalho (/include/mat.h), que contém a implementação da estrutura *‘mat_tipo’*, que contém os atributos e funções do TAD matriz. Em segundo lugar, temos o arquivo

(/src/mat.c), que implementa todas as funções do *'mat_tipo'.* Por fim, temos o arquivo (/src/matop.c), que é o código Main do programa, responsável por interpretar os comandos do Makefile e do usuário e executar uma das três operações de matrizes, soma, multiplicação ou transposição.

- **Estruturas de dados e funcionamento do programa**

A principal estrutura de dados utilizada foi o Tipo Abstrato de Dados, *matriz_tipo,* implementado utilizando *structs*, registros em C, quem implementa os atributos de uma matriz, como as suas dimensões x e y, número de identificação e seus elementos.

Nesse registro também são implementadas todas as funções, sendo as principais *somaMatrizes*, *multiplicaMatrizes* e *transpoeMatriz*. Além dessas três funções, também são implementadas algumas funções auxiliares, como *criaMatrizTxt* (lê um arquivo txt que apresenta as informações de uma matriz), *inicializaMatrizNula* (inicializa todos os elementos de uma matriz como zero), *acessaMatriz* (acessa a matriz para fins de registro de acesso), *escreveArquivoTxt* (escreve a matriz em um arquivo txt) e, por fim, *destroiMatriz* (destroi a matriz, que se torna inacessível).

O programa principal, de forma sucinta, baseia-se na função main do arquivo 'src/matop.c', que utiliza a função *parse_args* para receber os valores passados pelo usuário para a execução do programa e atribuição de variáveis. Dessa forma, uma das três seguintes opções terá sido escolhida pelo usuário:

- **OPSOMA:** Lê dois arquivos txt que contém as matrizes A e B. Assim, as matrizes a e b são somadas e formam a matriz C, que é escrita também em um arquivo txt. Por fim, todas as matrizes são destruídas.
- **OPMULTIPLICA:** Lê dois arquivos txt para a formação das matrizes A e B. Assim, as matrizes A e B são multiplicadas e formam a matriz C, que é escrita também em um arquivo txt. Por fim, todas as matrizes são destruídas.
- **OPTRANSPOR:** Lê um arquivo txt para a formação da matriz A. Dessa forma, a matriz A é transposta, escrita em um arquivo txt, e, por fim, destruída.

- **Especificações técnicas**

- Sistema operacional: WSL Ubuntu 20.04
- Linguagem de programação implementada: C
- Compilador: gcc (Ubuntu 9.3.0-17ubuntu1)
- Processador: Intel Core i5-9400F CPU @ 2.90GHz 2.90 GHz
- Memória RAM: 02 X Win Memory 8GB, DDR4, 2666MH

3. Análise de complexidade

Esta seção apresenta a análise de complexidade de tempo para as três principais operações sobre matrizes do programa, soma, multiplicação e transposição, como foi supracitado na seção 2. Para um melhor entendimento, faremos a análise de cada um dos três procedimentos de forma individual.

Para realizar as análises, utilizaremos a variável n para representar as dimensões da matriz, ou seja, n é igual a i ou j, mesmo que essas variáveis não sejam necessariamente iguais. A análise será feita dessa forma para simplificar a análise. Além disso, muitas funções são utilizadas em mais de um procedimento, dessa forma, para a leitura não se tornar repetitiva, cada função será analisada apenas uma vez.

SOMA

Durante o procedimento de soma, existem 3 fases:

Durante a **primeira fase**, as seguintes funções são chamadas: *criaMatrizTxt* (2 vezes, para inicializar a matriz A e B), *criaMatriz* (1 vez para inicializar a matriz C) e *inicializaMatrizNula* (1 vez para inicializar C como uma matriz nula).

Durante a **segunda fase**, as seguintes funções são chamadas: *acessaMatriz* (3 vezes para fins de registro das três matrizes) e *somaMatrizes* (1 vez, realiza a soma da matriz A e B).

Por fim, na **terceira fase**, as seguintes funções são chamadas: *acessaMatriz*(1 vez, para acessar a matriz C), *escreveArquivoTxt* (1 vez, escreve a matriz C) e, por último, *destroiMatriz* (3 vezes, destrói as três matrizes).

- **Complexidade de tempo:**

criaMatriz: Inicializa todos os atributos da struct *mat_tipo*, $\boldsymbol{O(1)}$. Além disso aloca todos os elementos da matriz de dimensão $\boldsymbol{i}$ x $\boldsymbol{j}$ passando por dois laços aninhados, $\boldsymbol{O(n^2)}$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{O(1) + O(n^2) = O(max(1, n^2)) = O(n^2)}$$

criaMatrizTxt: Essa função chama a função *criaMatriz,* $\boldsymbol{O(n^2)}$, além disso, percorre todo o arquivo txt da matriz e atribui cada elemento para a matriz da struct, passando também por dois laços aninhados $\boldsymbol{O(n^2)}$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{O(n^2) + O(n^2) = O(n^2)}$$

***inicializaMatrizNula*:** Inicializa todos os atributos da struct *mat_tipo* como nulos, passando por dois laços aninhados $\boldsymbol{O(n^2)}$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{O(n^2)}$$

acessaMatriz: Acessa todos os elementos da matriz i x j, passando por dois laços aninhados, $O(n^2)$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{O(n^2)}$$

somaMatrizes: Chama as funções *criaMatriz* e *inicializaMatrizNula,* $O(n^2) + O(n^2)$. Atribui todos os elementos da matriz C, passando por dois laços aninhados, $O(n^2)$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{\left(O(n^2) + O(n^2)\right) + O(n^2) = O(n^2)}$$

escreveArquivoTxt: Lê todos os elementos da matriz C, passando por dois laços, $\boldsymbol{O(n^2)}$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{O(n^2)}$$

destroiMatriz*:* Passa por apenas um laço, por todas as linhas da matriz, $\boldsymbol{O(n)}$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{O(n)}$$

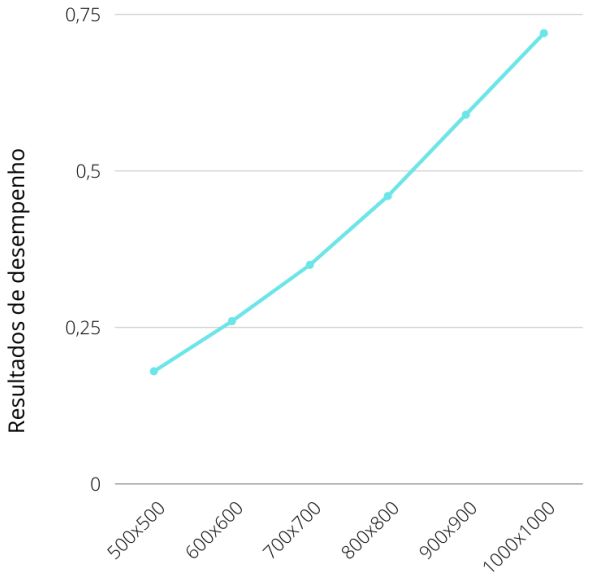
Por fim, podemos realizar a análise completa do procedimento de soma, somando todas a complexidades assintóticas de cada função multiplicado pelas vezes que são usadas.

$$2O(n^2) + 1O(n^2) + 1O(n^2) + 4O(n^2) + 1O(n^2) + 1O(n^2) + 1O(n)$$

$$= \mathrm{O}(\max(n^2, n))$$

$$= \boldsymbol{O(n^2)}$$

Assim, concluímos que a complexidade assintótica do procedimento de soma de matrizes é $O(n^2)$. Isso pode ser comprovado ao observar o gráfico de resultados de desempenho de cada matriz utilizada, de dimensões de 500x500 até 1000x1000, em que o gráfico é visivelmente exponencial.



MULTIPLICAÇÃO

Durante o procedimento de multiplicação, também existem 3 fases:

A **primeira fase** é muito semelhante a primeira fase da soma, em que se utiliza a função *criaMatrizTxt*, *criaMatriz* e *incializaMatrizNula*.

Já a **segunda fase**, também chama a função *acessaMatriz*, para as três matrizes e *multiplicaMatrizes*, que realiza a multiplicação de A x B.

Por último, a **terceira fase** também é muito semelhante ao procedimento de soma, em que são chamadas as funções de *acessaMatriz*, *escreveArquivoTxt* e *destroiMatriz*.

- **Complexidade de tempo:**

  O procedimento de multiplicação é muito semelhante ao procedimento de soma de matrizes. Dessa forma, quase todas as funções do procedimento de multiplicação de matrizes já tiveram sua complexidade assintótica determinada, apenas uma função ainda não foi analisada:

  multiplicaMatrizes: Primeiramente, esta função realiza a chamada de outras, *criaMatriz* e *inicializaMatrizNula,* $O(n^2) + O(n^2)$. Em seguida, para realizar a multiplicação das matrizes são necessários três laços aninhados (em A x B, é preciso associar cada linha de A com as colunas de B e realizar a multiplicação, cada linha de A será multiplicada por toda a matriz, n^2 elementos, e todas as linhas de A devem realizar o mesmo processo, n^3), $O(n^3)$. Dessa forma, teremos a complexidade assintótica da função como:

$$\left(\boldsymbol{O(n^2)} + \boldsymbol{O(n^2)}\right) + \boldsymbol{O(n^3)} = \boldsymbol{O\left(max(n^2, n^3)\right)}$$

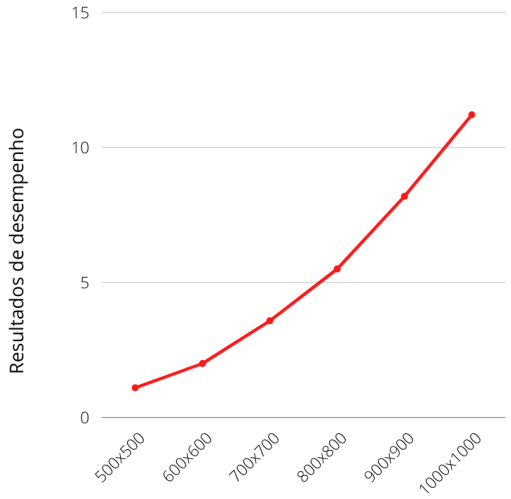
  Por fim, podemos determinar a complexidade assintótica do procedimento de multiplicação de matrizes como um todo, assim como fizemos com o procedimento de soma. Dessa forma, multiplicamos cada complexidade assintótica de cada função pelo número de vezes que é chamada no processo e, por fim, somamos todas as complexidades:

$$2O(n^2) + 1O(n^2) + 1O(n^2) + 4O(n^2) + 1O(n^2) + 1O(n^3) + 1O(n)$$

$$= \mathrm{O}(\max(n^3, n^2, n))$$

$$= \boldsymbol{O(n^3)}$$

  Portanto, concluímos que a complexidade assintótica do procedimento de multiplicação de matrizes é $\boldsymbol{O(n^3)}$, maior que o procedimento de soma. Assim como fizemos no caso anterior, isso também pode ser notado ao analisar os resultados de desempenho dessa função, que apresenta valores temporais maiores e um crescimento mais acelerado que a função de soma.



TRANSPOSIÇÃO

O procedimento de transposição é o mais distinto das três principais operações do programa, já que utiliza apenas uma matriz para realizar todos os processos. Porém, esse processo também apresenta três fases.

A **primeira fase** apenas chama a função *criaMatrizTxt*, criando a matriz A.

Já a **segunda fase**, também chama a função *acessaMatriz* e a função *transpoeMatriz*, passando a matriz A como parâmetro.

Por último, a **terceira fase** é a que mais se assemelha aos procedimentos supracitados, em que é chamada a função *acessaMatriz*, *escreveArquivoTxt* e *destroiMatriz*, todas passando a matriz A como parâmetro.

- **Complexidade de tempo:**

A grande maioria das funções já tiveram sua complexidade assintótica encontrada. Dessa forma, basta encontrarmos a complexidade da última função.

transpoeMatrizes: Primeiramente, esta função realiza a chamada da função criaMatriz, que cria uma matriz auxiliar, $\mathbf{O(n^2)}$. Em seguida, para realizar a transposição da matriz são necessários dois laços aninhados (para copiar todos os elementos da matriz A e passar de forma transposta para a matriz auxiliar), $\boldsymbol{O(n^2)}$. Por último a matriz A recebe o endereço dos atributos da matriz auxiliar, a deixando transposta, $\boldsymbol{O(1)}$. Dessa forma, teremos a complexidade assintótica da função como:

$$\boldsymbol{O(n^2) + O(n^2) + O(1)}$$

$$\boldsymbol{= O(\max(n^2, 1))}$$

$$\boldsymbol{= O(n^2)}$$

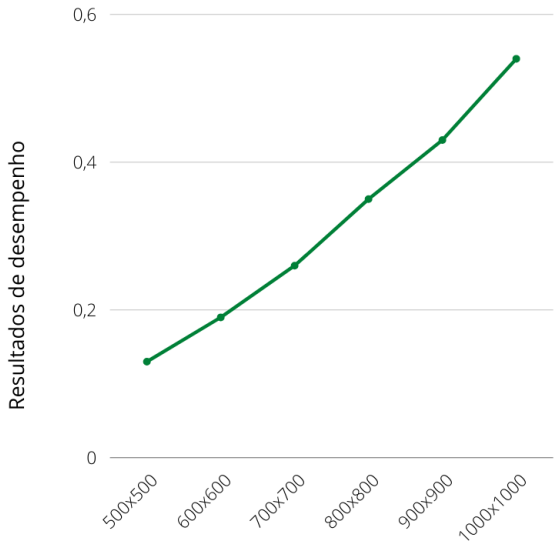
Por fim, podemos determinar a complexidade assintótica do procedimento de transposição de matriz de forma geral. Dessa forma, multiplicamos cada complexidade assintótica de cada função pelo número de vezes que é chamada no processo e, por fim, somamos todas as complexidades:

$$1O(n^2) + 2O(n^2) + 1O(n^2) + 1O(n^2) + 1O(n)$$

$$= O(\max(n^2, n))$$

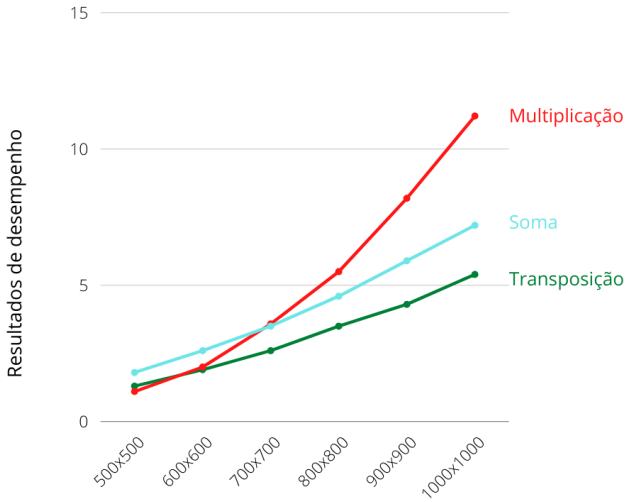
$$= \boldsymbol{O(n^2)}$$

Portanto, concluímos que a complexidade assintótica do procedimento de transposição de matriz é $O(n^2)$, o mesmo que o procedimento de soma e menor que o procedimento de multiplicação. Assim como fizemos nos outros casos, isso também pode ser notado ao analisar os resultados de desempenho dessa função, que apresenta valores menores que as outras duas (devido ao uso de apenas uma matriz e menor número de funções chamadas), mas um crescimento muito semelhante ao crescimento do processo de soma, que apresenta a mesma complexidade de tempo, $O(n^2)$.



CONCLUSÃO – Análise de complexidade

Podemos concluir, portanto, que entre os três procedimentos utilizados no programa, (soma, multiplicação e transposição), o processo que mais demanda tempo de execução é a multiplicação, assim como o que tem a maior complexidade de tempo. Enquanto os outros dois processos apresentam complexidade assintótica igual a $O(n^2)$, a multiplicação apresenta $O(n^3)$. Para deixar isso mais evidente, podemos observar o seguinte gráfico, que compara o crescimento dos resultados das três operações. Porém, para realizar a construção desse gráfico, os resultados de desempenho de soma e transposição foram multiplicados por 10, já que os números de multiplicação são muito maiores.



4. Estratégias de robustez

Robustez é a capacidade do sistema funcionar mesmo em condições anormais. Nesse sentido, todas as funções necessárias apresentam verificação de entradas inadequadas e mal funcionamento do programa. Para tornar o código mais robusto, foram utilizadas as funções da biblioteca *msgassert.h,* disponibilizada no moodle da matéria estrutura de dados. Dessa forma, podemos analisar a implementação de robustez em cada uma das funções:

criaMatriz: verifica se os parâmetros de dimensão passados são maiores que zero, caso contrário, a execução é interrompida e a seguinte mensagem é enviada: “Dimensão nula”. Ademais, também é verificado se a matriz foi devidamente alocada na memória, caso contrário, a seguinte mensagem é enviada: “Não foi possível alocar a matriz”.

criaMatrizTxt: verifica se o arquivo foi possível acessar o arquivo recebido como parâmetro, caso contrário, a execução é interrompida e a seguinte mensagem é enviada: “Não foi possível acessar o arquivo.”. Além disso, também ocorrem as mesmas verificações da função *criaMatriz*.

escreveArquivoTxt: assim como a função *criaMatrizTxt*, também verifica se foi possível acessar o arquivo solicitado, caso contrário, a execução é interrompida e a seguinte mensagem é enviada: “Não foi possível abrir o arquivo.”.

somaMatrizes: verifica se as dimensões da matriz A e da matriz B permitem a realização da operação de soma, para isso, a dimensão x de A deve ser igual a dimensão x de B, e a mesma verificação ocorre para a dimensão y. Caso contrário, a execução é interrompida e a seguinte mensagem é enviada: “Dimensões incompatíveis.”.

multiplicaMatrizes: verifica se as dimensões da matriz A e da matriz B permitem a realização da operação de multiplicação, para isso, a dimensão y de A deve ser igual a dimensão x de B. Caso contrário, a execução é interrompida e a seguinte mensagem é enviada: “Dimensões incompatíveis.”.

5. Testes

6. Análise Experimental

Nessa sessão, realizaremos uma análise dos experimentos realizados em termos de desempenho computacional, eficiência de acesso à memória e análise dos resultados. Neste sentido, avaliaremos o impacto da variação de parâmetros e do tamanho da entrada, além de analisar o padrão de acesso, a localidade de referência e o conjunto de trabalho. Para realizar esse processo analisaremos cada um dos principais procedimentos do programa de forma individual (soma, multiplicação e transposição), utilizando como ferramentas as bibliotecas *gprof*, *memlog* e *analisamem*.

Dessa forma, para realizar a análise de desempenho computacional, utilizaremos doze matrizes de dimensões 500x500 até 1000x1000 (utilizando duas matrizes por chamada de função). Nesse sentido, utilizaremos valores maiores para receber tempos de execução significativos, além de uma maior quantidade de chamada de funções, para analisar como o programa lida com uma grande quantidade de dados (uma matriz 1000x1000 por exemplo, apresenta um milhão de elementos). Já durante a análise de padrão de acesso de memória e localidade de referência, utilizaremos matrizes menores, de dimensões 5x5, para que seja possível analisar minuciosamente a utilização de memória durante as operações, o que seria muito difícil de realizar utilizando matrizes de dimensões grandes.

SOMA

- **Desempenho computacional**

  Os gráficos abaixo representam a saída da chamada *gprof* do Makefile. Dessa forma, ao analisar as tabelas, percebemos que a função principal *somaMatrizes*, não toma tanto tempo de execução assim, por outro lado, a função *acessaMatriz* toma metade do tempo de execução, isso se deve principalmente a ser chamada 4 vezes durante o procedimento de soma.

  Como podemos perceber, a chamada das funções acessaMatriz e as funções da biblioteca memlog apresentam um certo gasto que poderia ser reduzido durante a compilação normal do programa, já que essas funções servem apenas para a documentação de resultados de desempenho e uso de memória.

Perfil raso:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
50.07	0.01	0.01	4	2.50	2.50	acessaMatriz
50.07	0.02	0.01	2	5.01	5.01	inicializaMatrizNula
0.00	0.02	0.00	4	0.00	0.00	criaMatriz
0.00	0.02	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.02	0.00	3	0.00	0.00	destroiMatriz
0.00	0.02	0.00	2	0.00	0.00	criaMatrizTxt
0.00	0.02	0.00	1	0.00	0.00	clkDifMemLog
0.00	0.02	0.00	1	0.00	0.00	desativaMemLog
0.00	0.02	0.00	1	0.00	0.00	escreveArquivoTxt
0.00	0.02	0.00	1	0.00	0.00	finalizaMemLog
0.00	0.02	0.00	1	0.00	0.00	iniciaMemLog
0.00	0.02	0.00	1	0.00	0.00	parse_args
0.00	0.02	0.00	1	0.00	5.01	somaMatrizes

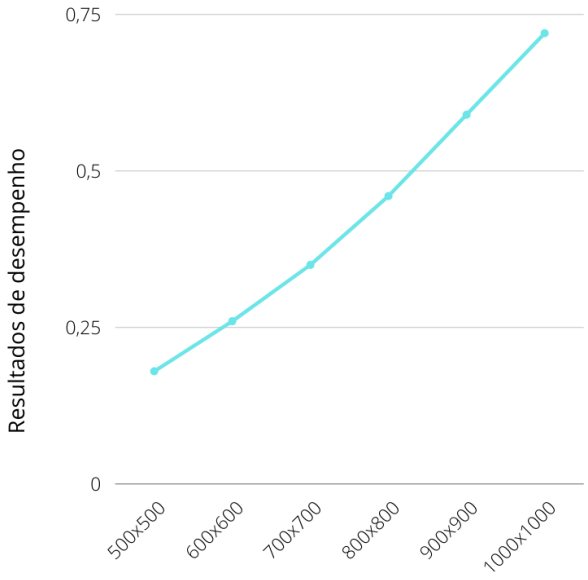
Grafo de execução:

```
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    0.02                 main [1]
                0.01    0.00       4/4           acessaMatriz [2]
                0.01    0.00       1/2           inicializaMatrizNula [3]
                0.00    0.01       1/1           somaMatrizes [4]
                0.00    0.00       3/3           defineFaseMemLog [6]
                0.00    0.00       3/3           destroiMatriz [7]
                0.00    0.00       2/2           criaMatrizTxt [8]
                0.00    0.00       1/1           parse_args [14]
                0.00    0.00       1/1           iniciaMemLog [13]
                0.00    0.00       1/1           desativaMemLog [10]
                0.00    0.00       1/4           criaMatriz [5]
                0.00    0.00       1/1           escreveArquivoTxt [11]
                0.00    0.00       1/1           finalizaMemLog [12]
-----------------------------------------------
                0.01    0.00       4/4           main [1]
[2]     50.0    0.01    0.00       4         acessaMatriz [2]
-----------------------------------------------
                0.01    0.00       1/2           main [1]
                0.01    0.00       1/2           somaMatrizes [4]
[3]     50.0    0.01    0.00       2         inicializaMatrizNula [3]
-----------------------------------------------
                0.00    0.01       1/1           main [1]
[4]     25.0    0.00    0.01       1         somaMatrizes [4]
                0.01    0.00       1/2           inicializaMatrizNula [3]
                0.00    0.00       1/4           criaMatriz [5]
-----------------------------------------------
                0.00    0.00       1/4           main [1]
                0.00    0.00       1/4           somaMatrizes [4]
                0.00    0.00       2/4           criaMatrizTxt [8]
[5]      0.0    0.00    0.00       4         criaMatriz [5]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[6]      0.0    0.00    0.00       3         defineFaseMemLog [6]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[7]      0.0    0.00    0.00       3         destroiMatriz [7]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[8]      0.0    0.00    0.00       2         criaMatrizTxt [8]
                0.00    0.00       2/4           criaMatriz [5]
-----------------------------------------------
                0.00    0.00       1/1           finalizaMemLog [12]
[9]      0.0    0.00    0.00       1         clkDifMemLog [9]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[10]     0.0    0.00    0.00       1         desativaMemLog [10]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[11]     0.0    0.00    0.00       1         escreveArquivoTxt [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         finalizaMemLog [12]
                0.00    0.00       1/1           clkDifMemLog [9]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[13]     0.0    0.00    0.00       1         iniciaMemLog [13]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[14]     0.0    0.00    0.00       1         parse_args [14]
-----------------------------------------------
```

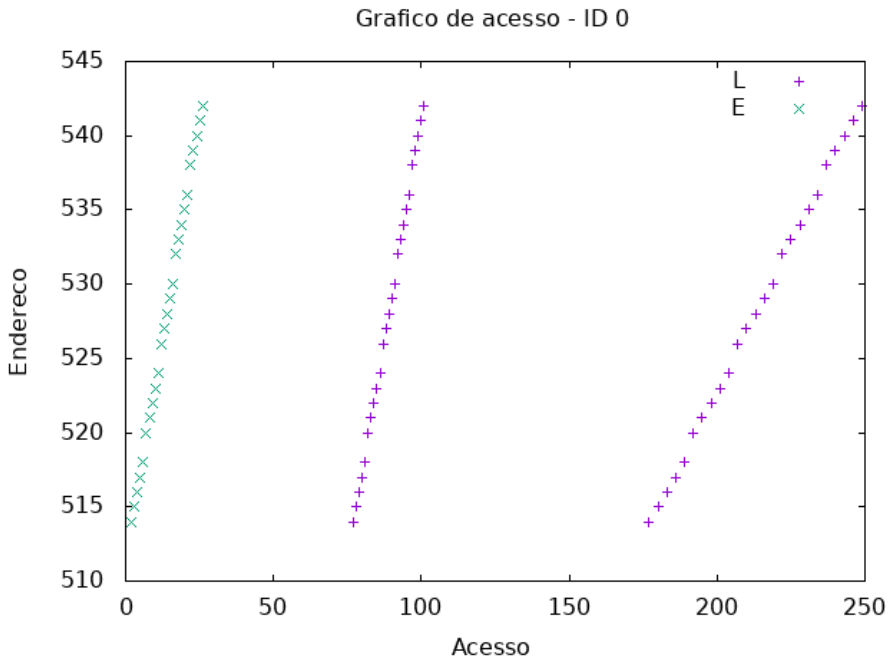
Além disso, podemos também analisar os resultados de desempenho que já foram inicialmente observados na sessão 3. Os resultados de desempenho foram gerados utilizando matrizes de 500x500 até 1000x1000.

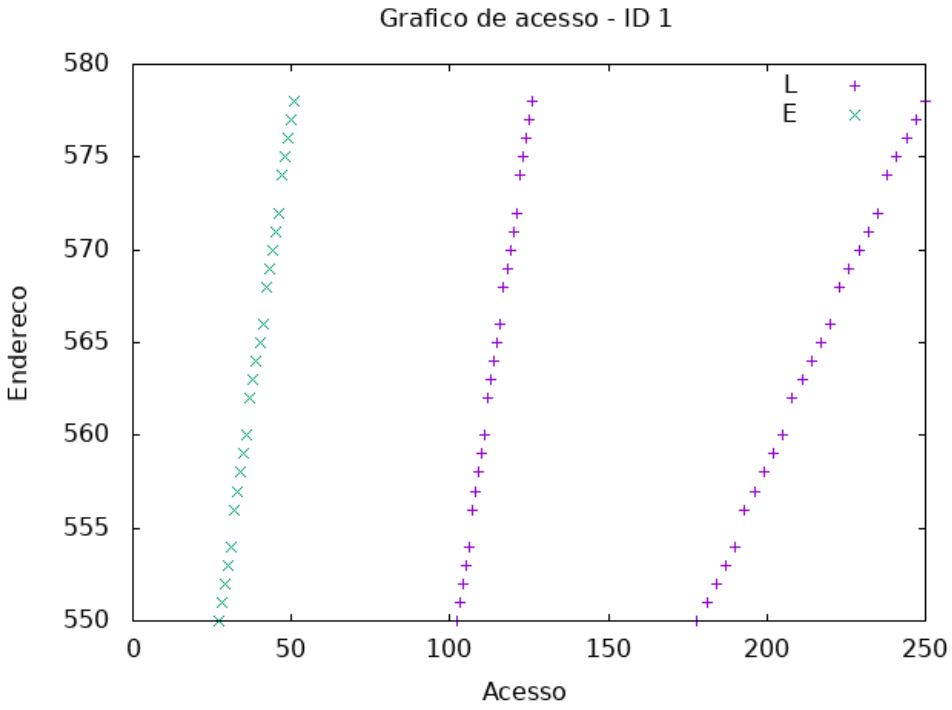
MATRIZES UTILIZADAS	RESULTADO DE DESEMPENHO
500x500	I 1 20789.977822600 F 2 20790.159073500 R 0.181250900
600x600	I 1 20787.509879300 F 2 20787.771113900 R 0.261234600
700x700	I 1 20783.293985200 F 2 20783.653245400 R 0.359260200
800x800	I 1 20776.967936900 F 2 20777.430625400 R 0.462688500
900x900	I 1 20767.745079000 F 2 20768.335778500 R 0.590699500
1000x1000	I 1 20755.257231100 F 2 20755.980181800 R 0.722950700

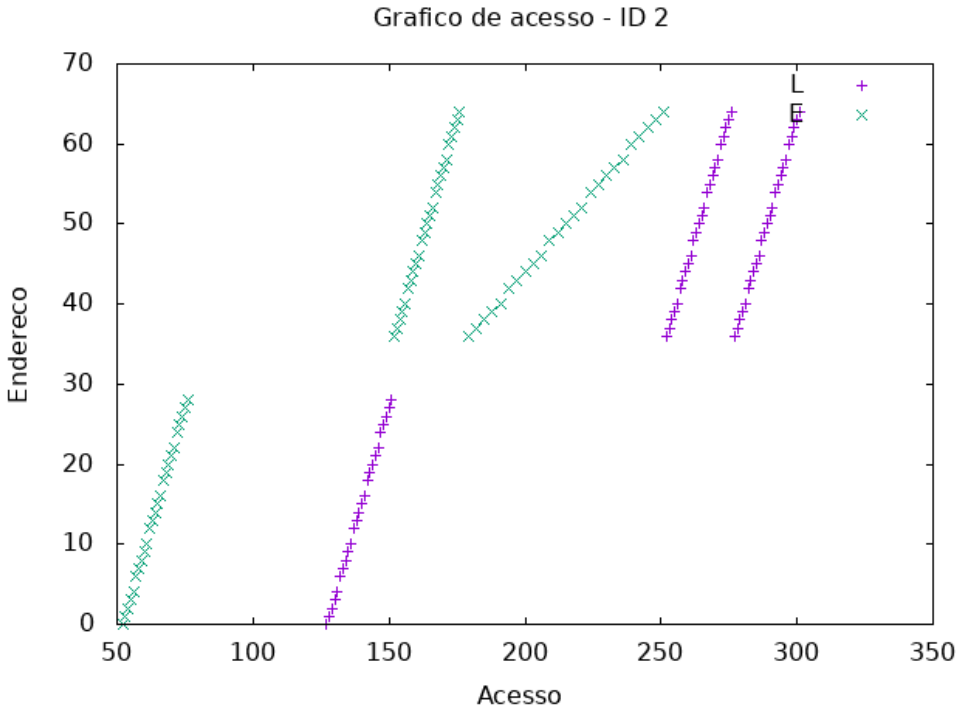
Como podemos perceber, sempre que as dimensões das matrizes utilizadas como parâmetros aumentam, o resultado de desempenho também aumenta, ou seja, leva mais tempo para executar o procedimento. Além disso, pode-se notar que com o aumento das dimensões de 500x500 para 600x600 existe uma diferença de tempo de 0.08, porém, ao final, com o aumento de 900x900 para 1000x1000 essa diferença aumenta para 0.13. Nesse sentido, os resultados de desempenho nos mostram um crescimento exponencial, assim como foi mostrado na análise de tempo na sessão 3. Por fim, podemos criar um gráfico que mostra o crescimento destes resultados de desempenho da operação de soma de matrizes.



- **Análise de Padrão de Acesso à Memória e Localidade de Referência**





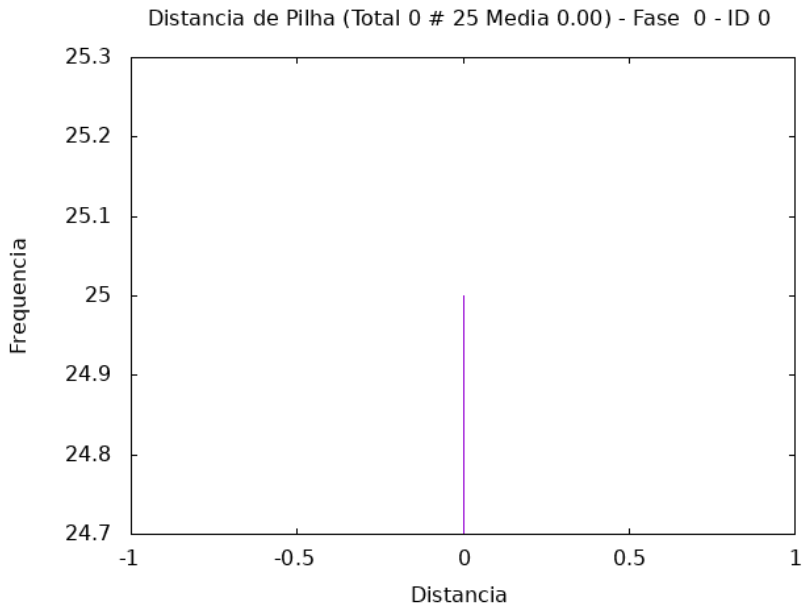


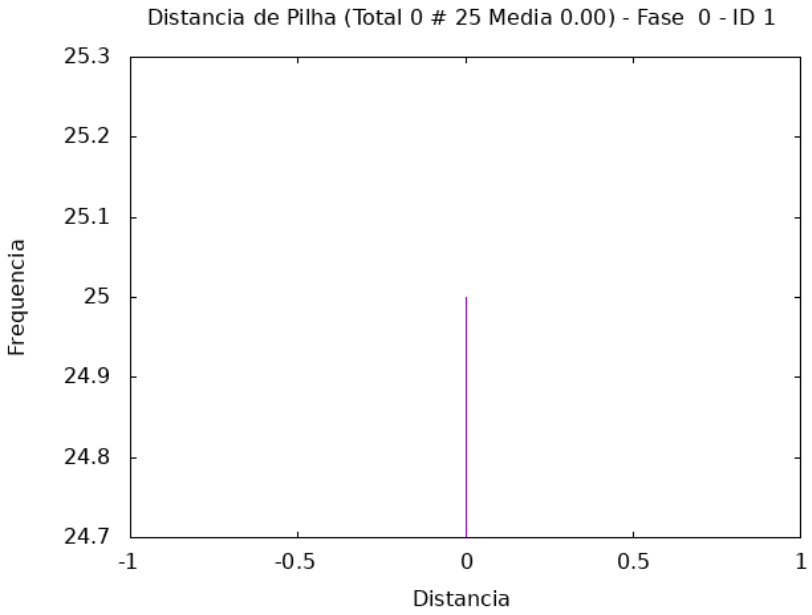
Estamos analisando uma soma de matrizes no formato $\boldsymbol{A} + \boldsymbol{B} = \boldsymbol{C}$, dessa forma, cada um dos três gráficos acima representa uma matriz, ou seja, A = ID0, B = ID1, C = ID2. Os gráficos informam o endereço acessado em determinado acesso, em que os acessos são realizados um após o outro. Além disso, existe uma diferenciação entre um simples acesso, que é representado na cor roxa, e uma atribuição de valor, que é representado na cor verde. Dito isso, podemos realizar a análise dos gráficos.

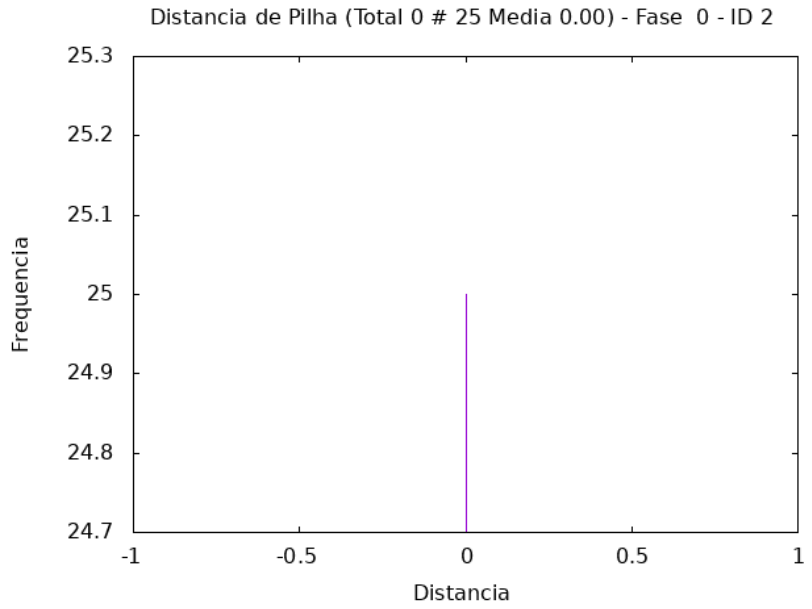
Primeiramente, temos no primeiro e segundo gráfico as matrizes A e B,

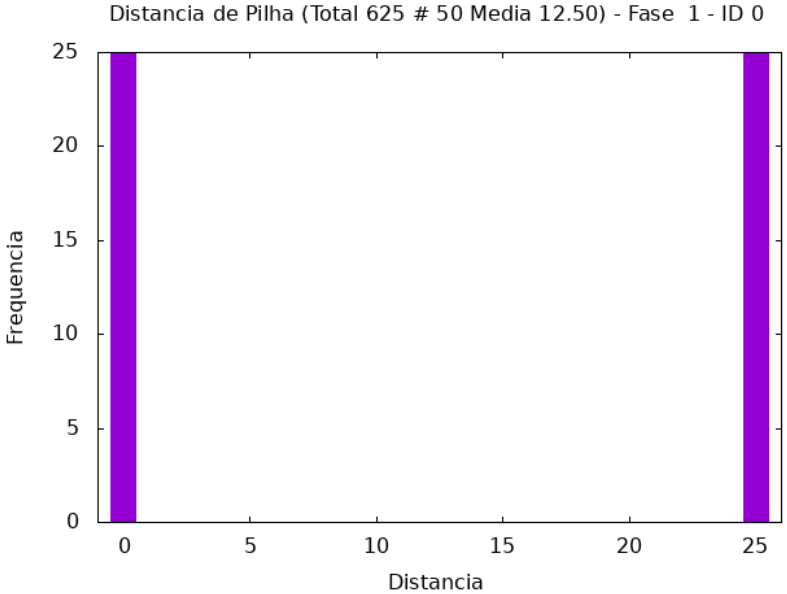
que apresentam um comportamento muito semelhante, como já foi visto na análise de tempo. Primeiro a matriz é preenchida com os valores de um arquivo txt, e em seguida é acessada pela função *acessaMatriz*, o que é representado no gráfico pelas primeiras linhas de escrita e leitura. Depois disso, existe mais uma linha de acesso, em que são acessados os valores das duas matrizes para realizar a operação de soma.

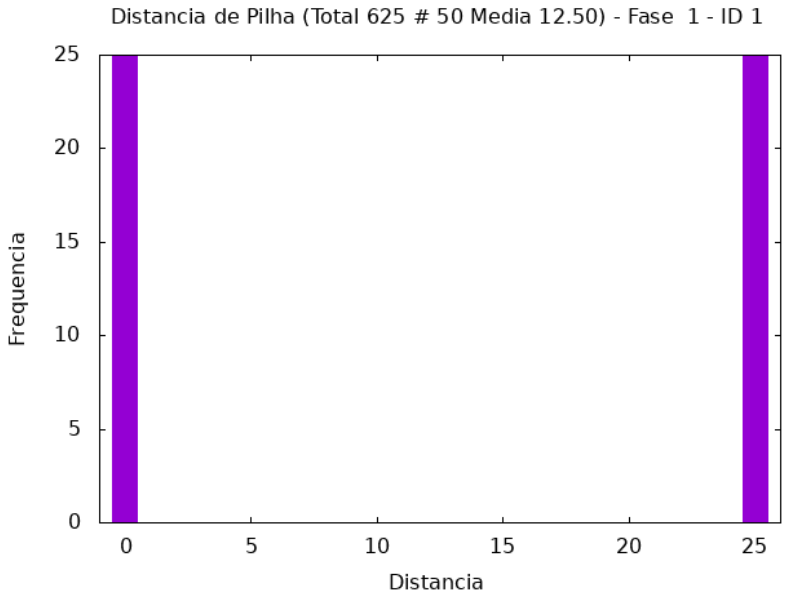
Em segundo lugar, temos a matriz C, que possui um gráfico distinto das matrizes A e B. Primeiramente, a função *inicializaMatrizNula* é chamada e inicializa todos os elementos de C, e logo em seguida a função acessaMatriz também é chamada, gerando as linhas verde e roxa. Em seguida, a função de soma das matrizes é chamada, por questão de segurança a matriz C é inicializada com zeros, e, posteriormente cada elemento da matriz C é atribuído o valor da soma das matrizes A e B, formando a linha verde um pouco mais horizontal que a outra (a linha é um pouco mais horizontal devido ao acesso de um elemento da matriz A, um elemento da matriz B e em seguida a escrita do elemento da matriz C, um processo que se repete até o final da função). Por último, a matriz é acessada novamente duas vezes, uma pela função *acessaMatriz* e outra pela função *escreveArquivoTxt*.

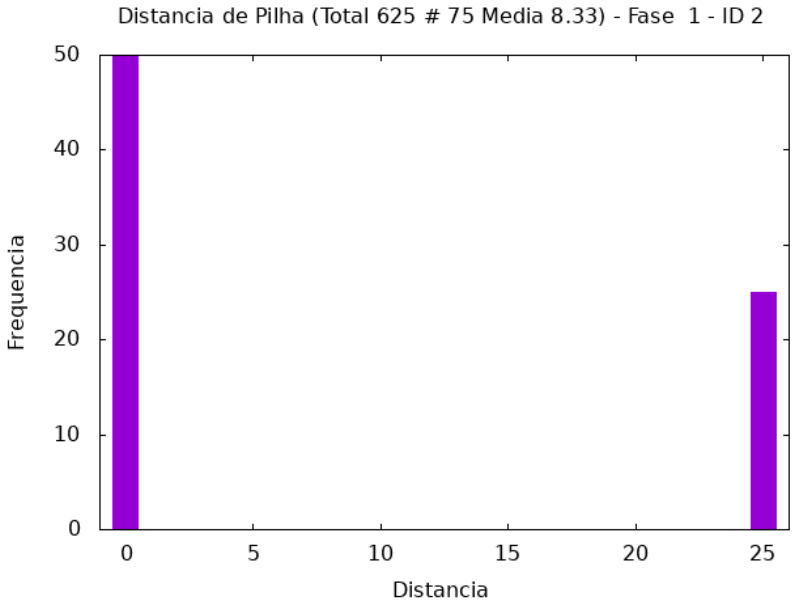


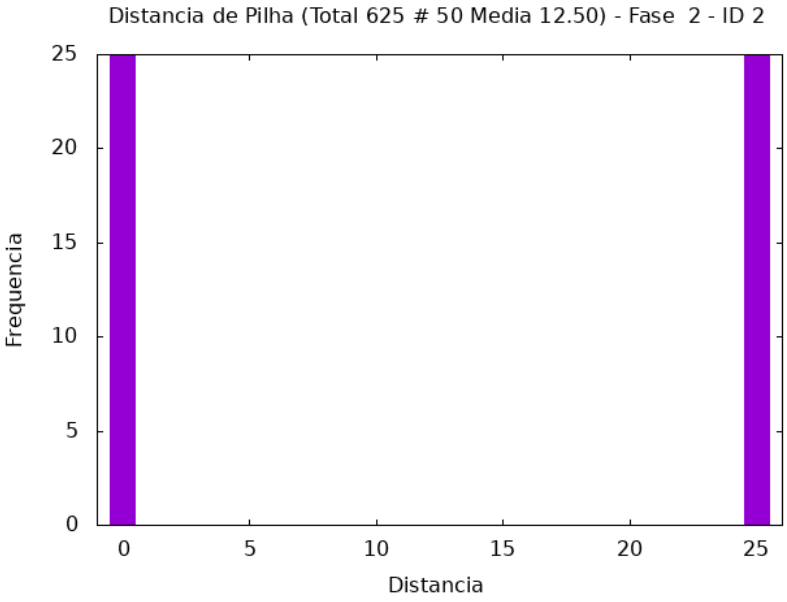

Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 1
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1


Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 2
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1


Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 0
Frequencia
Distancia
25
20
15
10
5
0
0
5
10
15
20
25


Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 1
Frequencia
Distancia
0
5
10
15
20
25


Distancia de Pilha (Total 625 # 75 Media 8.33) - Fase 1 - ID 2
Frequencia
Distancia
0
10
20
30
40
50
5
15
25

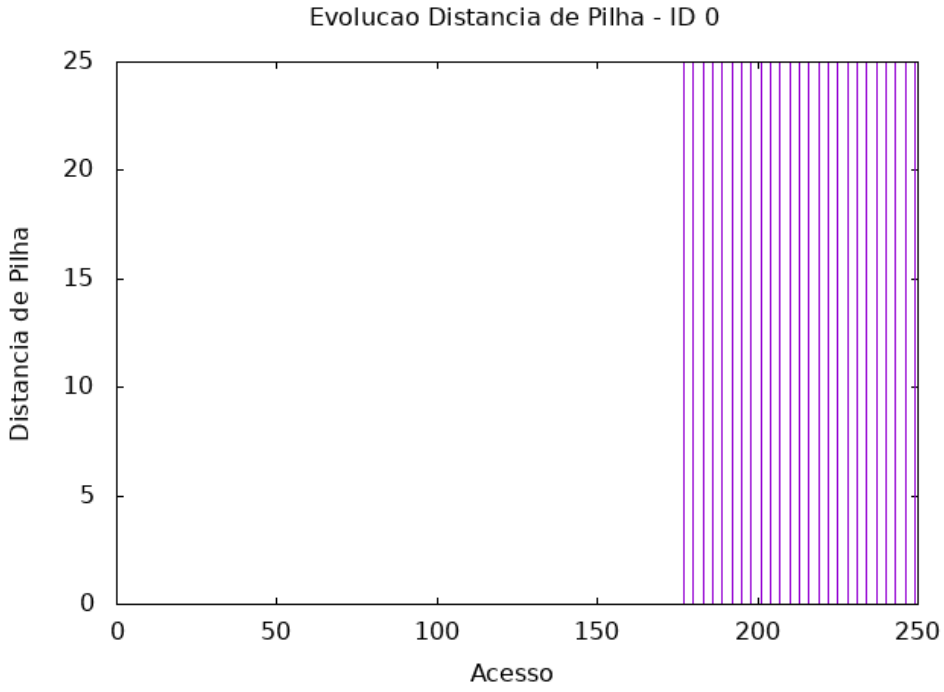

Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 2 - ID 2
Frequencia
Distancia
0
5
10
15
20
25

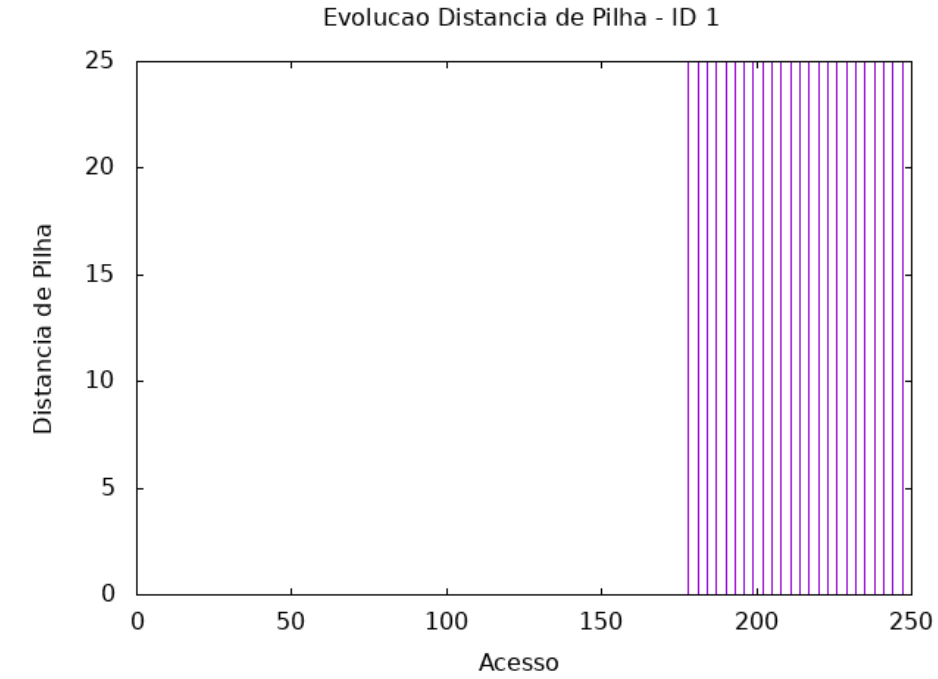
Agora analisaremos os gráficos de distância de pilha, em que além de serem divididos entre as três matrizes, também são divididos entre as três fases do procedimento de soma.

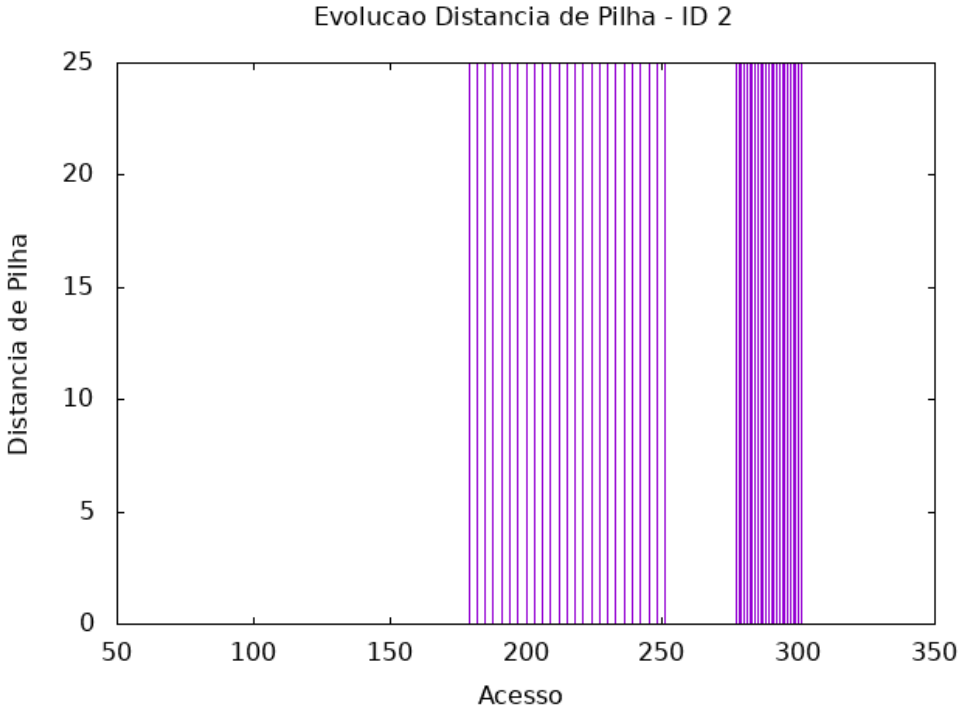
Primeiramente, observando os três gráficos da primeira fase, percebemos que todas as distâncias de pilha das três matrizes são zero. Isso ocorre já que na primeira fase essas matrizes estão sendo inicializadas pelas funções *criaMatrizTxt* e *inicializaMatrizNula*, começando todas pelo primeiro endereço da matriz e atribuindo valores até o final. Como um endereço está sempre ao lado do outro, a distância de pilha é zero.

Durante a segunda fase, os gráficos tornam-se mais distintos. Primeiramente, as matrizes A e B também apresentam um comportamento muito semelhante, já que são acessados pelas mesmas funções. Nesse sentido, primeiramente todos os elementos são acessados pela função acessaMatriz, em seguida, para acessar o primeiro elemento e realizar a soma, é preciso percorrer todas as 25 posições da matriz, o que gera uma distância de pilha igual a 25. Esse processo ocorre durante toda a função *somaMatrizes*. O gráfico da matriz C é um pouco diferente porque a função *inicializaMatrizNula* também é chamada, o que gera mais 25 distâncias de pilha iguais a zero.

Por último, durante a terceira fase, apenas a matriz C é acessada, pela função *acessaMatriz* e *escreveArquivoTxt.* Esse acesso antecipado gera o mesmo efeito da segunda fase, que é preciso percorrer 25 elementos para escrever o arquivo de teto, o que gera uma distância de pilha igual a 25.







Por fim, analisando os gráficos de evolução de distância de pilha, podemos reforçar os aspectos encontrados na análise dos gráficos de distância de pilha. Além disso, ao observar o gráfico da matriz C, notamos que primeiramente, durante a realização da soma, os acessos são um pouco mais espaçados, já que é necessário acessar um elemento da matriz A, um da matriz B, e depois atribuir o elemento na matriz C. Esse espaçamento não ocorre, por exemplo, durante a execução da função *escreveArquivoTxt*.

MULTIPLICAÇÃO

- **Desempenho computacional**

Os gráficos abaixo representam a saída da chamada *gprof* do Makefile. Diferentemente dos resultados encontrados no procedimento de soma, no caso da função principal de multiplicação *multiplicaMatrizes* é a função que mais demanda tempo de execução. Mesmo sendo chamada apenas uma vez, essa função gasta 99.93% do tempo de execução, o que mostra a grande complexidade desse processo.

Esse grande gastos se deve à implementação dessa função de multiplicação, que precisa passar por três laços aninhados, o que gera uma complexidade assintótica $O(n^3)$. Seria necessário pensar um algoritmo mais rápido para contornar esse grande problema de gasto de memória e tempo.

Perfil raso:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.93	10.07	10.07	1	10.07	10.08	multiplicaMatrizes
0.10	10.08	0.01	4	0.00	0.00	acessaMatriz
0.10	10.09	0.01	2	0.01	0.01	inicializaMatrizNula
0.00	10.09	0.00	4	0.00	0.00	criaMatriz
0.00	10.09	0.00	3	0.00	0.00	defineFaseMemLog
0.00	10.09	0.00	3	0.00	0.00	destroiMatriz
0.00	10.09	0.00	2	0.00	0.00	criaMatrizTxt
0.00	10.09	0.00	1	0.00	0.00	clkDifMemLog
0.00	10.09	0.00	1	0.00	0.00	desativaMemLog
0.00	10.09	0.00	1	0.00	0.00	escreveArquivoTxt
0.00	10.09	0.00	1	0.00	0.00	finalizaMemLog
0.00	10.09	0.00	1	0.00	0.00	iniciaMemLog
0.00	10.09	0.00	1	0.00	0.00	parse_args

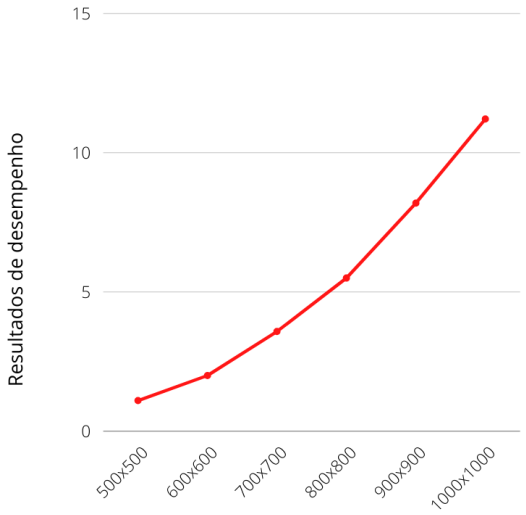
Grafo de execução:

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00   10.09                 main [1]
               10.07    0.01       1/1           multiplicaMatrizes [2]
                0.01    0.00       4/4           acessaMatriz [3]
                0.01    0.00       1/2           inicializaMatrizNula [4]
                0.00    0.00       3/3           defineFaseMemLog [6]
                0.00    0.00       3/3           destroiMatriz [7]
                0.00    0.00       2/2           criaMatrizTxt [8]
                0.00    0.00       1/1           parse_args [14]
                0.00    0.00       1/1           iniciaMemLog [13]
                0.00    0.00       1/1           desativaMemLog [10]
                0.00    0.00       1/4           criaMatriz [5]
                0.00    0.00       1/1           escreveArquivoTxt [11]
                0.00    0.00       1/1           finalizaMemLog [12]
-----------------------------------------------
               10.07    0.01       1/1           main [1]
[2]     99.9   10.07    0.01       1         multiplicaMatrizes [2]
                0.01    0.00       1/2           inicializaMatrizNula [4]
                0.00    0.00       1/4           criaMatriz [5]
-----------------------------------------------
                0.01    0.00       4/4           main [1]
[3]      0.1    0.01    0.00       4         acessaMatriz [3]
-----------------------------------------------
                0.01    0.00       1/2           main [1]
                0.01    0.00       1/2           multiplicaMatrizes [2]
[4]      0.1    0.01    0.00       2         inicializaMatrizNula [4]
-----------------------------------------------
                0.00    0.00       1/4           main [1]
                0.00    0.00       1/4           multiplicaMatrizes [2]
                0.00    0.00       2/4           criaMatrizTxt [8]
[5]      0.0    0.00    0.00       4         criaMatriz [5]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[6]      0.0    0.00    0.00       3         defineFaseMemLog [6]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[7]      0.0    0.00    0.00       3         destroiMatriz [7]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[8]      0.0    0.00    0.00       2         criaMatrizTxt [8]
                0.00    0.00       2/4           criaMatriz [5]
-----------------------------------------------
                0.00    0.00       1/1           finalizaMemLog [12]
[9]      0.0    0.00    0.00       1         clkDifMemLog [9]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[10]     0.0    0.00    0.00       1         desativaMemLog [10]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[11]     0.0    0.00    0.00       1         escreveArquivoTxt [11]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[12]     0.0    0.00    0.00       1         finalizaMemLog [12]
                0.00    0.00       1/1           clkDifMemLog [9]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[13]     0.0    0.00    0.00       1         iniciaMemLog [13]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[14]     0.0    0.00    0.00       1         parse_args [14]
-----------------------------------------------
```

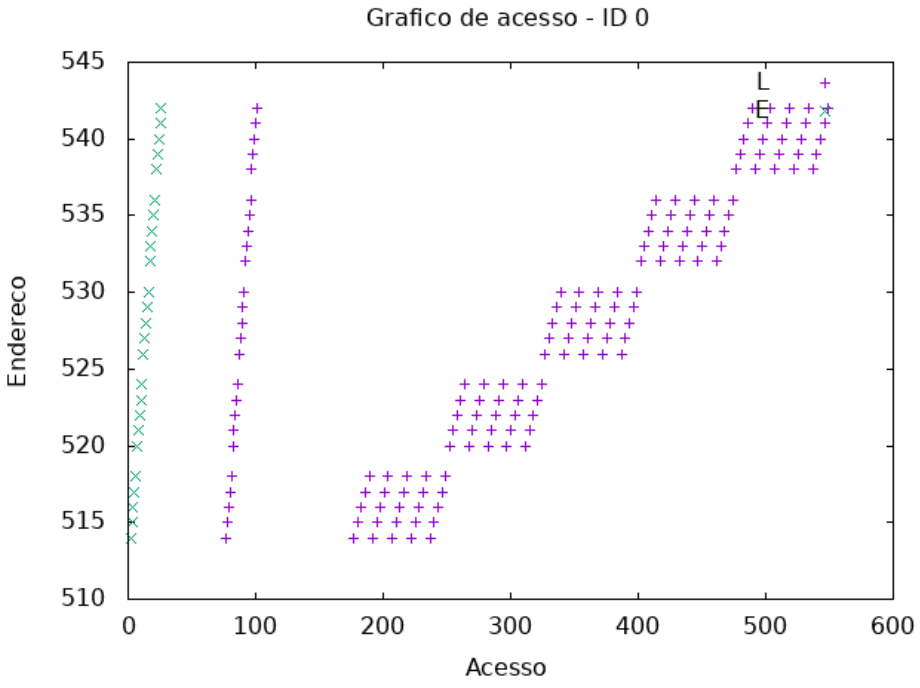
Além disso, podemos também analisar os resultados de desempenho que já foram inicialmente observados na sessão 3. Os resultados de desempenho foram gerados utilizando matrizes de 500x500 até 1000x1000.

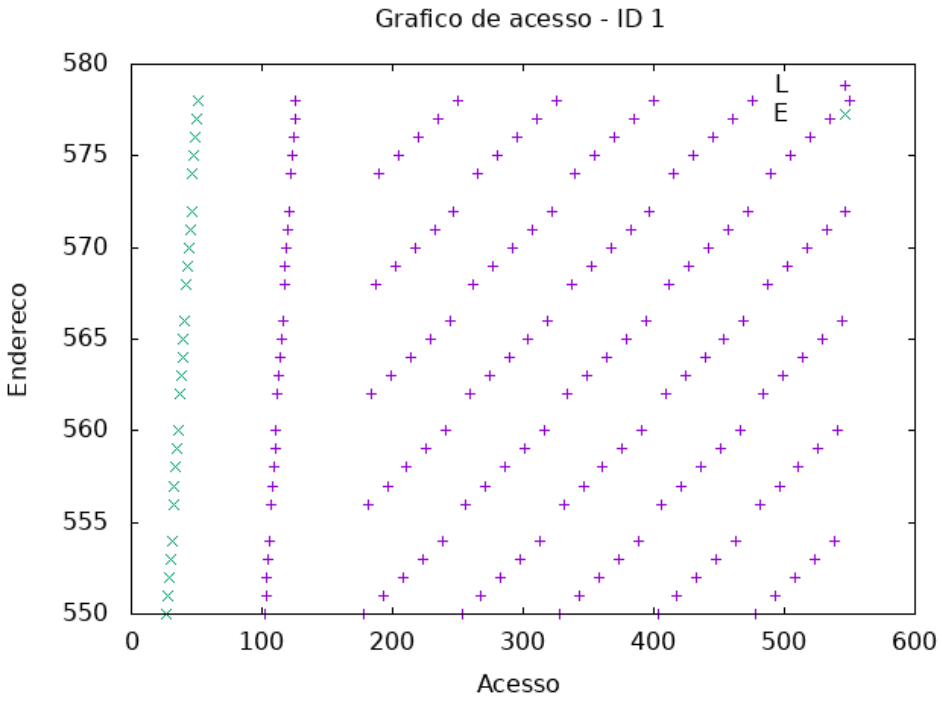
MATRIZES UTILIZADAS	RESULTADO DE DESEMPENHO
500x500	I 1 20790.160021100 F 2 20791.269130300 R 1.109109200
600x600	I 1 20787.772098500 F 2 20789.779107800 R 2.007009300
700x700	I 1 20783.654330700 F 2 20787.242031700 R 3.587701000
800x800	I 1 20777.431724900 F 2 20782.939679800 R 5.507954900
900x900	I 1 20768.337113000 F 2 20776.530002400 R 8.192889400
1000x1000	I 1 20755.981481600 F 2 20767.199015600 R 11.217534000

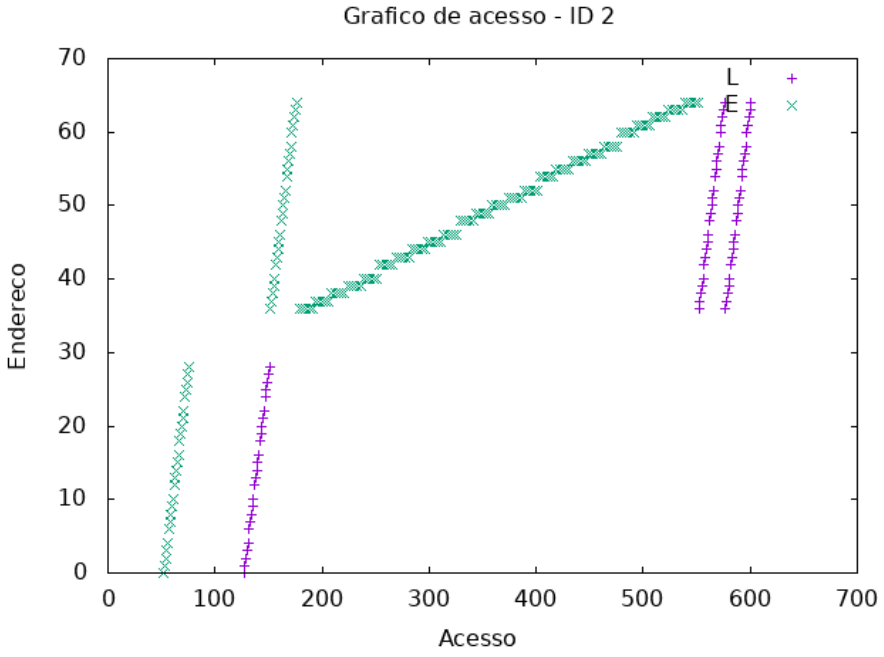
Perceptivelmente, os resultados de desempenho de multiplicação são muito maiores que os de soma. Além disso, sempre que as dimensões das matrizes utilizadas como parâmetros aumentam, o resultado de desempenho também aumenta, ou seja, leva mais tempo para executar o procedimento. Além disso, pode-se notar que com o aumento das dimensões de 500x500 para 600x600 existe uma diferença de tempo de 0.9, porém, ao final, com o aumento de 900x900 para 1000x1000 essa diferença aumenta para 3.02. Nesse sentido, os resultados de desempenho nos mostram um crescimento exponencial maior que o crescimento encontrado no procedimento de soma, assim como foi mostrado na análise de tempo na sessão 3. Por fim, podemos criar um gráfico que mostra o crescimento destes resultados de desempenho da operação de multiplicação de matrizes.



- **Análise de Padrão de Acesso à Memória e Localidade de Referência**





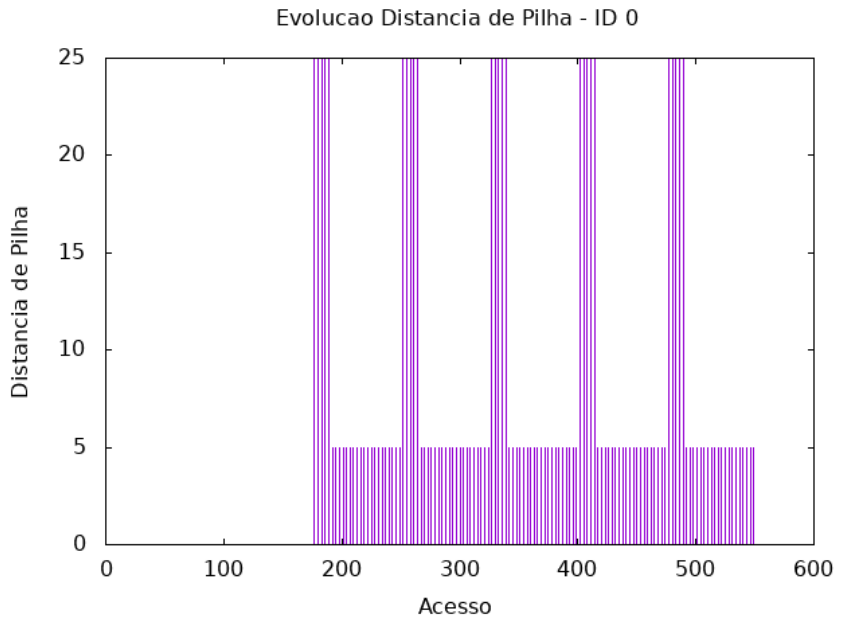


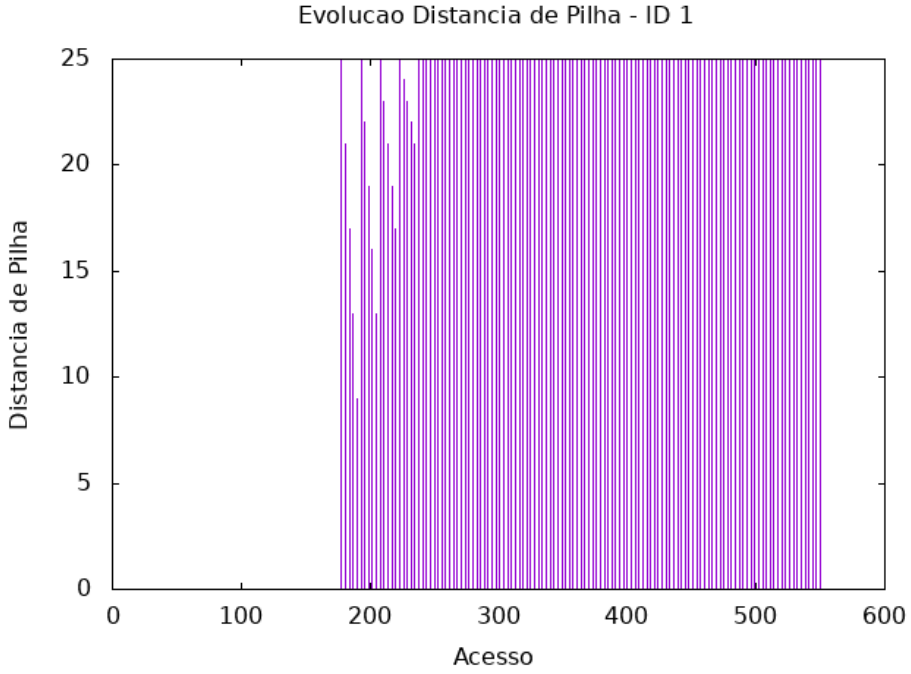
Ao observar os três gráficos de acesso, percebemos que todos apresentam um aspecto em comum, uma função de escrita e outra de leitura. Nesse sentido, todas as matrizes são inicializadas e depois são acessados todos os valores atribuídos. Porém, é perceptível que após essa primeira parte, os acessos de cada uma das três matrizes são muito distintos.

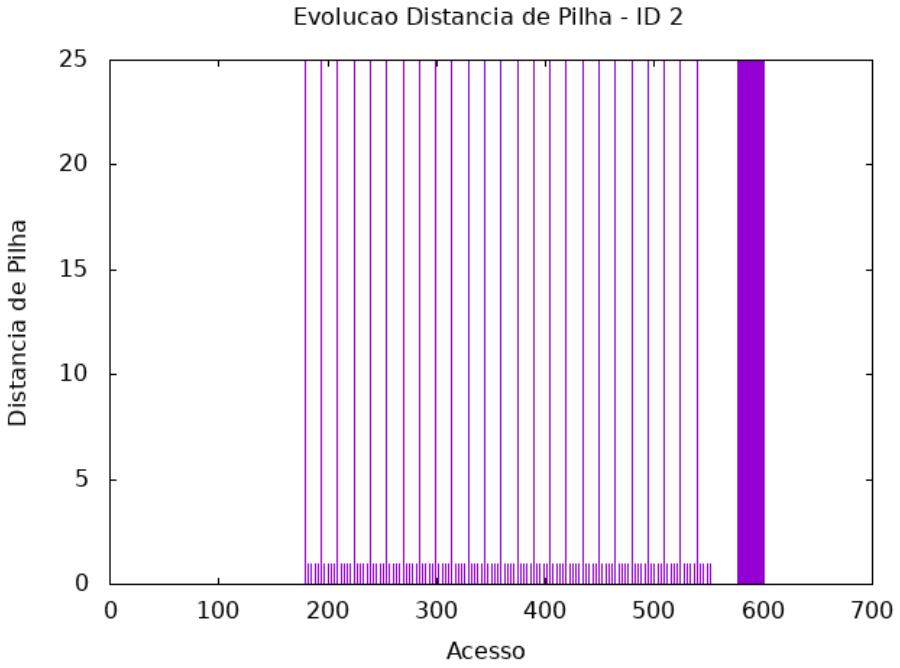
Partiremos agora para o uso da função *multiplicaMatrizes*, em que as linhas da matriz A são acessadas 5 vezes cada uma, enquanto as colunas da matriz B também são acessadas da mesma forma, isso ocorre porque o algoritmo de multiplicação de matrizes precisa multiplicar os elementos das linhas de A pelos elementos das colunas de B. Como cada linha precisa passar

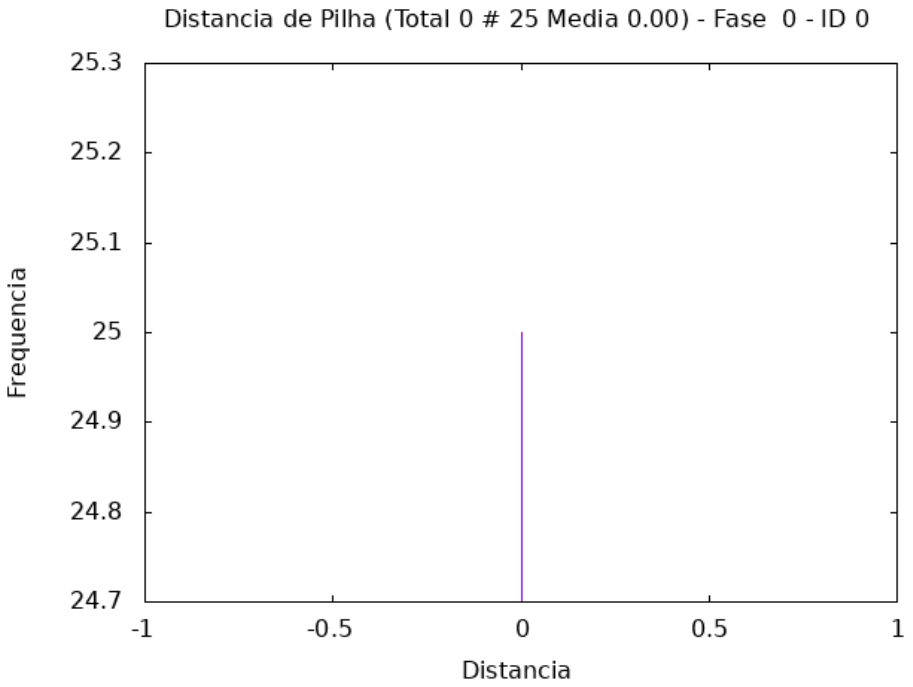
por todas as colunas de B, cada linha precisa ser percorrida 5 vezes.

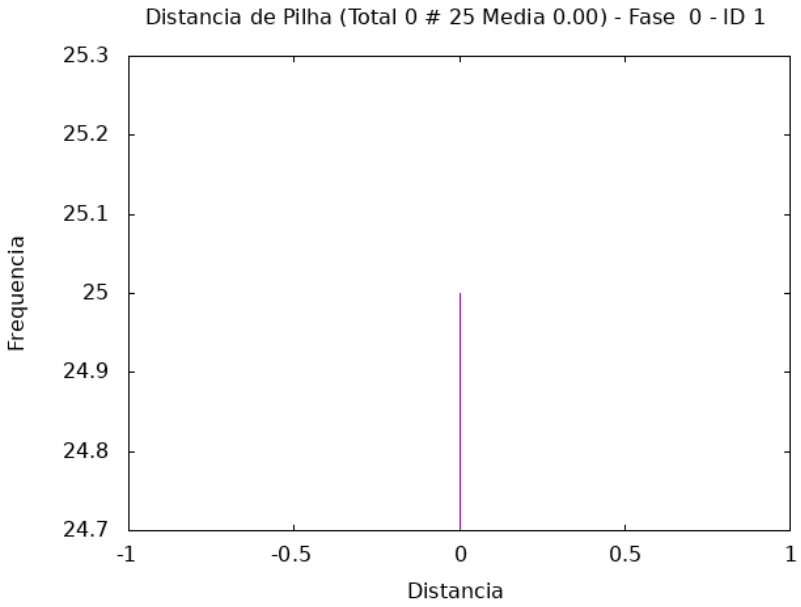
Por outro lado, a matriz C recebe todos os seus atributos novos, como em uma função de inicializar a matriz nula. Porém, a grande diferença que pode ser notada no gráfico de acesso, é que a alocação é realizada de uma maneira mais lenta e os elementos são escritos diversas vezes, isso ocorre já que é preciso percorrer toda uma linha de A e toda uma coluna de B para armazenar apenas um elemento de C, somando todos os valores, um de cada vez, o que gera 5 acessos seguidos à um mesmo endereço de memória. Por último todos os elementos de C são acessados duas vezes, pela função *acessaMatriz* e *escreveArquivoTxt*.

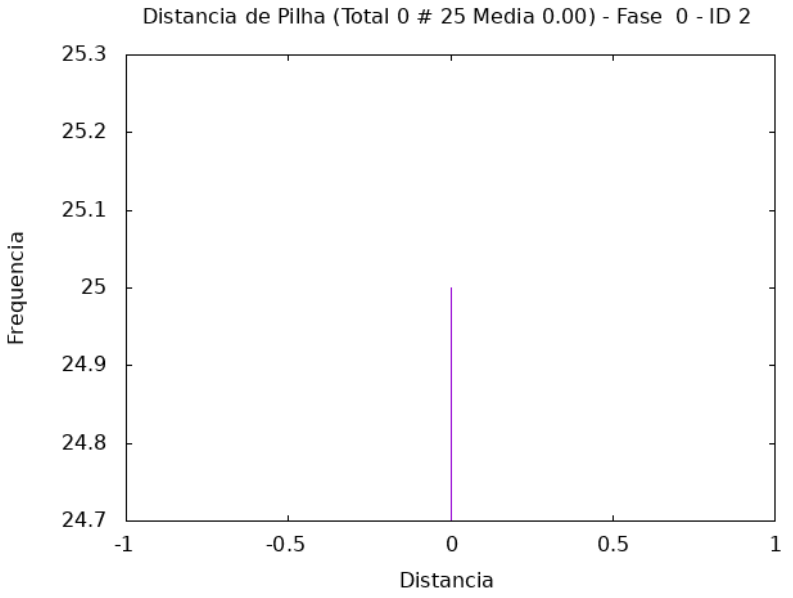


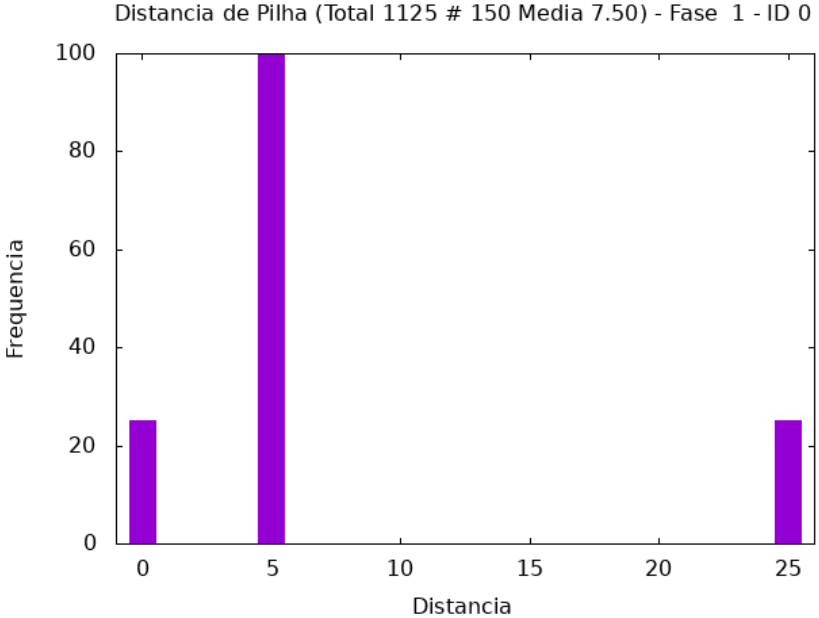


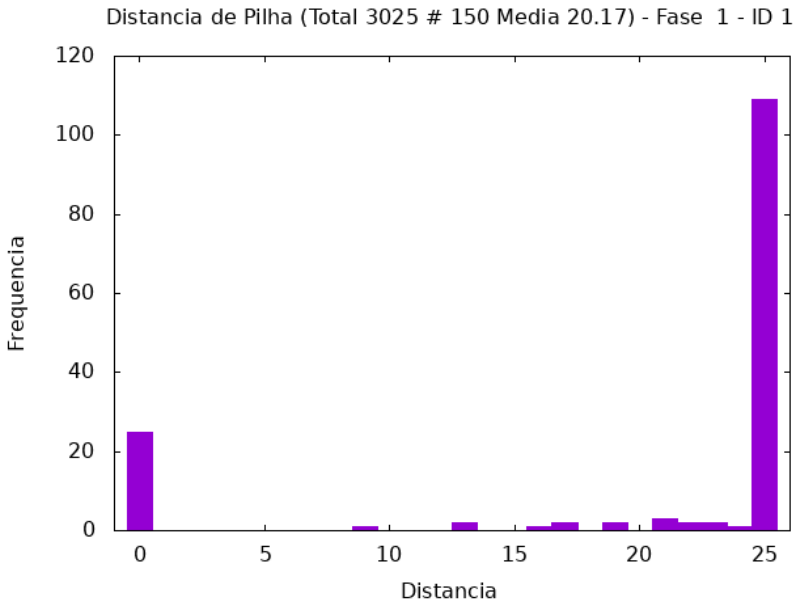

Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso
0
5
10
15
20
25
0
100
200
300
400
500
600
700

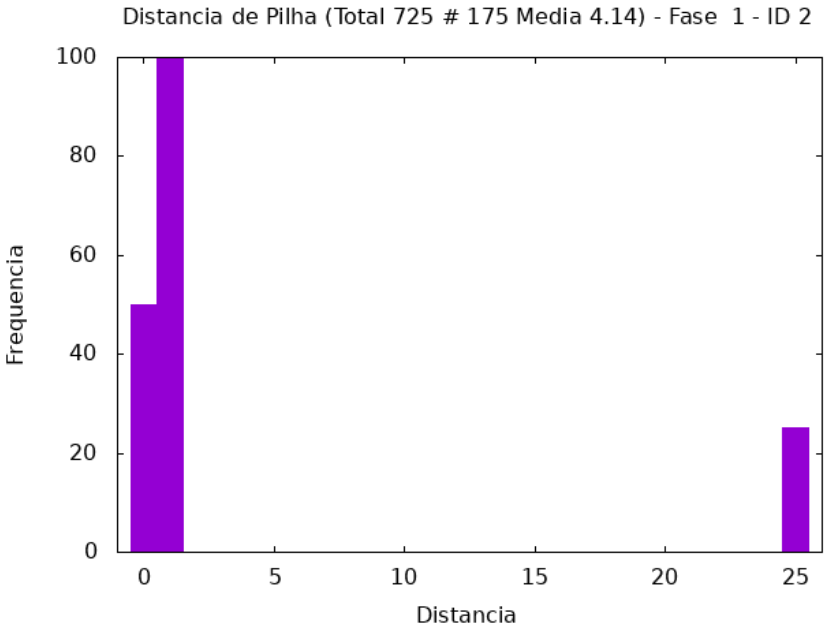

Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 0
Frequencia
Distancia
24.7
24.8
24.9
25
25.1
25.2
25.3
-1
-0.5
0
0.5
1

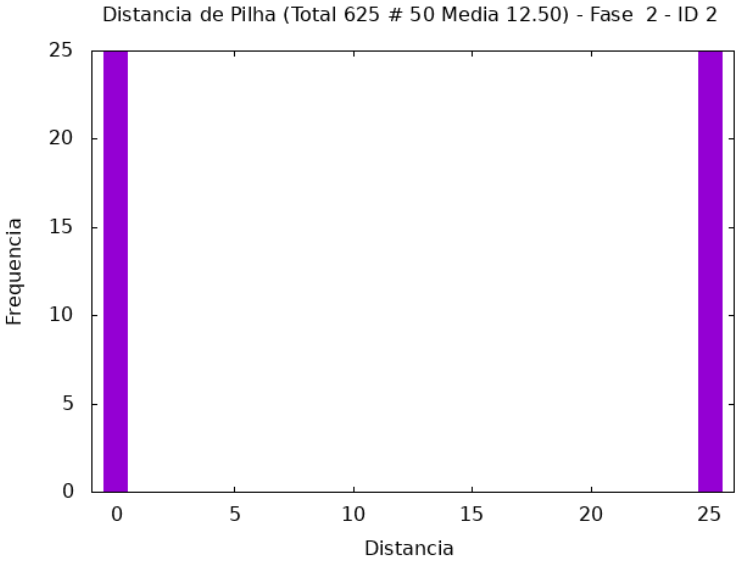

Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 1
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1


Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 2
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1


Distancia de Pilha (Total 1125 # 150 Media 7.50) - Fase 1 - ID 0
Frequencia
Distancia
100
80
60
40
20
0
0
5
10
15
20
25


Distancia de Pilha (Total 3025 # 150 Media 20.17) - Fase 1 - ID 1
Frequencia
Distancia
0
20
40
60
80
100
120
0
5
10
15
20
25


Distancia de Pilha (Total 725 # 175 Media 4.14) - Fase 1 - ID 2
Frequencia
Distancia
0
20
40
60
80
100
0
5
10
15
20
25


Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 2 - ID 2
Frequencia
Distancia
0
5
10
15
20
25
0
5
10
15
20
25

Ao observar os gráficos de distância de pilha e de evolução da distância de pilha, podemos realizar algumas análises e alcançar algumas conclusões interessantes. Lembrando que existem três matrizes de diferentes números de identificação e três fases do processo de multiplicação. Dito isso, podemos realizar uma análise focada na distância de pilha do processo de multiplicação.

Primeiramente, a fase 0 é muito semelhante ao procedimento de soma, em que a distância de pilha das três matrizes é zero, já que como ainda não foram inicializadas, o primeiro acesso começa com número de pilha zero. Dessa forma, todas as matrizes são preenchidas com seus devidos valores sem grandes gastos de memória.

Durante a segunda fase, cada matriz apresenta um comportamento muito distinto das outras duas, começaremos analisando a matriz A. Ao observar o gráfico de distância de pilha da matriz A fica nítido que a maior parte dos acessos apresentam distância de pilha 5. Além disso, o gráfico de evolução nos mostra que primeiramente o acesso apresenta distância de pilha 25 e depois 5. Dessa forma, fica claro que no começo do acesso da linha a distância é 25, mas depois é preciso acessar a mesma linha novamente, o que retorna a distância de pilha igual a 5. Já o gráfico de distância de pilha da matriz B gera um pouco de dúvida, mas é melhor esclarecido ao analisar o gráfico de evolução. Nesse sentido, as colunas de B são acessadas uma a uma, porém, como o método *acessaMatriz* acessa os endereços de B linha a linha, os elementos de B estão “fora de ordem”, então ocorrem algumas inconsistências em relação a distância de pilha da matriz B, o que gera alguns números que em primeira mão não fazem muito sentido. Porém, após percorrer por completo todas as colunas de B, ao realizar o processo novamente, todas as distâncias de pilha serão iguais a 25.

Por último, podemos analisar a matriz C de modo mais individual. Ainda durante a segunda fase, percebemos que prevalecem as distâncias de pilha iguais a 0, 25, e 1. As distâncias de pilha de 0 e 25 já foram explicadas em outros exemplos, a distância 0 ocorre quando o elemento seguinte da pilha é acessado e 25 ocorre quando é preciso percorrer 25 espaços na pilha até chegar ao elemento desejado. Porém, o número de pilha 1 não é muito comum, mas nesse caso ele é muito recorrente. Nesse sentido, o número 1 ocorre quando o mesmo endereço é acessado em sequência, o que ocorre no nosso código, já que cada elemento da matriz C é somado cinco vezes até chegar ao seu valor final. Por fim, durante a última fase as funções *acessaMatriz* e *escreveArquivoTxt* são chamadas, o que gera as distâncias de pilha 0 e 25, assim como analisado no procedimento de soma.

TRANSPOSIÇÃO

- **Desempenho computacional**

Os gráficos abaixo representam a saída da chamada *gprof* do Makefile. Semelhante aos resultados encontrados no procedimento de multiplicação, a função principal de transposição, *transpoeMatriz* é a função que mais demanda tempo de execução. Mesmo sendo chamada apenas uma vez, essa função gasta praticamente 100% do tempo de execução.

Esse fenômeno se deve ao fato que esse procedimento lida com apenas uma matriz ao invés de duas. Nesse sentido, as outras funções não precisam ser executadas muitas vezes, como exemplo a função *acessaMatriz*, que durante o procedimento de multiplicação e soma precisa ser chamada quatro vezes em cada.

Perfil raso:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.15	0.01	0.01	1	10.02	10.02	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	2	0.00	0.00	criaMatriz
0.00	0.01	0.00	1	0.00	0.00	clkDifMemLog
0.00	0.01	0.00	1	0.00	0.00	criaMatrizTxt
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	escreveArquivoTxt
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	parse_args

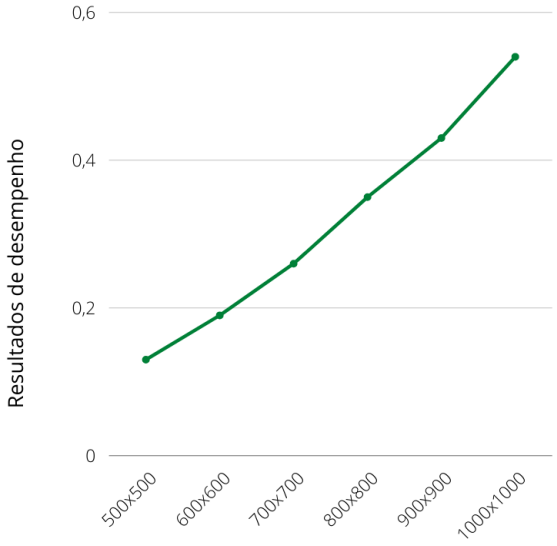
Grafo de execução:

```
index % time    self  children    called     name
                0.01    0.00       1/1           main [2]
[1]    100.0    0.01    0.00       1         transpoeMatriz [1]
                0.00    0.00       1/2           criaMatriz [5]
-----------------------------------------------
                                               <spontaneous>
[2]    100.0    0.00    0.01                 main [2]
                0.01    0.00       1/1           transpoeMatriz [1]
                0.00    0.00       3/3           defineFaseMemLog [3]
                0.00    0.00       2/2           acessaMatriz [4]
                0.00    0.00       1/1           parse_args [13]
                0.00    0.00       1/1           iniciaMemLog [12]
                0.00    0.00       1/1           desativaMemLog [8]
                0.00    0.00       1/1           criaMatrizTxt [7]
                0.00    0.00       1/1           escreveArquivoTxt [10]
                0.00    0.00       1/1           destroiMatriz [9]
                0.00    0.00       1/1           finalizaMemLog [11]
-----------------------------------------------
                0.00    0.00       3/3           main [2]
[3]      0.0    0.00    0.00       3         defineFaseMemLog [3]
-----------------------------------------------
                0.00    0.00       2/2           main [2]
[4]      0.0    0.00    0.00       2         acessaMatriz [4]
-----------------------------------------------
                0.00    0.00       1/2           criaMatrizTxt [7]
                0.00    0.00       1/2           transpoeMatriz [1]
[5]      0.0    0.00    0.00       2         criaMatriz [5]
-----------------------------------------------
                0.00    0.00       1/1           finalizaMemLog [11]
[6]      0.0    0.00    0.00       1         clkDifMemLog [6]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[7]      0.0    0.00    0.00       1         criaMatrizTxt [7]
                0.00    0.00       1/2           criaMatriz [5]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[8]      0.0    0.00    0.00       1         desativaMemLog [8]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[9]      0.0    0.00    0.00       1         destroiMatriz [9]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[10]     0.0    0.00    0.00       1         escreveArquivoTxt [10]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[11]     0.0    0.00    0.00       1         finalizaMemLog [11]
                0.00    0.00       1/1           clkDifMemLog [6]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[12]     0.0    0.00    0.00       1         iniciaMemLog [12]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[13]     0.0    0.00    0.00       1         parse_args [13]
-----------------------------------------------
```

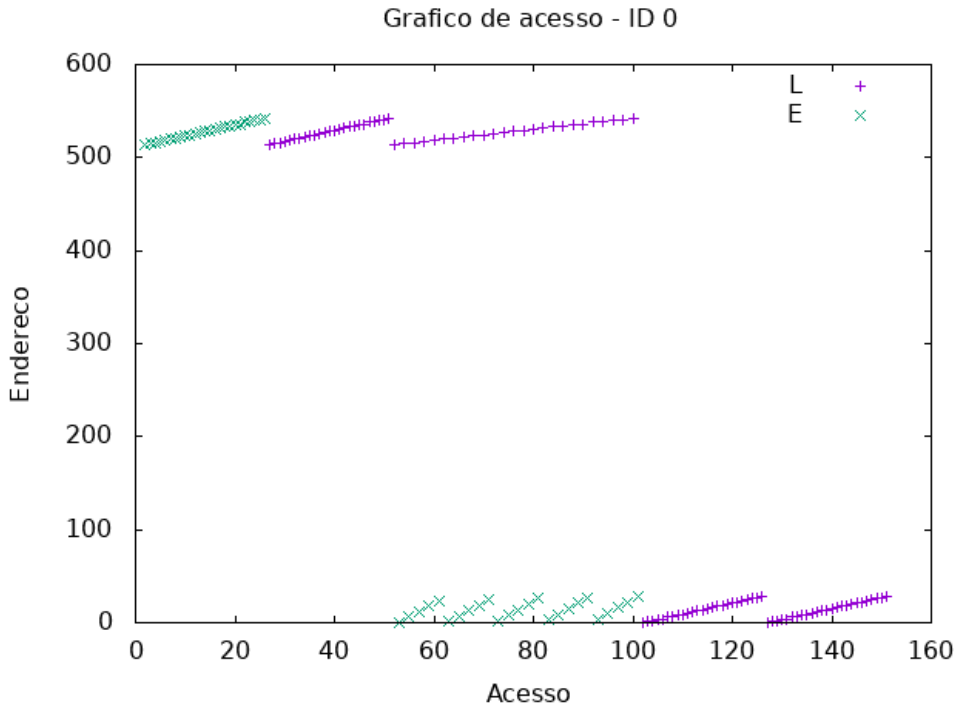
Além disso, podemos também analisar os resultados de desempenho que já foram inicialmente observados na sessão 3. Os resultados de desempenho foram gerados utilizando matrizes de 500x500 até 1000x1000.

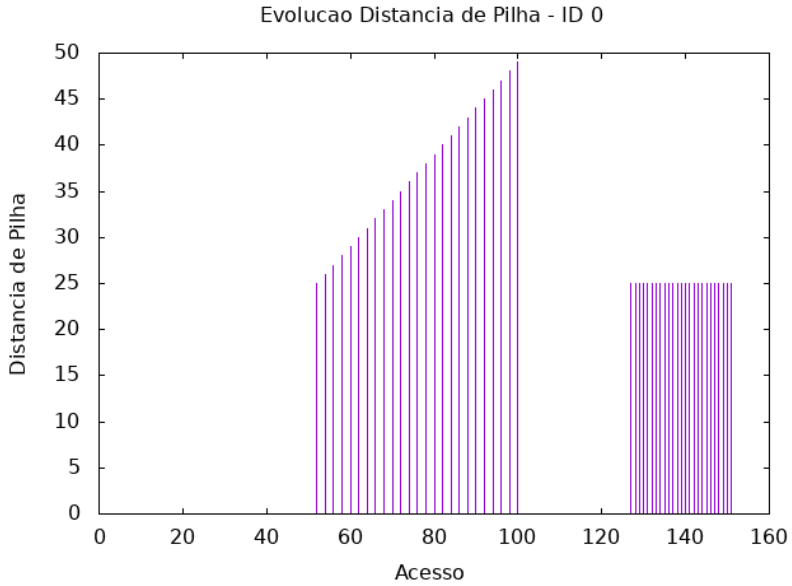
MATRIZES UTILIZADAS	RESULTADO DE DESEMPENHO
500x500	I 1 20791.270292600 F 2 20791.407806300 R 0.137513700
600x600	I 1 20789.780065400 F 2 20789.977054300 R 0.196988900
700x700	I 1 20787.243104500 F 2 20787.508974500 R 0.265870000
800x800	I 1 20782.940981500 F 2 20783.293113800 R 0.352132300
900x900	I 1 20776.531247500 F 2 20776.967109300 R 0.435861800
1000x1000	I 1 20767.200328800 F 2 20767.744061600 R 0.543732800

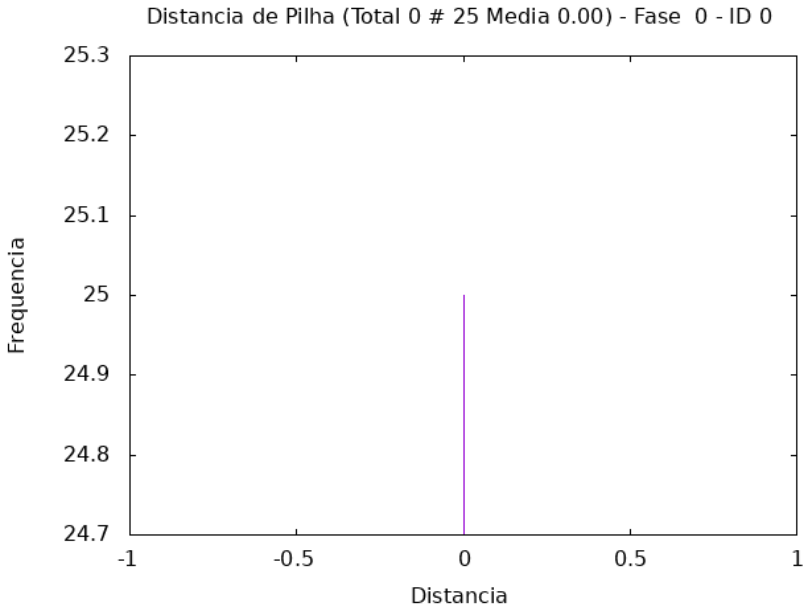
Perceptivelmente, os resultados de desempenho de transposição são os menores entre as três operações. Além disso, sempre que as dimensões das matrizes utilizadas como parâmetros aumentam, o resultado de desempenho também aumenta, ou seja, leva mais tempo para executar o procedimento. Além disso, pode-se notar que com o aumento das dimensões de 500x500 para 600x600 existe uma diferença de tempo de 0.06, porém, ao final, com o aumento de 900x900 para 1000x1000 essa diferença aumenta para 0.11. Nesse sentido, os resultados de desempenho nos mostram um crescimento exponencial semelhante ao crescimento encontrado no procedimento de soma, assim como foi mostrado na análise de tempo na sessão 3. Por fim, podemos criar um gráfico que mostra o crescimento destes resultados de desempenho da operação de transposição de matrizes.

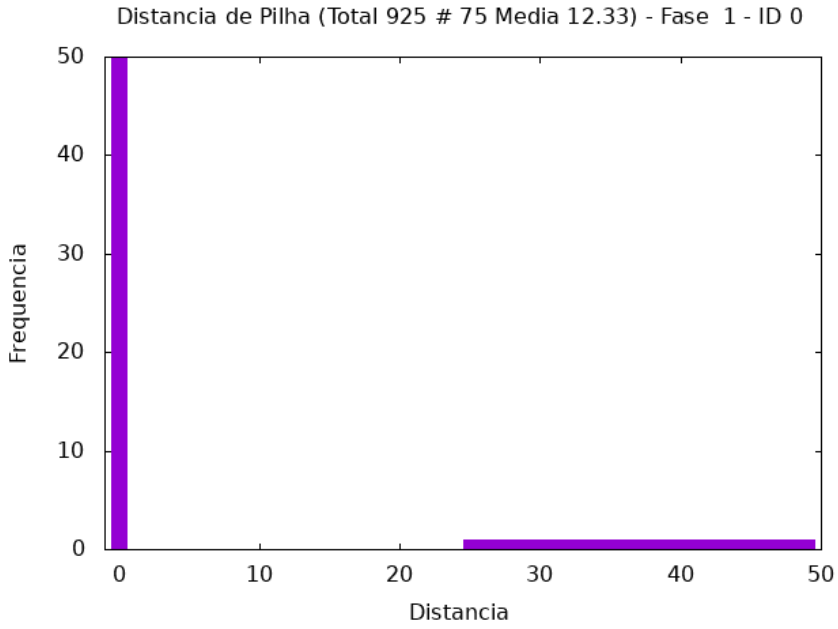


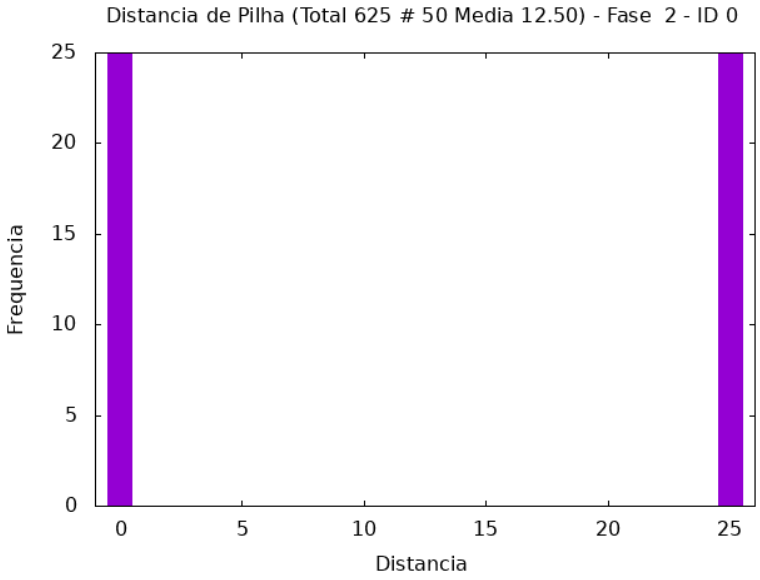
- **Análise de Padrão de Acesso à Memória e Localidade de Referência**




Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
5
10
15
20
25
30
35
40
45
50
0
20
40
60
80
100
120
140
160


Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 0
Frequencia
Distancia
24.7
24.8
24.9
25
25.1
25.2
25.3
-1
-0.5
0
0.5
1


Distancia de Pilha (Total 925 # 75 Media 12.33) - Fase 1 - ID 0
Frequencia
Distancia
0
10
20
30
40
50
0
10
20
30
40
50



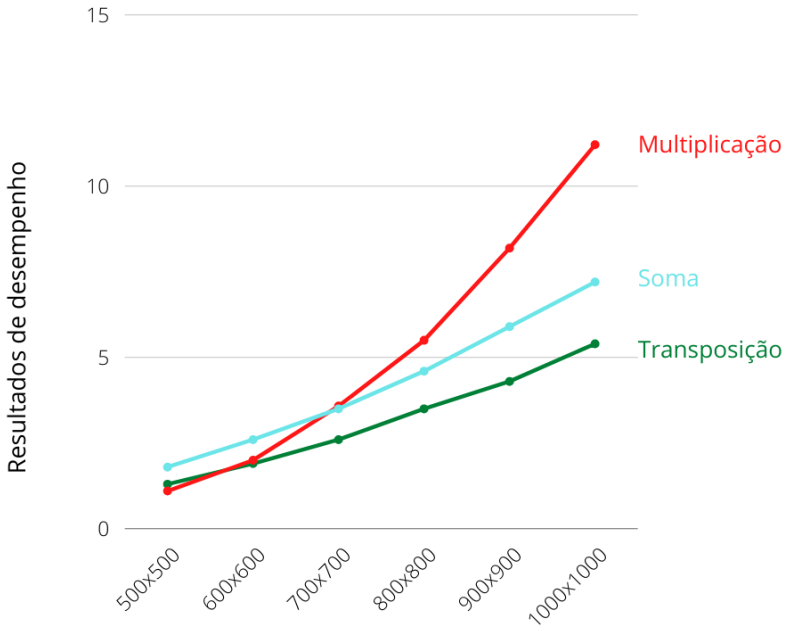
Analisando primeiramente o gráfico de acesso, percebemos que a matriz é inicializada com a função *criaMatrizTxt*, que lê um arquivo e atribui todos os valores da matriz. Em seguida, a matriz é acessada pela função *acessaMatriz* e a função *transpoeMatriz* também é chamada. Nessa função, uma matriz auxiliar é criada para transpor os elementos da matriz A, que são acessados linha a linha, formando a linha roxa nos endereços superiores. Dessa forma, quando os valores de A são lidos linha a linha, eles são atribuídos coluna a coluna na matriz auxiliar, que possui um endereço diferente, por isso a distância dos endereços no gráfico. Por fim, a matriz A recebe o endereço dos atributos da matriz auxiliar, que passa pela função *acessaMatriz* e *escreveArquivoTxt*.

Além disso, podemos interpretar os gráficos de distância e evolução de pilha, para entender melhor o funcionamento desse procedimento de transposição. Na primeira fase, ocorre um comportamento semelhante às outras funções, que é a inicialização da matriz, em que a distância de pilha é zero, já que os endereços nunca haviam sido acessados. Em seguida, na segunda fase, a distância de pilha aumenta linearmente até chegar em 50, já que todos os elementos da matriz auxiliar ainda não foram acessados, sua memória é adicionada a pilha, gerando uma distância de pilha zero, mas quando um elemento da matriz original A é acessado novamente, a distância de pilha aumenta cada vez mais, já que agora existe mais 25 elementos na pilha. Por fim, na última fase ocorre um processo muito semelhante ao procedimento das operações de soma e multiplicação, que é a chamada das funções *acessaMatriz* e *escreveArquivoTxt*, que gera as distâncias de pilha 0 e 25.

CONCLUSÃO – Análise Experimental

Por fim, podemos concluir que a função que mais demanda tempo de execução, além de ter o maior crescimento entre as três principais operações do programa, é o procedimento de multiplicação. Além disso, as funções de soma e transposição apresentam um crescimento semelhante, porém, a transposição ainda apresenta resultados de desempenho menores.

Para deixar a análise dos dados mais visíveis, foram utilizados gráficos para representar o crescimento dos resultados de desempenho de cada uma das operações do programa. Para realizar uma comparação entre as três operações, pode-se observar o seguinte gráfico que comparar o crescimento de cada uma das funções (os valores dos gráficos de soma e transposição foram multiplicados por 10, para deixar o gráfico mais visível).



7. Conclusão

Este trabalho lidou com o problema de implementação de um tipo abstrato de dados de matrizes, em que suas dimensões são alocadas dinamicamente, o que nos permite lidar com dimensões arbitrariamente grandes. Nesse sentido, a abordagem utilizada foi a criação de um programa na linguagem C, que utiliza uma estrutura chamada *matriz,* que implementa seus atributos. Também foram implementadas algumas funções para melhorar a legibilidade e o reuso do código, como por exemplo funções que somam, multiplicam e transpõe matrizes.

Com a solução adotada, pode-se verificar que é possível realizar operações com matrizes de dimensões arbitrariamente grandes, como foi exemplificado durante os testes de performance (seção 4), em que foram utilizadas matrizes de tamanho 1000x1000, ou seja, 1 milhão de elementos. Essa solução permitiu realizar uma análise sucinta de como cada operação funciona na memória, tanto em relação ao tempo como em relação ao espaço, observando os gráficos e resultados de desempenho gerados pela execução do Makefile juntamente com as bibliotecas *analisamem* e *gprof*.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados aos Tipos Abstratos de Dados implementados na linguagem C e melhor entendimento a respeito dos arquivos Makefile. Além disso, durante a implementação da solução para o problema, um dos grandes desafios foi realizar a análise de complexidade espacial e temporal, que ao início do trabalho não estava muito familiarizado, mas o trabalho prático facilitou muito a compreensão.

8. Bibliografia

Slides virtuais e códigos da disciplina de estruturas de dados.Disponibilizado via *moodle*. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

9. Instruções de compilação e execução

- Primeiramente, utilizando um terminal, acesse o diretório TP.
- Ao digitar o comando *make all*, o arquivo Makefile será executado no terminal, e, dessa forma, realizará todas as seguintes operações, que também podem ser executadas de maneira separada (*make perf*, por exemplo):
    - *mem:* Executa a soma, multiplicação e transposição utilizando matrizes quadradas com dimensões 5 por 5, utilizando as matrizes 'm1_5.txt' e 'm2_5.txt' localizadas na pasta *assets*. Por fim, as matrizes de saída são armazenadas na pasta *out*. Além disso, os acessos a memória são registrados e utilizados para a plotagem de gráficos utilizando a biblioteca disponibilizada *analisamem.* Os gráficos são disponibilizados na pasta *tmp*, em que os gráficos da soma, multiplicação e transposição são disponibilizados nas pastas *sumadin*, *multdin*, *transdin*, respectivamente.
    - *perf: Executa também* a soma, multiplicação e transposição utilizando matrizes quadradas com dimensões que variam de 1000 por 1000 para 500 por 500, variando de 100 em 100. Da mesma forma que a execução *mem*, os arquivos são retirados da pasta *assets*, e armazenados os resultados na pasta *out*. Além disso, os resultados de desempenho são armazenados por essa função, que retorna os tempos de execução de cada operação sobre cada matriz. Os resultados de desempenho também são armazenados na pasta *tmp*, para ver os resultados, basta acessar a pasta tmp no terminal e escrever comandos no seguinte formato: '*more mult1000.out*'.
    - *gprof:* Executa também as três operações de matrizes, utilizando matrizes quadradas com dimensões de 1000 por 1000. Ademais, utiliza o gprof para criar o perfil raso e grafo de execução de cada operação, que são armazenadas em arquivos na pasta *tmp* dessa forma: *'tmp/soma1000gprof.txt'*.
    - *clean:* Limpa o diretório de objetos *obj*, excluindo os arquivos *mat.o, matop.o* e *memlog.o*. Além disso, exclui também o arquivo *gmon.out*, localizado na raíz do programa.
    - *all:* Executa os comandos *mem*, *perf* e *gprof*.
- Caso o usuário perfira, é possível executar apenas uma operação utilizando matrizes de sua escolha. Para isso, é preciso executar uma das seguintes linhas de comando, substituindo os nomes dos arquivos txt para os arquivos desejados:
    - **Soma:** 'bin/matop -s 1 m1.txt -2 m2.txt -o res.txt'
    - **Multiplicação:** 'bin/matop -m 1 m1.txt -2 m2.txt -o res.txt'
    - **Soma:** 'bin/matop -s 1 m1.txt -o res.txt'
- Em caso de dúvidas, é possível executar o comando 'bin/matop -h' para visualizar a mensagem de uso do programa.