Trabalho Prático 0
Operações com matrizes alocadas dinamicamente

Flávio Marcílio de Oliveira

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
fmo@ufmg.br

1. INTRODUÇÃO

Matrizes são, sem dúvida nenhuma, representações matemáticas largamente utilizadas em várias áreas do conhecimento. As operações sobre matrizes são de fundamental importância desde as origens da Computação. Exemplos de operações populares são a soma, a multiplicação e a transposição de matrizes.

Um aspecto desafiador é que os tamanhos das matrizes podem variar significativamente e, frequentemente, essas matrizes estão armazenadas em arquivos, sendo necessário lê-las antes de iniciar o processamento.

O problema proposto consiste em implementar um programa capaz de realizar as três operações sobre matrizes alocadas dinamicamente. Esta implementação deve ser capaz de ler as matrizes de um arquivo texto e executar as operações escolhidas pela passagem de parâmetros na linha de comando e salvar os resultados das operações em outros arquivos texto, também informado na linha de comando. As operações devem ser implementadas seguindo a definição apresentada, de forma a preservar o contrato com as partes que utilizam essas funções.

Este trabalho prático tem como objetivo recordar alguns conceitos fundamentais do desenvolvimento de programas em linguagem C/C++, além de introduzir alguns conceitos em termos de tipos abstratos de dados, em particular a sua abstração, desempenho e robustez.

Para resolver o problema citado, foi criado uma estrutura de dados simples com dois atributos para armazenar as dimensões da matriz e um vetor de vetores do tipo double para armazenar os valores da matriz. As operações foram implementadas seguindo as definições passadas.

Este documento está organizado da seguinte forma: Na seção 2 são apresentados os aspectos de implementação, detalhando a estrutura de dados utilizada, depois é apresentada a forma como o projeto está organizado e por último uma explicação do funcionamento de cada função implementada. Na seção 3 é apresentada as análises de complexidade tanto de tempo quanto de espaço. Na seção 4 são apresentadas as estratégias que garantem a robustez do código. Na seção 5 são apresentadas as análises experimentais realizadas quantificando o desempenho quanto ao tempo de execução e quanto a localidade de referência no uso da

memória. Por fim, a seção 6 apresenta as conclusões obtidas com o trabalho. No apêndice A são apresentadas as informações para compilação e execução do programa.

2. IMPLEMENTAÇÃO

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

2.1. Estrutura de Dados

A implementação do programa teve como base o TAD Matriz apresentado nos slides da aula[1] e o código de exemplo disponibilizado na disciplina com três operações principais: soma, multiplicação e transposição.

Esta estrutura de dados consiste em um vetor de vetores do tipo 'double', dois inteiros para representar a quantidade de linhas e colunas da matriz, considerando apenas matrizes bidimensionais e, por fim, um inteiro para representar uma identificação de cada matriz para as análises de Localidade de Referência.

Para facilitar o desenvolvimento e implementação desse TAD com suas operações, foram criadas algumas funções úteis: *criaMatriz* para fazer a alocação dinâmica da matriz com os dados passados por parâmetro; *inicializaMatrizNula* para inicializar uma matriz com zeros em todas as posições; *inicializaMatrizAleatoria* para inicializar uma matriz com valores gerados aleatoriamente; *acessaMatriz* responsável por acessar os elementos da matriz (utilizada para fins de análise de Localidade de Referência); *imprimeMatriz* imprime uma matriz com identificação de linhas e colunas; *escreveElemento* para escrever um valor em uma posição específica da matriz; *leElemento* para ler um elemento específico da matriz; *copiaMatriz* para fazer uma cópia da matriz; *destroiMatriz* para fazer a desalocação de memória das matrizes;

As três operações principais da estrutura de dados Matriz foram implementadas nas seguintes funções: *somaMatrizes*, *multiplicaMatrizes* e *transpoeMatriz*.

2.2. Organização do Código

O projeto foi separado em diretórios para melhor organização. Na raiz do projeto está apenas o Makefile responsável por fazer a compilação de todo o projeto. Na pasta **bin** estará o executável do programa gerado após a compilação. Na pasta **include** estão os arquivos de cabeçalho com as declarações da estrutura de dados e funções relacionadas (mat.h), da biblioteca de análise de desempenho e de localidade de referência (memlog.h) e da biblioteca para garantir a robustez do código (msgassert.h). Na pasta **obj** ficarão os arquivos gerados para linkar na compilação do projeto e gerar o executável. Por fim, na pasta **src** estão os arquivos de implementação (mat.c, matop.c, memlog.c).

2.3. Funcionamento do Programa

Para uma melhor compreensão do projeto, apresento uma descrição do funcionamento de cada função da estrutura de dados e, por último, apresento as sequências de chamadas destas funções na execução do programa principal.

2.3.1. Código de operações - mat.c

criaMatriz - Esta função recebe, como parâmetros, um ponteiro para o TAD Matriz, e três inteiros. Os primeiro e segundo inteiros representam, respectivamente, os números de linhas e colunas da matriz e o terceiro representa o identificador desta matriz. Todos estes valores são passados para o TAD. Com o número de linhas N, é feita a alocação dinâmica de um vetor de tamanho N para armazenar vetores de double. Para cada N linha é feita a alocação dinâmica de vetores para armazenar M valores double, onde M é o número de colunas passado.

inicializaMatrizNula - Esta função recebe um ponteiro para o TAD Matriz como parâmetro e preenche esta matriz com zeros em todas as suas posições.

inicializaMatrizAleatoria - Esta função recebe um ponteiro para o TAD Matriz como parâmetro e preenche todas as posições desta matriz com valores gerados aleatoriamente utilizando uma função própria da linguagem C de geração de número aleatórios com um limite especificado.

acessaMatriz - Esta função recebe, como parâmetro, um ponteiro para uma Matriz e faz o acesso a cada elemento desta matriz. O objetivo desta função é fazer um primeiro acesso a todos os elementos da matriz e, dessa forma, permitir que a análise de localidade de referência possa ser realizada.

imprimeMatriz - Esta função imprime na tela a matriz passada como parâmetro, identificando as linhas e colunas. Utilizado para computar, na análise de localidade de referência, o acesso final da matriz no momento de apresentar o resultado para o usuário.

escreveElemento - Esta função é responsável por escrever um valor, passado por parâmetro, em uma posição específica de uma matriz também passados por parâmetros.

leElemento - Esta função é responsável por ler e retornar um valor de uma posição específica de uma matriz que foram passados por parâmetro.

copiaMatriz - Esta função tem o objetivo criar uma cópia de uma matriz e para isto, recebe duas referências de matrizes e um identificador para a segunda matriz. Dentro desta função, as funções *criaMatriz* e *inicializaMatrizNula* são chamadas para criar e inicializar a matriz de destino. Depois cada elemento da matriz original é copiado para a matriz de destino.

somaMatrizes - Esta função recebe a referência de três matrizes, faz a soma de duas e escreve o resultado na terceira. Para executar essa operação, os elementos são somados um a um.

multiplicaMatrizes - Esta função recebe a referência de três matrizes, faz a multiplicação de duas e guarda o resultado na terceira. A multiplicação das matrizes é feita utilizando o produto interno.

transpoeMatriz - Esta função recebe a referência de uma matriz apenas. Para fazer esta operação é criada uma matriz auxiliar com as dimensões trocadas, ou seja, número de linhas igual ao número de colunas da matriz recebida e número de colunas igual ao número de linhas da matriz recebida. Esta matriz é inicializada para evitar valores espúrios e em seguida cada elemento da matriz recebida por referência é lido e então é gravado na matriz criada na posição invertida. A seguir, é feita a desalocação da memória da matriz recebida e, então, a referência é direcionada para a matriz criada.

destroiMatriz - Esta função é responsável por fazer a liberação da memória das matrizes. Primeiro é feita a desalocação dos vetores correspondentes a cada linha (armazenava as colunas) e depois, a desalocação do vetor das linhas.

2.3.2. Código principal - matop.c

Dentro deste arquivo tem a função ***parse_args*** responsável por ler e interpretar todos os comandos passados na linha de comando ou apresentar as opções da linha de comando quando os argumentos não são especificados. Ainda temos a função ***leMatriz*** que é responsável por ler as matrizes a partir de um arquivo de texto passado por referência e gravar na memória. Temos também a função ***escreveMatriz*** que é responsável por escrever o resultado em um arquivo de texto.

main - Função principal do programa. Inicialmente são criadas as referências para três matrizes que serão utilizadas em todo o programa. Depois é feita a leitura dos parâmetros da linha de comando utilizando a função ***parse_args*** e iniciado a gravação do arquivo para análise de desempenho chamando a função ***iniciaMemLog***. Se a opção de registro de acesso (***-l***) estiver na linha de comando, o registro de acesso à memória é ativado utilizando a função ***ativaMemLog*** e desativa caso contrário (***desativaMemLog***). Em seguida é executada a operação que o usuário escolheu (***-s*** para soma, ***-m*** para multiplicação ou ***-t*** para transposição). Em cada uma destas operações seguimos praticamente os mesmos passos: a) Fase 0 do registro de acesso da memória - alocação de memória e inicialização das matrizes (caso seja escolhida a geração aleatória de matrizes) ou leitura das matrizes a partir de um arquivo de texto; b) Fase 1 do registro de acesso da memória - faz-se um primeiro acesso das matrizes e em seguida é realizada a operação escolhida; c) Fase 2 do registro de acesso da memória - gravação dos resultados e desalocação de memória. O programa é finalizado e a gravação do acesso à memória é encerrada chamando a função ***finalizaMemLog***.

3. ANÁLISE DE COMPLEXIDADE

3.1. Complexidade de Tempo

A complexidade de tempo foi calculada considerando que as operações de atribuição, de alocação e desalocação de memória e escrever na tela é *O(1)*.

criaMatriz: esta função é *O(n)*, pois tem três atribuições, uma alocação de memória externa e um loop de alocações de memória, ou seja, $O(1)+O(1)+O(1)+O(1)+nO(1) = O(n)$.

inicializaMatrizNula: esta função é $O(n^2)$, pois tem dois loops aninhados e uma operação de atribuição no loop mais interno, ou seja, $n^2O(1) = O(n^2)$.

inicializaMatrizAleatória: esta função é $O(n^2)$, pois tem dois loops aninhados e uma operação de atribuição no loop mais interno. Na operação de atribuição temos uma chamada de uma função de geração de números aleatórios, porém, também é uma função *O(1)* pois não depende de nenhum valor externo. Assim, temos, $n^2O(1) = O(n^2)$.

acessaMatriz: esta função é $O(n^2)$, pois, tem uma atribuição e dois loops aninhados com uma operação de atribuição e outra operação de soma e atribuição no loop mais interno, ou seja, $O(1) + n^2(O(1) + O(1)) = O(1) + O(n^2) = O(n^2)$.

imprimeMatriz: esta função é $O(n^2)$, pois temos uma operação de impressão, um loop com uma operação de impressão interna e outra impressão externa a este loop, depois temos dois loops aninhados com operações de impressão interna a ambos os loops. Desta forma, podemos calcular da sequinte forma: $O(1) + nO(1) + O(1) + n(O(1) + nO(1) + O(1)) = O(1) + O(n) + n(O(1)+O(n)) = O(n) + nO(n) = O(n) + O(n^2) = O(n^2)$.

escreveElemento: esta função é *O(1)* pois tem apenas uma atribuição.

leElemento: esta função é *O(1)* pois tem apenas a leitura de um elemento.

copiaMatriz: esta função é $O(n^2)$, pois, chama as funções *criaMatriz* que é *O(n)* e *incializaMatrizNula* que é $O(n^2)$ e também tem dois loops aninhados com uma operação de atribuição interna, ou seja, temos $O(n) + O(n^2) + n^2O(1) = O(n) + O(n^2) + O(n^2) = O(n^2)$.

somaMatrizes: esta função é $O(n^2)$, pois, tem dois loops aninhados com uma operação de atribuição interna com uma operação de soma, ou seja, temos $n^2O(1) = O(n^2)$.

multiplicaMatrizes: esta função é $O(n^3)$, pois tem três loops aninhados com operações que são *O(1)* no loop mais interno, ou seja, $n^3O(1) = O(n^3)$.

transpoeMatriz: esta função é $O(n^2)$, pois chama as funções *criaMatriz* que é *O(n)* e *incializaMatrizNula* que é $O(n^2)$, tem dois loops aninhados com com uma operação de atribuição que é *O(1)* no loop mais interno e fora dos loops tem mais uma chamada a função *destroiMatriz* que é *O(n)*, ou seja, $O(n) + O(n^2) + n^2O(1) + O(n) = O(n^2)$.

destroiMatriz: esta função é *O(n)*, pois tem um loop com uma operação de desalocação de memória interna e outra operação de desalocação de memória fora do loop, ou seja, *nO(1) + O(1) = O(n)*.

3.2. Complexidade de espaço

criaMatriz: como esta função armazena um vetor de vetores, ela é *O(n)*.

inicializaMatrizNula: esta função é $O(n^2)$ pois armazena uma matriz bidimensional.

inicializaMatrizAleatória: esta função é $O(n^2)$ pois armazena uma matriz bidimensional.

acessaMatriz: esta função é *O(1)* pois a quantidade de memória utilizada é independente da entrada.

imprimeMatriz: esta função é *O(1)* pois armazena apenas os valores dos contadores.

escreveElemento: esta função é *O(1)* pois a quantidade de memória é independente da entrada.

leElemento: esta função é *O(1)* pois a quantidade de memória é independente da entrada.

copiaMatriz: esta função é $O(n^2)$ pois utiliza memória para armazenar uma nova matriz bidimensional.

somaMatrizes: esta função é *O(1)* pois não utiliza nenhuma memória adicional sendo que as matrizes passadas por referência já foram alocadas. Esta função só faz a substituição de valores na matriz C.

multiplicaMatrizes: esta função é *O(1)* pois não utiliza nenhuma memória adicional para a computação, sendo que as matrizes passadas por referência já foram alocadas.

transpoeMatriz: esta função é $O(n^2)$ pois utiliza memória para armazenar uma nova matriz bidimensional auxiliar para fazer a transposição.

destroiMatriz: esta função é *O(1)* pois não utiliza memória adicional. Seu objetivo é apenas liberar a memória utilizada por uma matriz, então, sua complexidade de espaço é constante.

4. ESTRATÉGIAS DE ROBUSTEZ

Para garantir que o programa seja tolerante a falhas, foi utilizada a biblioteca *msgassert.h* disponibilizada junto com o código de exemplo e que contém duas macros: uma de aviso e outra de erro que quando ocorre encerra o programa. Essas macros foram utilizadas nas funções, junto com verificações de consistência dos parâmetros passados para as funções que devem ser satisfeitas antes de prosseguir com a execução das operações.

Nas funções *criaMatriz*, *inicializaMatrizNula*, *inicializaMatrizAleatoria*, *acessaMatriz* e *imprimeMatriz* foram verificados se as dimensões das matrizes são maiores que zero, para

garantir que essas funções sejam executadas em matrizes que tenham sido instanciadas de maneira correta com dimensões válidas.

Nas funções *escreveElemento* e *leElemento* são verificados se os índices para busca e escrita dos valores estão dentro dos limites da matriz passada como parâmetro.

Na função *copiaMatriz* são verificadas as dimensões da matriz que será copiada para evitar que a função seja executada passando uma matriz inválida para ser copiada.

Na função *somaMatrizes* são verificadas se as dimensões das duas matrizes que serão somadas e da matriz que guardará o resultado são compatíveis, ou seja, se as matrizes possuem as mesmas dimensões.

Na função *multiplicaMatrizes* é verificado se o número de colunas da matriz A é igual ao número de linhas da matriz B e se o número de linhas da matriz C é igual ao número de linhas da matriz A e se o número de colunas da matriz C é igual ao número de colunas de B.

Na função *transpoeMatriz* é verificado se as dimensões da matriz são maiores que zero, garantido assim que não seja executada para matrizes inválidas.

Por fim, na função *destroiMatriz* é verificado se a função é chamada para uma matriz que ainda não foi desalocada.

5. ANÁLISE EXPERIMENTAL

A análise experimental foi conduzida para as três operações buscando avaliar tanto o desempenho em questão de tempo de execução quanto para a localidade de referência.

As análises apresentadas nesta seção foram realizadas utilizando um computador com as seguintes especificações:

Processador: Intel(R) Core™ i3 CPU M 370 @ 2.40GHz

RAM: 8,00GB

Sistema operacional de 64 bits: Windows 10 Pro versão 21H1

WSL2: Ubuntu-20.04

5.1. Análise de Desempenho

Para a análise de desempenho do programa foram criados cinco casos de testes para cada operação (multiplicação, soma, transposição). Estes casos de testes consistem na geração de matrizes aleatórias de dimensões 100x100, 200x200, 300x300, 400x400 e 500x500 e em seguida a realização das operações com estas matrizes e o registro do tempo para esta execução.

O desempenho do programa para as três operações pode ser visualizado na Figura 1 abaixo:

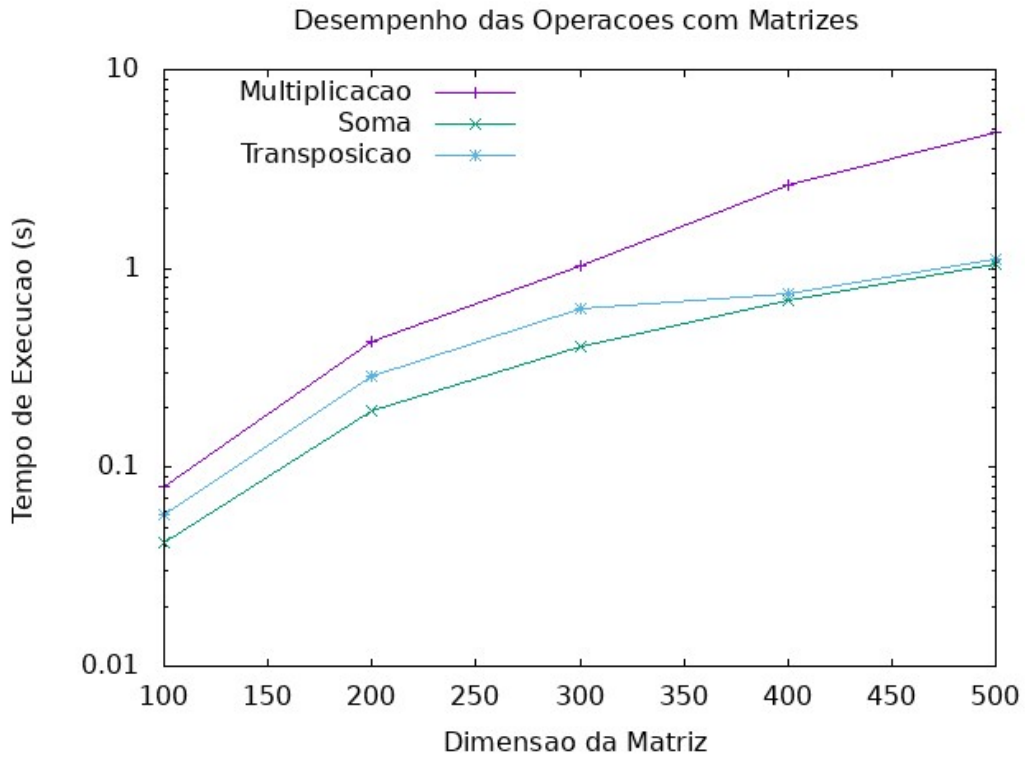

Figura 1 - Análise de desempenho para as operações com matrizes

Neste gráfico em escala logarítmica podemos ver que o tempo de execução da operação de multiplicação cresce muito mais que para as operações de soma e transposição e com um comportamento assintótico, mostrado na Figura 2, que comprova a análise da complexidade de tempo feita na seção 3 deste documento, sendo para a operação de multiplicação $O(n^3)$ e para as operações de soma e transposição $O(n^2)$. Além disso, outra comprovação que obtemos está demonstrada nas curvas de desempenho das operações de soma e transposição, ambas com o mesmo comportamento assintótico, comprovando mais uma vez a análise feita anteriormente.

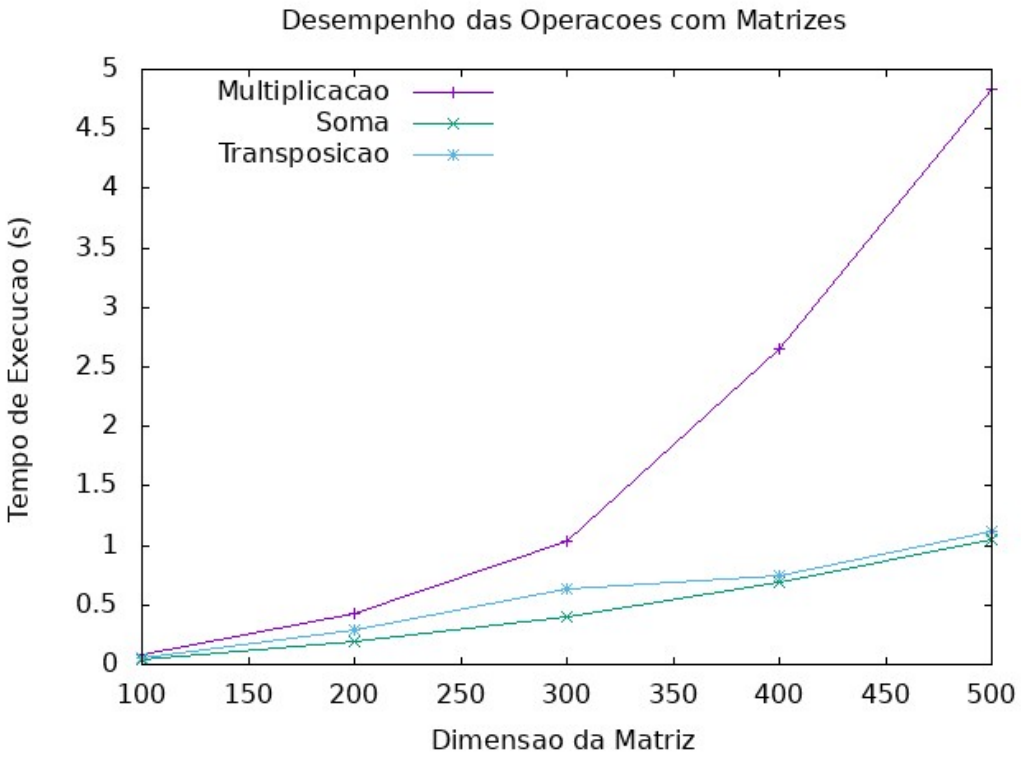

Figura 2 - Comportamento assintótico das operações com matrizes

5.2. Análise de Localidade de Referência

A Localidade de Referência é a tendência de um processador acessar o mesmo conjunto de locais de memória repetidamente por um período curto de tempo[2], ou seja, em períodos curtos de tempo o acesso à memória tende a ser em endereços próximos. Para fazer a análise da Localidade de Referência foram utilizados o Mapa de Acesso à Memória e a Distância de Pilha.

5.2.1. Mapa de Acesso à Memória

Como o processo de registrar o acesso à memória tem um custo maior, então para a análise de localidade de referência foram realizados apenas um teste para cada operação, utilizando matrizes de dimensão 5x5 geradas aleatoriamente.

Multiplicação

O mapa de acesso à memória para a operação de multiplicação é mostrado na Figura 3 para a matriz A (ID = 0).

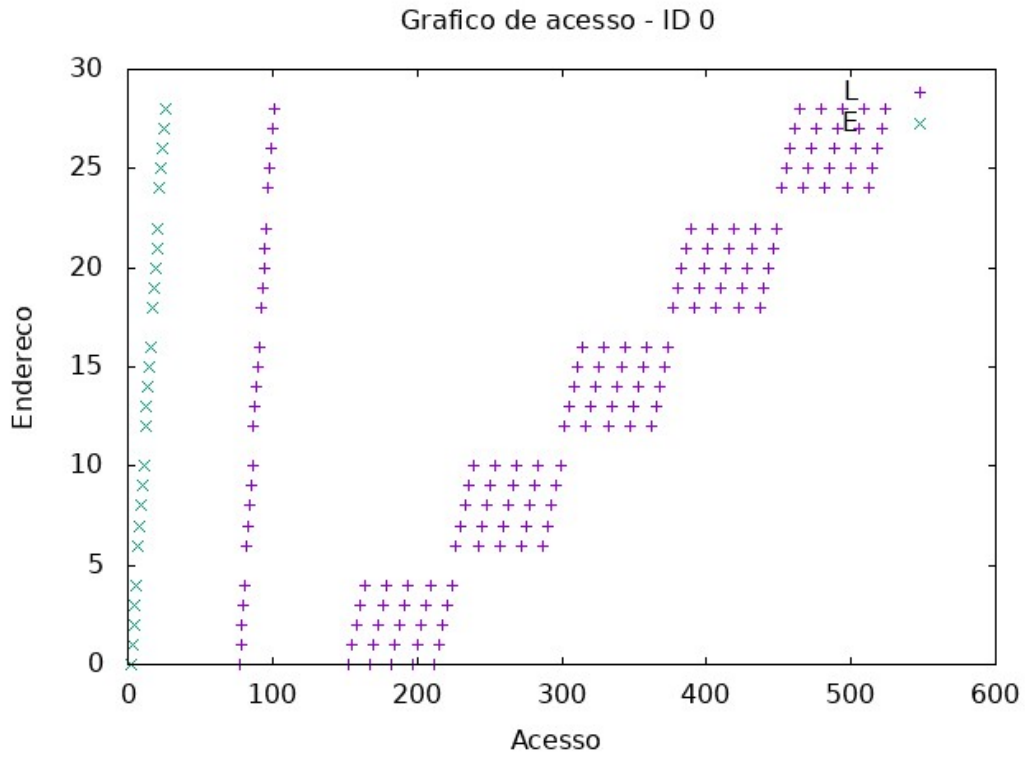

Figura 3 - Mapa de acesso à memória da matriz A

No mapa apresentado vemos o primeiro processo de escrita da matriz A chamada pela função *inializaMatrizAleatoria* e o primeiro processo de leitura feita na Fase 1 com a chamada da função *acessaMatriz*. Depois vemos o processo de leitura feito dentro da função multiplicaMatrizes, onde todas as colunas são percorridas para cada linha, ou seja, todos os elementos de uma linha é percorrido tantas vezes quanto forem o número de linhas da matriz B (neste caso 5 vezes) antes de passar para a próxima linha.

O mapa de acesso à memória para a matriz B (ID = 1) é mostrado na Figura 4.

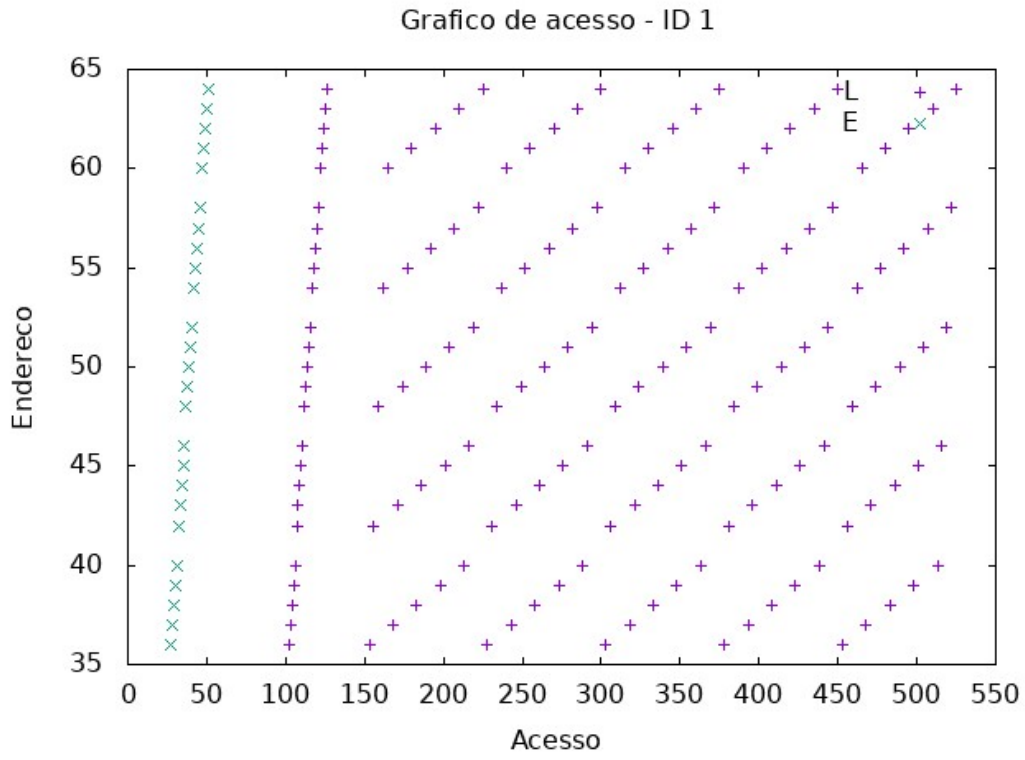

Figura 4 - Mapa de acesso à memória da matriz B

Como pode ser observado na Figura 4, a inicialização da matriz B e o acesso antes da operação de multiplicação também estão sendo mostrados na região mais à esquerda do gráfico. No processo de multiplicação, vemos que a matriz B é acessada de forma diferente, ou seja, as linhas são acessadas para cada coluna, seguindo o processo de multiplicação de matrizes por meio do produto interno.

O resultado da multiplicação das matrizes A e B é gravado na matriz C (ID = 2) que possui um mapa de acesso mostrado na Figura 5.

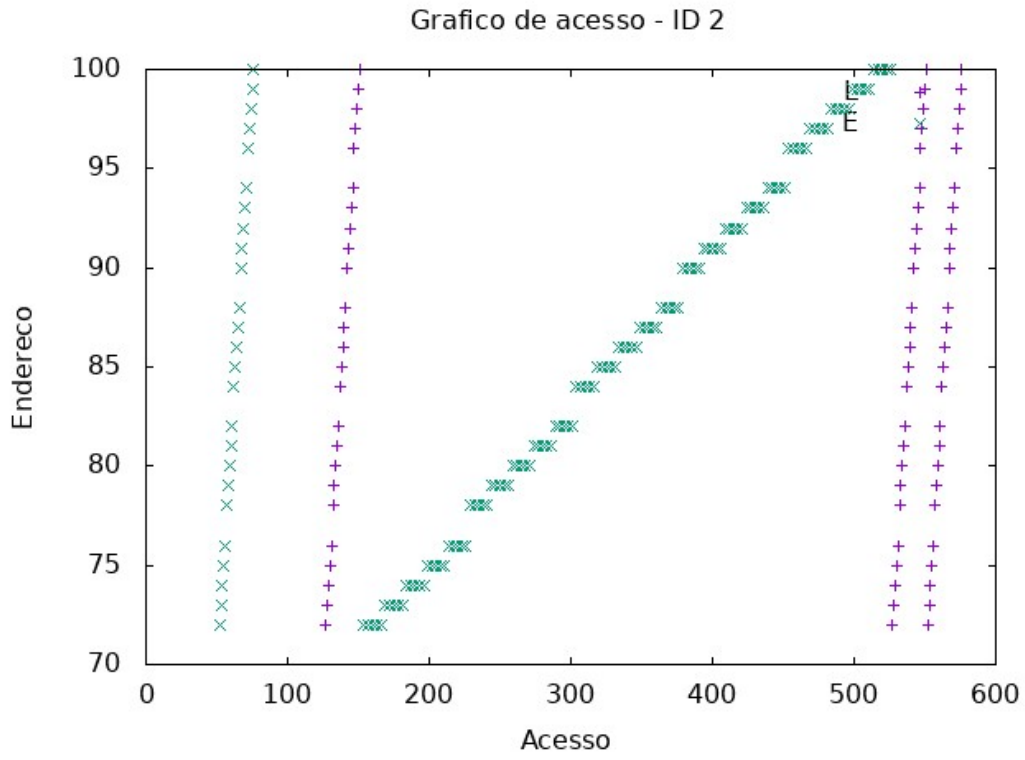

Figura 5 - Mapa de acesso à memória da matriz C

Vemos também a inicialização e a leitura inicial antes da operação nas duas primeiras partes do gráfico. No processo de multiplicação, vemos o processo incremental de escrita para cada linha, ou seja, cada endereço correspondente a um elemento da linha é acessado tantas vezes quanto forem o número de colunas de A e linhas de B (que devem ser iguais e neste caso 5). Depois temos dois acessos de leitura que representam a chamada da função *acessaMatriz* e *imprimeMatriz* na Fase 2 de impressão do resultado para o usuário.

Soma

Como é uma operação de soma elemento com elemento, o mapa de acesso à memória para a matriz A (ID = 0) e para a matriz B (ID = 1) são idênticos e são mostrados, respectivamente, pelas Figuras 6 e 7.

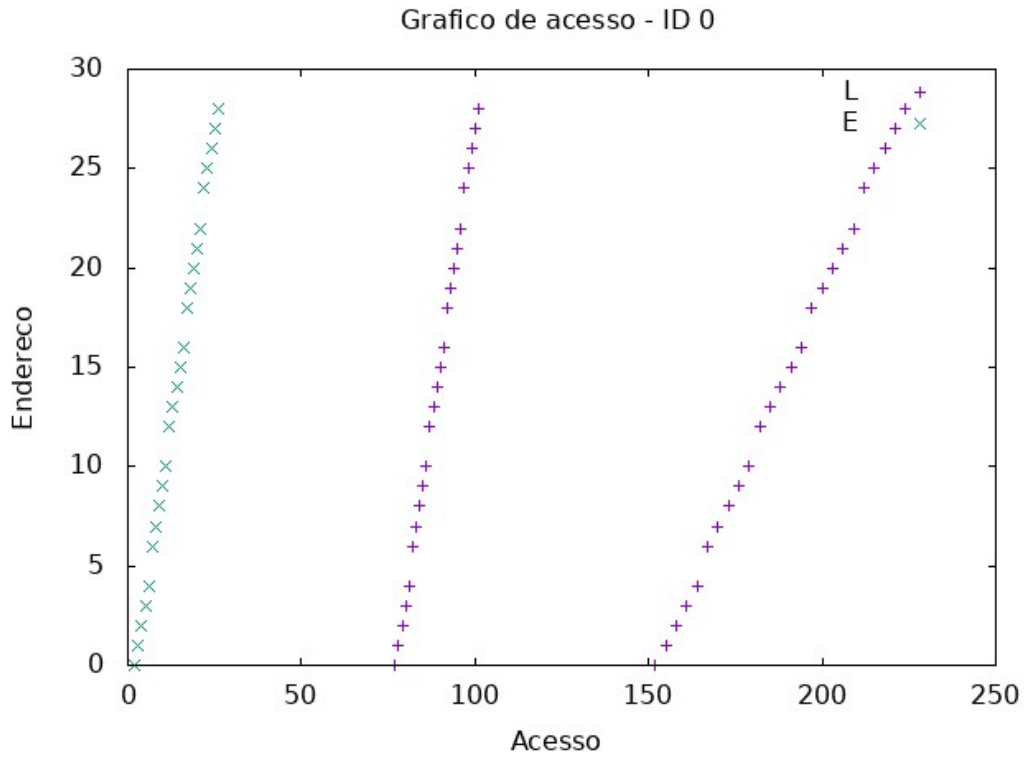

Figura 6 - Mapa de acesso a memória da matriz A

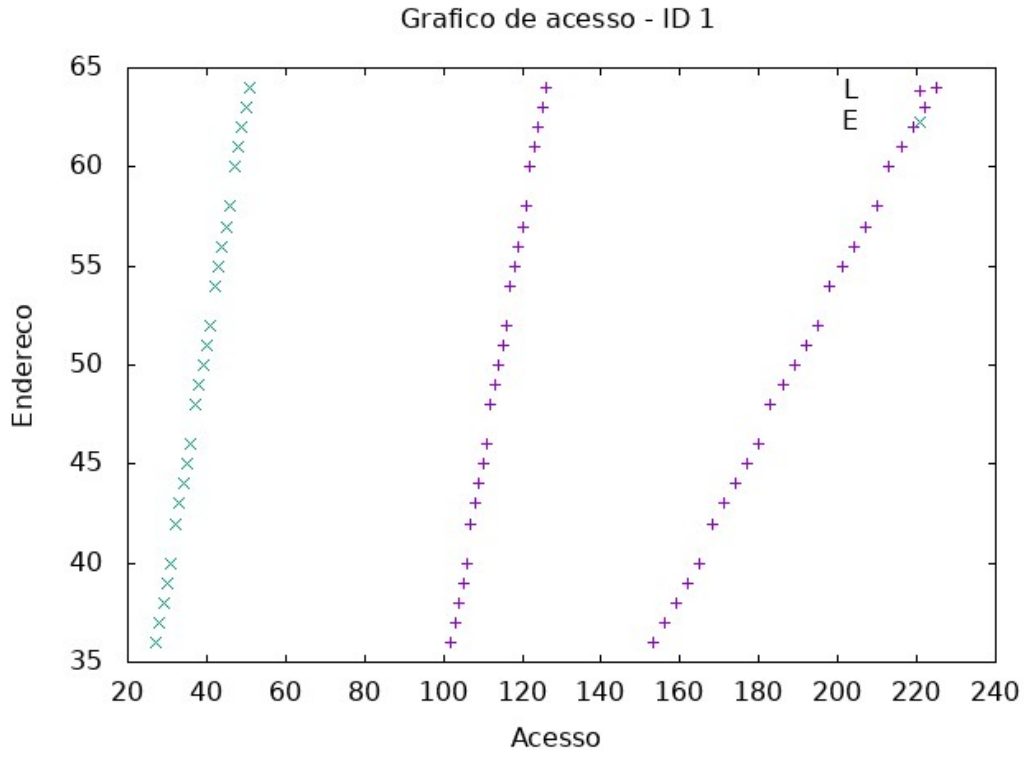

Figura 7 - Mapa de acesso à memória da matriz B

Nas duas Figuras acima, observamos o primeiro acesso de escrita com a inicialização da matriz e um acesso de leitura feita pela função *acessaMatriz*. Na operação de soma, o acesso de leitura é feito lendo as colunas para cada linha de forma sucessiva.

Para a matriz C, temos além das fases apresentadas nos gráficos das matriz A e B, o processo de leitura para apresentação do resultado ao usuário. E como fazemos o uso da função *acessaMatriz* para fins de cálculo da distância de pilha, temos dois processos de leitura da memória como mostrado na Figura 8 abaixo.

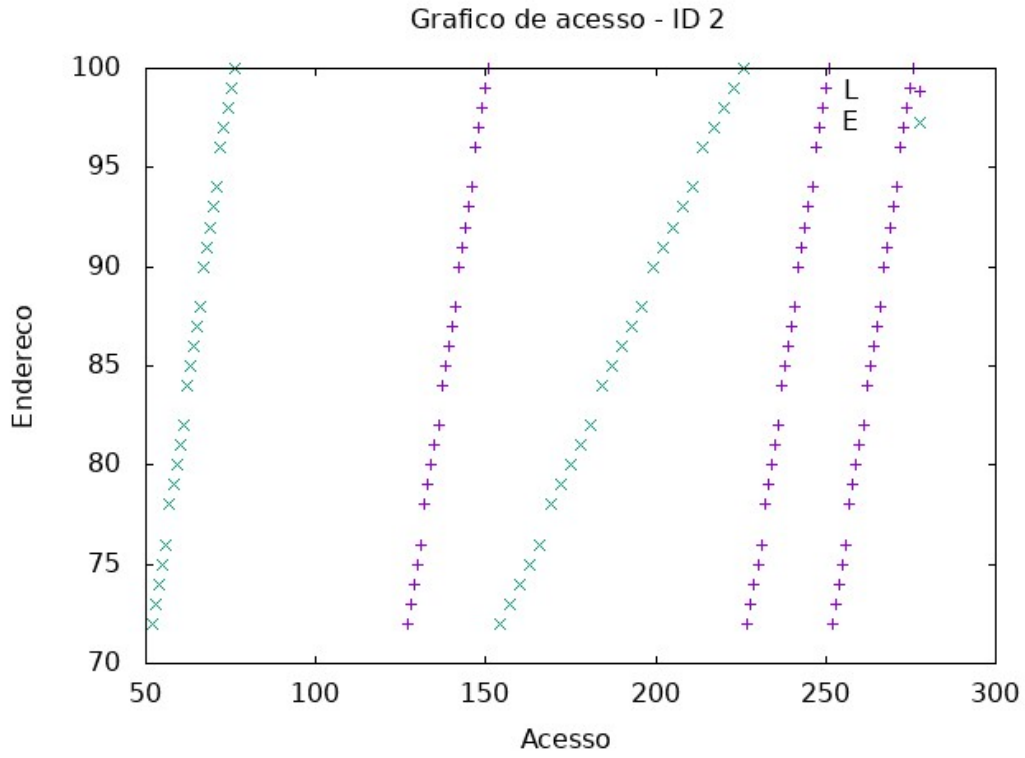

Figura 8 - Mapa de acesso à memória da matriz C

<u>Transposição</u>

Como foi utilizada uma estratégia de criação de uma matriz auxiliar para fazer a transposição da matriz A temos, nesta operação, o mapa de acesso à memória para duas matrizes sendo que o ID 0 é da matriz A até o momento de leitura e o ID 1 da matriz auxiliar que é criada e inicializada dentro da função transpoeMatriz e depois passa a ser a matriz que é impressa para o usuário. Assim, a Figura 9 mostra o mapa de acesso para a matriz A e a Figura 10 mostra o mapa de acesso à memória para a matriz B.

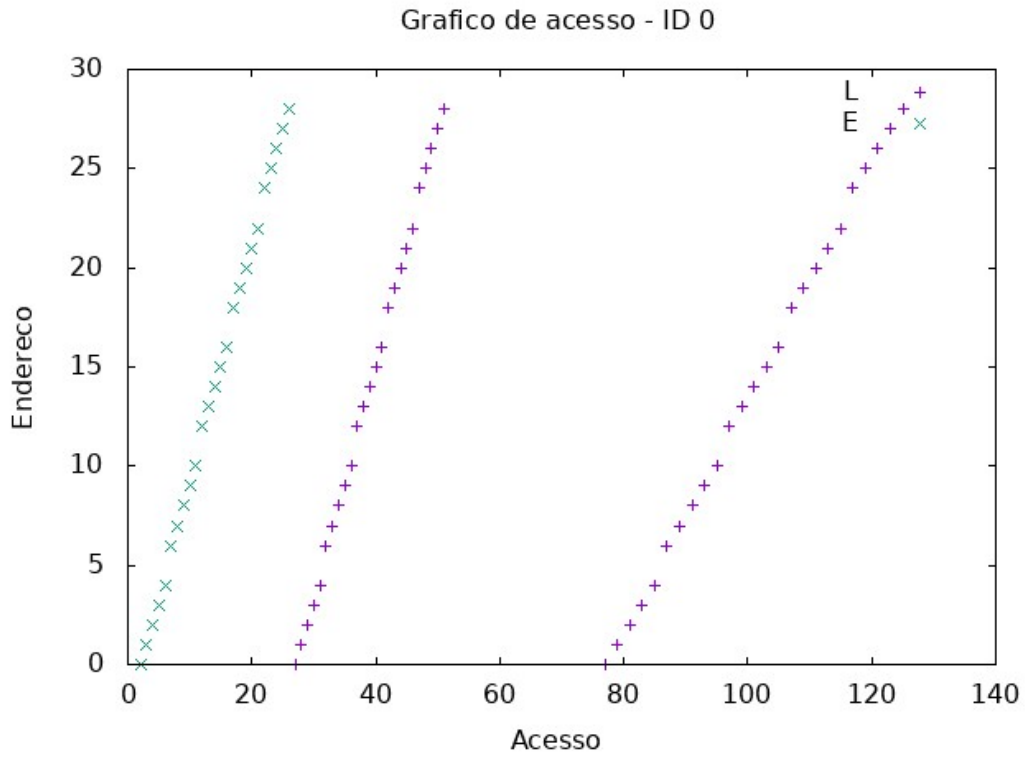

Figura 9 - Mapa de acesso à memória da matriz A

Neste mapa vemos o processo de inicialização da matriz A na Fase 0 e o processo de leitura feita pela função *acessaMatriz* na Fase 1. O segundo processo de leitura ocorre dentro da função transpoeMatriz. Nesta estratégia, a matriz A criada inicialmente é desalocada e o endereço da memória passado é direcionado para a matriz auxiliar que possui um ID diferente. Na Figura 10 é possível ver o acesso da matriz na Fase 2 de apresentação dos resultados.

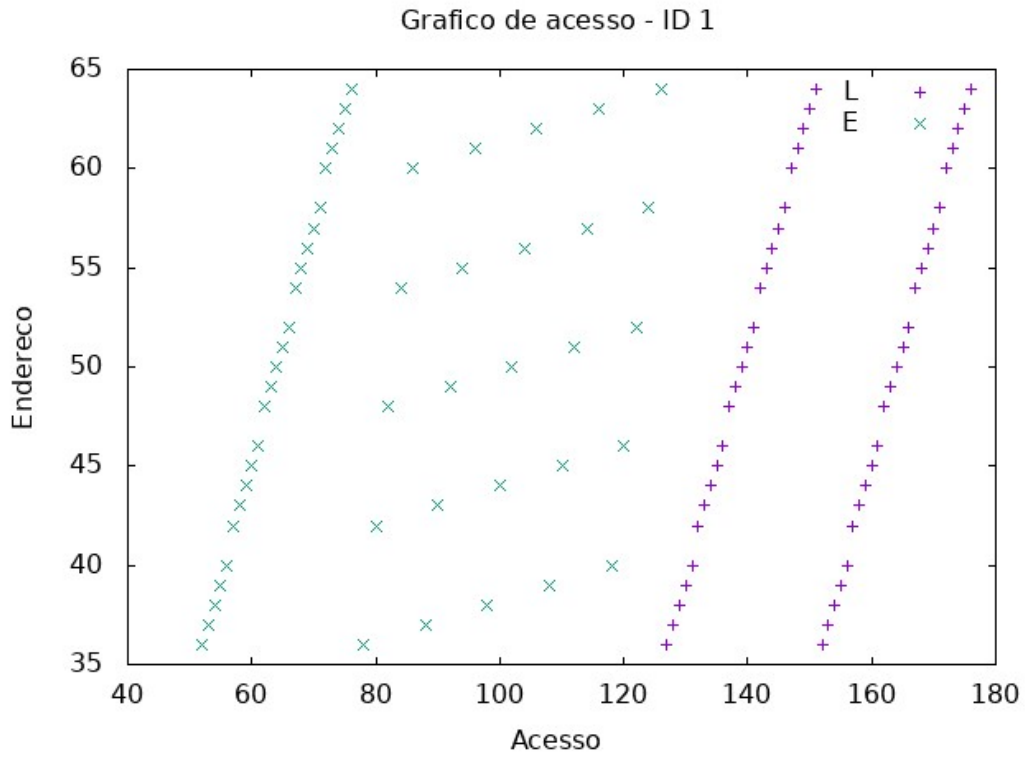

Figura 10 - Mapa de acesso à memória da matriz auxiliar

Neste gráfico da matriz auxiliar é mostrado o processo de escrita inicial na inicialização e em seguida o processo de escrita fazendo a transposição. É possível observar que a matriz é preenchida percorrendo todas as linhas para cada coluna, ao passo que a matriz A está sendo percorrida por todas as colunas para cada linha. Depois do processo de escrita, essa matriz é referenciada pelo ponteiro da matriz A e, assim, temos mais duas leituras que ocorrem no processo de apresentação dos resultados para o usuário na Fase 2.

5.2.2. Distância de Pilha

A distância de pilha corresponde ao número de endereços de memória exclusivos acessados durante um período de reutilização, onde um período de reutilização é o tempo entre dois acessos sucessivos ao mesmo endereço de memória[3].

Multiplicação

O gráfico que mostra as distâncias de pilha para a matriz A na Fase 1 (operação de multiplicação) é mostrado na Figura 11.

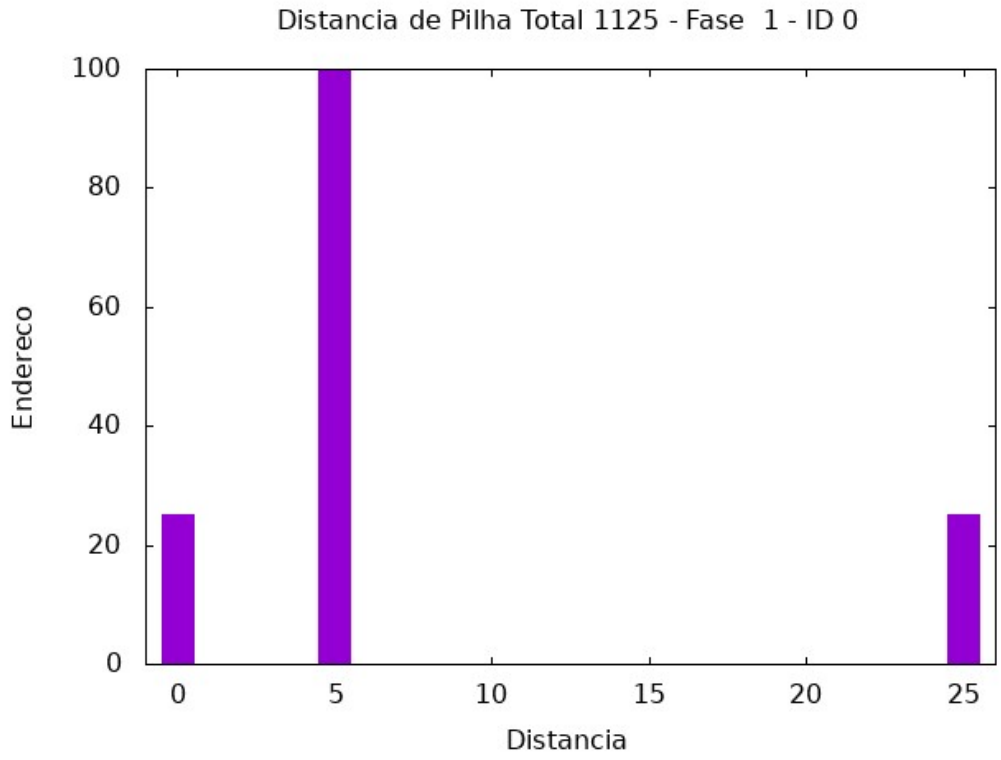

Figura 11 - Distância de pilha no uso da matriz A durante a multiplicação

No gráfico nota-se o primeiro acesso aos elementos da matriz realizado ao chamar a função acessa matriz e, portanto, com distância de pilha 0. Ao realizar este primeiro acesso, é possível calcular a distância de pilha integralmente dentro do processo de multiplicação das matrizes. Neste caso, vemos que, essencialmente, a distância de pilha para a matriz A é igual a 5, e isto é devido ao processo de percorrer a matriz primeiro nas colunas e depois nas linhas. Vemos que este processo tem um custo total medido em distância de pilha de 1125.

Para a matriz B, o gráfico das distâncias de pilha na Fase de multiplicação é mostrado na Figura 12.

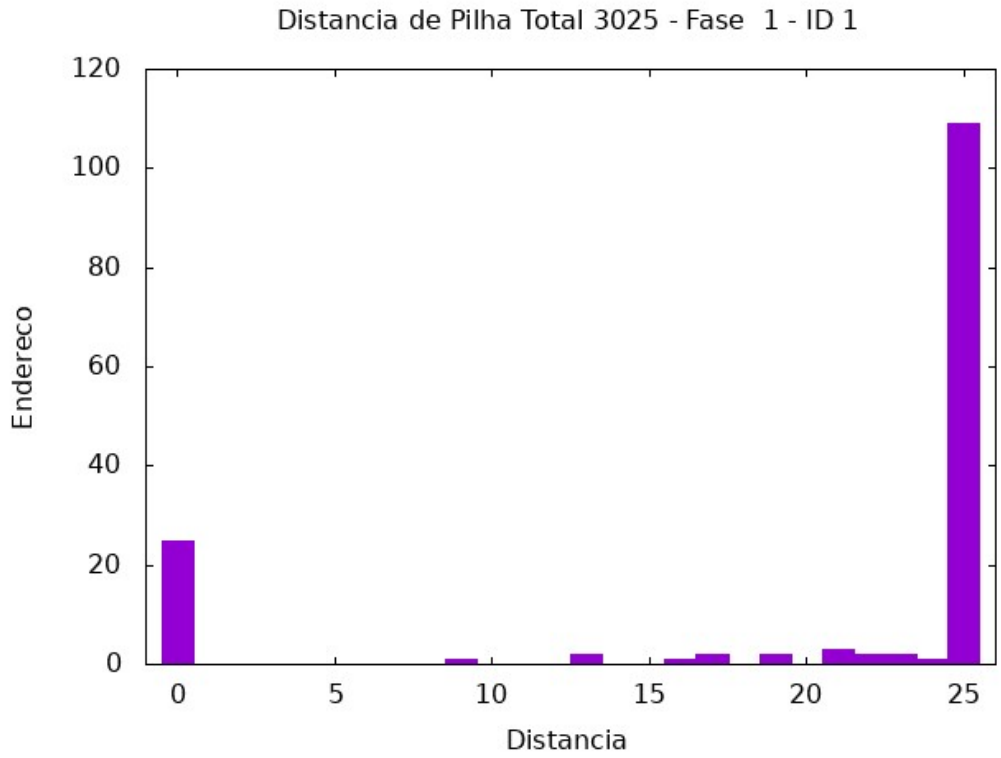

Figura 12 - Distância de pilha no uso da matriz B durante a multiplicação

Seguindo a mesma abordagem, tem-se o primeiro acesso da matriz e em seguida a contagem da distância de pilha dentro do processo de multiplicação. Neste caso, a matriz B é percorrida de forma diferente, ou seja, primeiro as linhas e depois as colunas e isto tem o impacto mostrado no gráfico gerando uma distância de pilha total ao fim do processo de 3025.

As distâncias de pilha para a matriz C na operação de multiplicação são mostradas no gráfico da Figura 13.

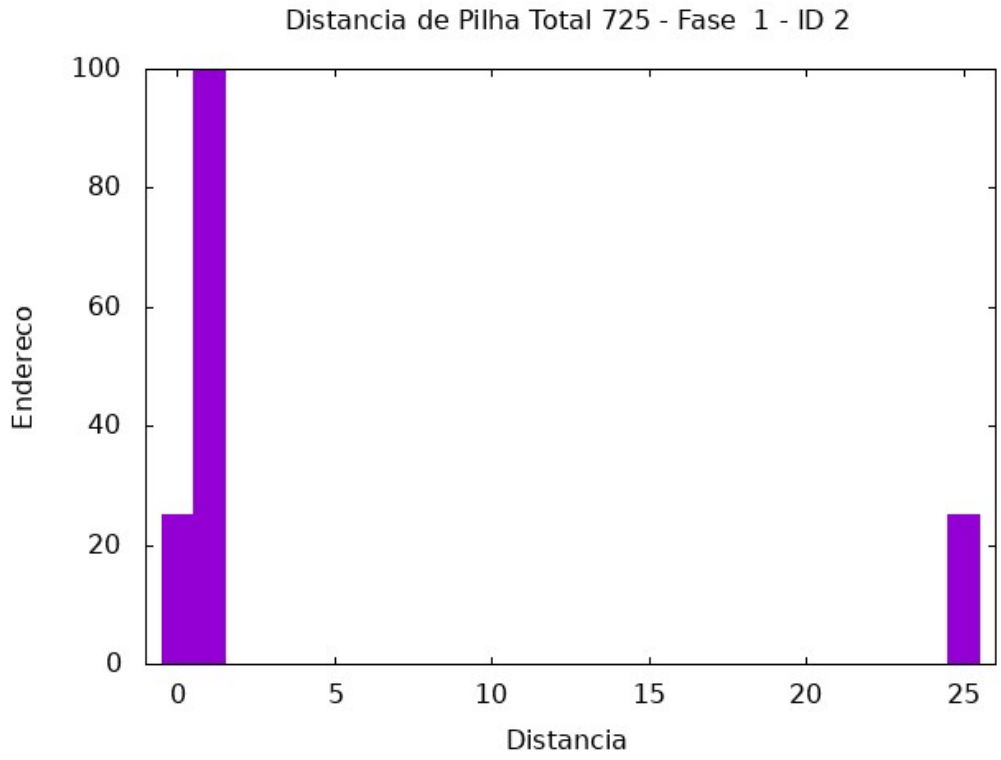

Figura 13 - Distância de pilha no uso da matriz C durante a multiplicação

Neste caso, observa-se que a distância de pilha predominante é 1 que ocorre para o processo incremental utilizado na computação dos valores de cada elemento da matriz. Este processo gera uma distância de pilha total de 725.

Na Fase 2, de apresentação dos resultados, a matriz C é acessada uma vez pela função *acessaMatriz* para permitir a computação da distância de pilha o que gera o acesso com distância 0 no gráfico mostrado na Figura 14. Depois a matriz é acessada para a apresentação dos resultados, gerando uma distância de pilha total de 625.

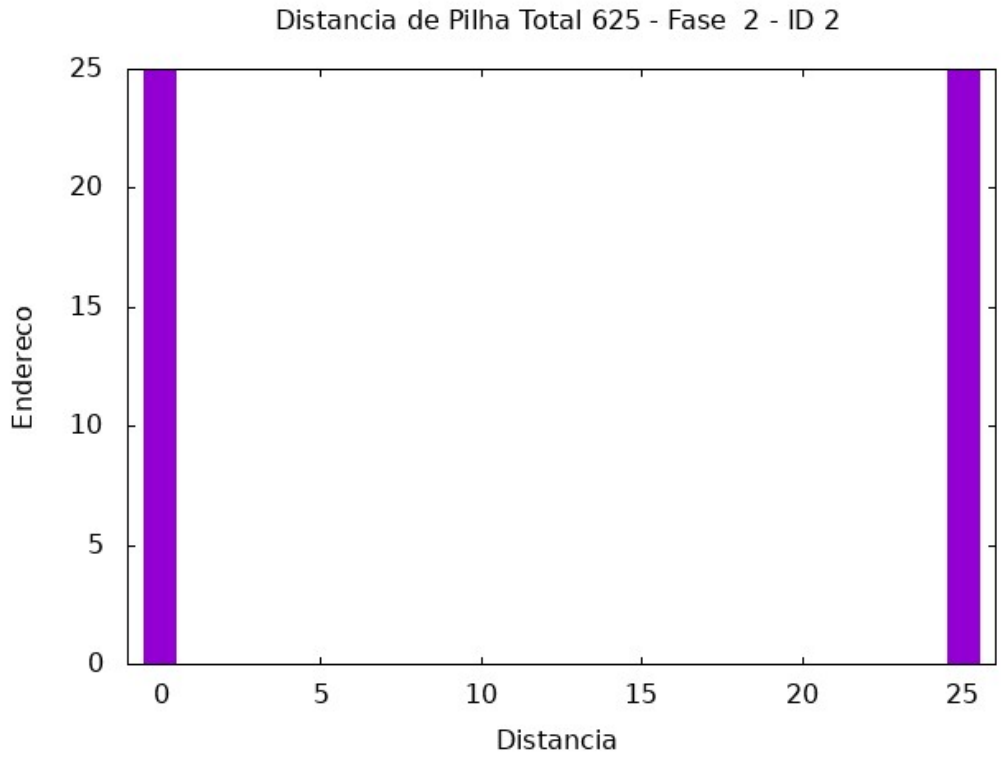

Figura 14 - Distância de pilha no uso da matriz C durante a impressão dos resultados

Soma

Como a operação de soma é um processo de acesso a cada elemento de cada matriz, a distância de pilha para todas as matrizes é a mesma, como pode ser observado nos gráficos apresentados nas Figuras 15 a 18. Para cada matriz é apresentado o histograma contendo a distância de pilha no processo de adição correspondendo à Fase 1 e para a matriz C também é apresentado, na Figura 18, a distância de pilha durante a fase de apresentação dos resultados.

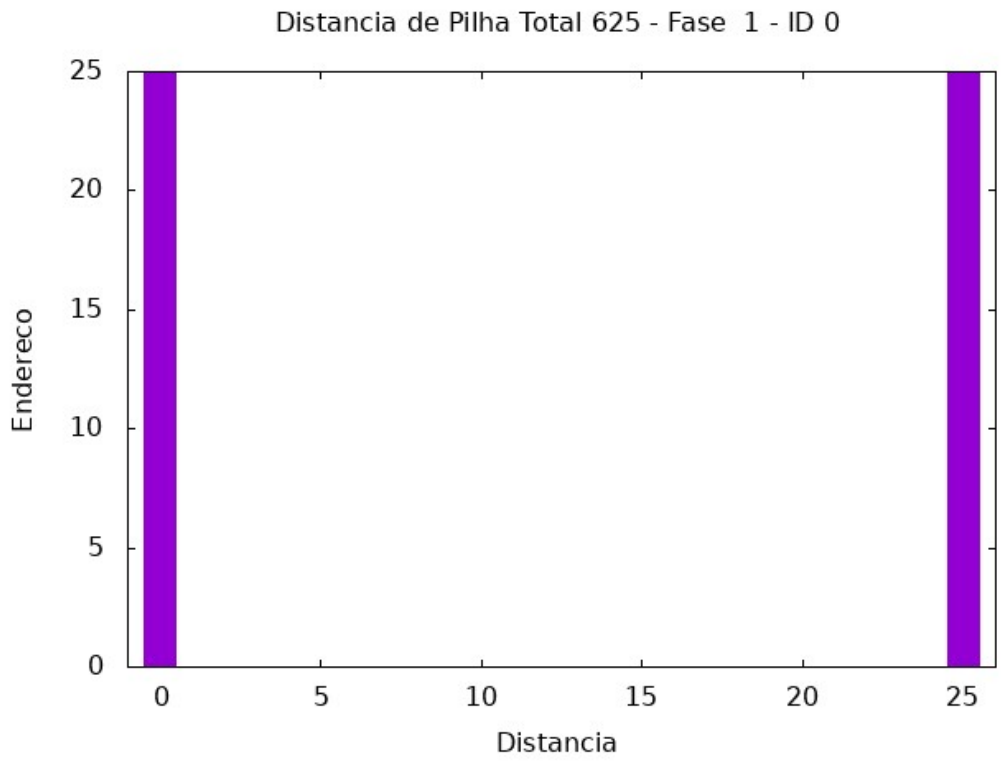

Figura 15 - Distância de pilha no uso da matriz A durante a adição

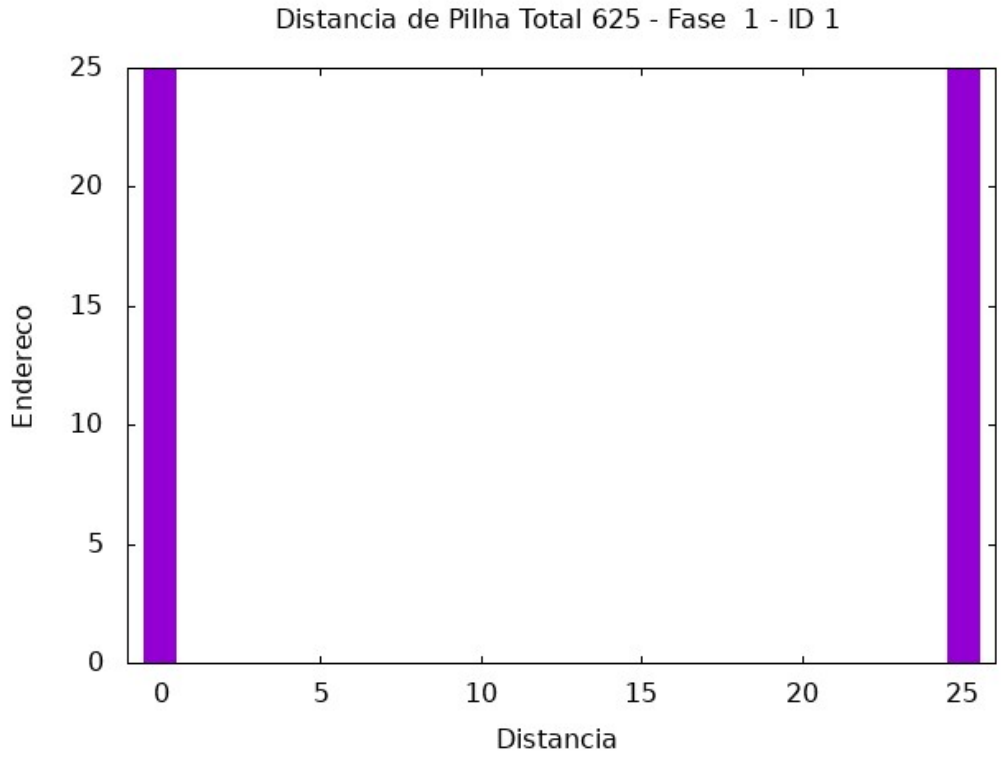

Figura 16 - Distância de pilha no uso da matriz B durante a adição

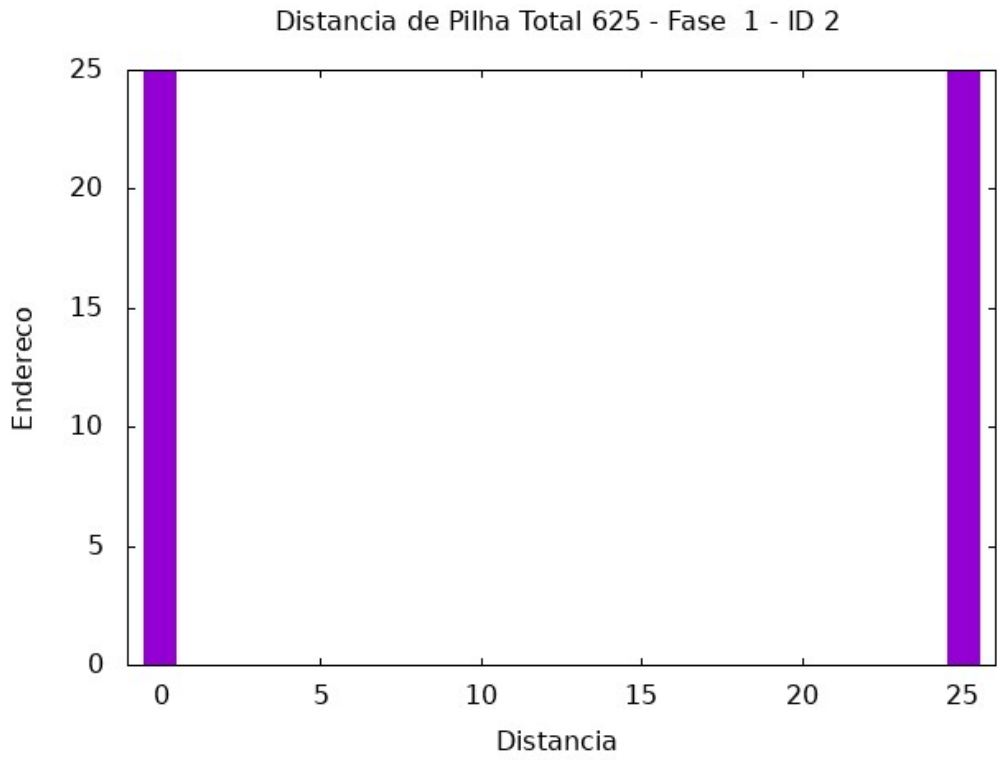

Figura 17 - Distância de pilha no uso da matriz C durante a adição

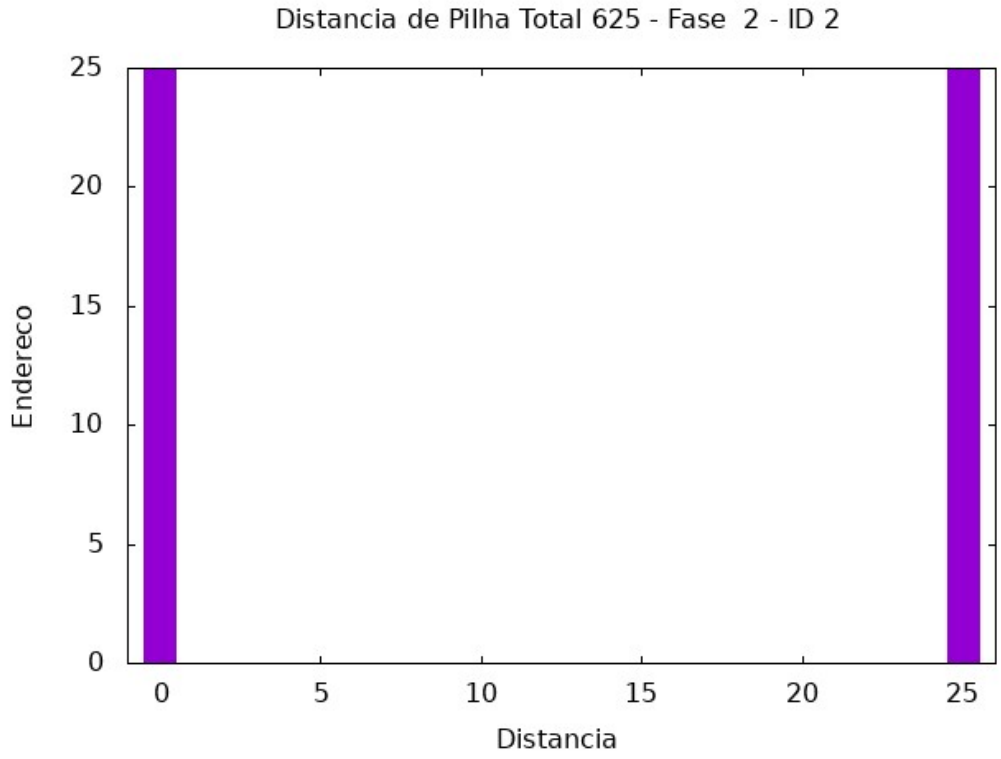

Figura 18 - Distância de pilha no uso da matriz C durante a apresentação dos resultados

Transposição

Na operação de transposição da matriz A (ID = 0) a distância de pilha é mostrada na Figura 19. Neste gráfico é possível notar que o acesso à matriz apresenta uma distância de pilha total de 625, sendo feito percorrendo as colunas e depois as linhas.

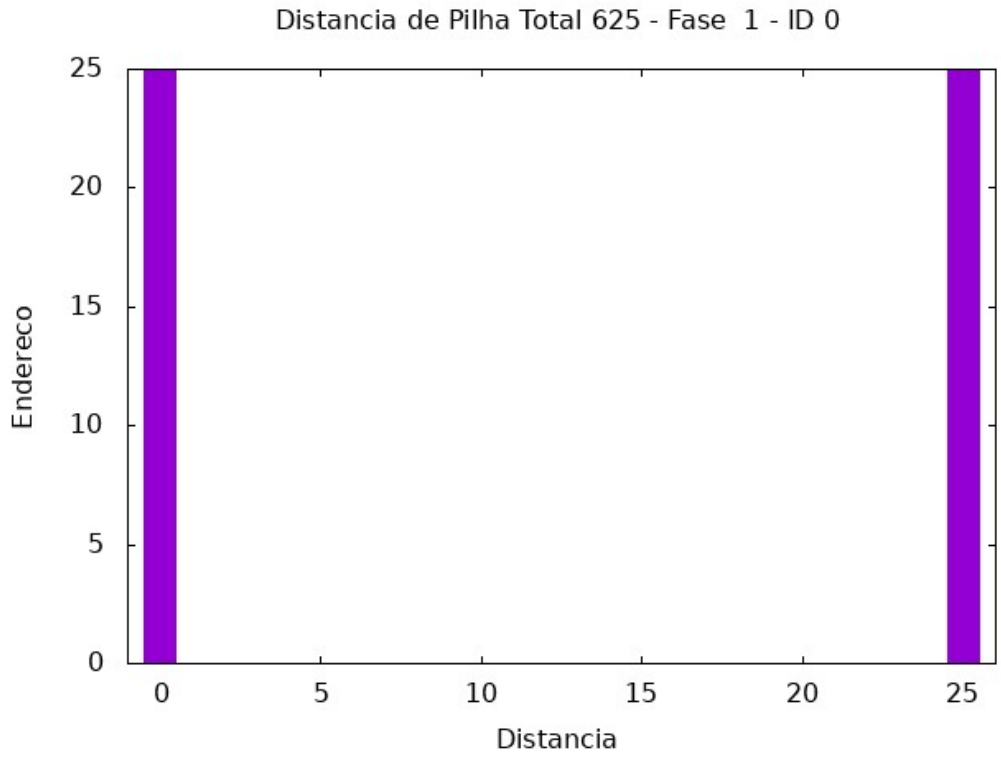

Figura 19 - Distância de pilha no uso da matriz A durante a transposição

A Figura 20 mostra a distância de pilha para a matriz B. Na abordagem utilizada, a matriz B (ID = 1) é criada para receber os valores correspondentes a transposta de A e passa a ser referenciada para a apresentação do resultado. Neste caso, a matriz é percorrida primeiramente as linhas e depois as colunas gerando uma distância de pilha total de 525.

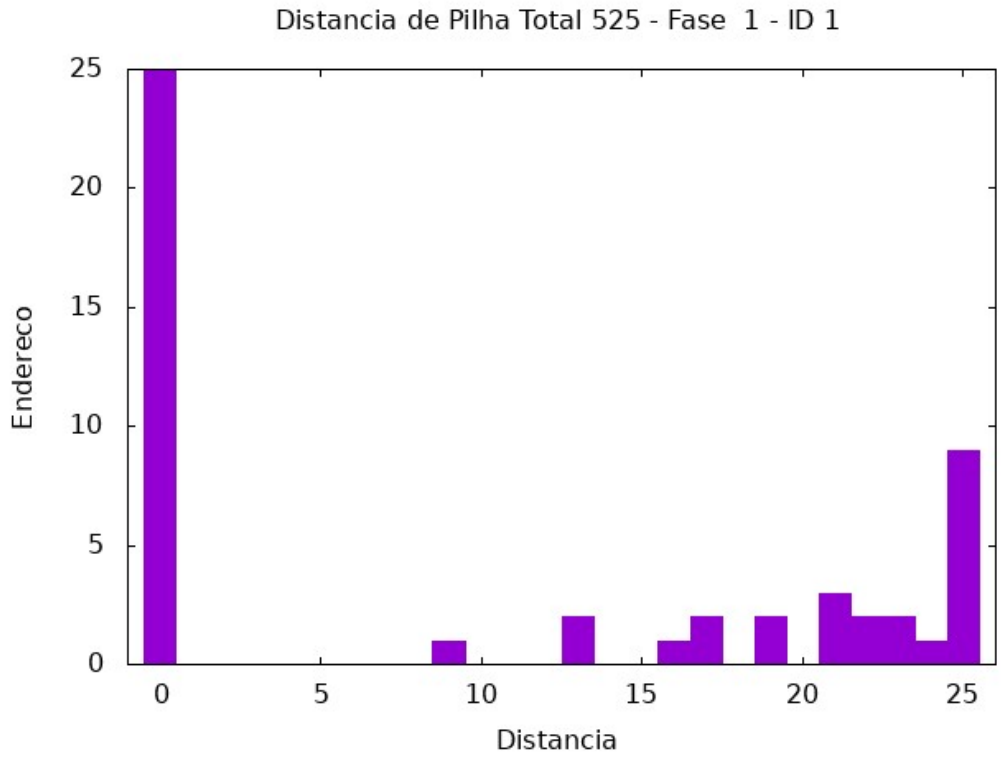

Figura 20 - Distância de pilha no uso da matriz B durante a transposição

Na Fase 2 de apresentação dos resultados, a distância de pilha é mostrada na Figura 21. Neste caso a matriz é percorrida primeiramente pelas colunas e depois as linhas gerando uma distância de pilha total de 625.

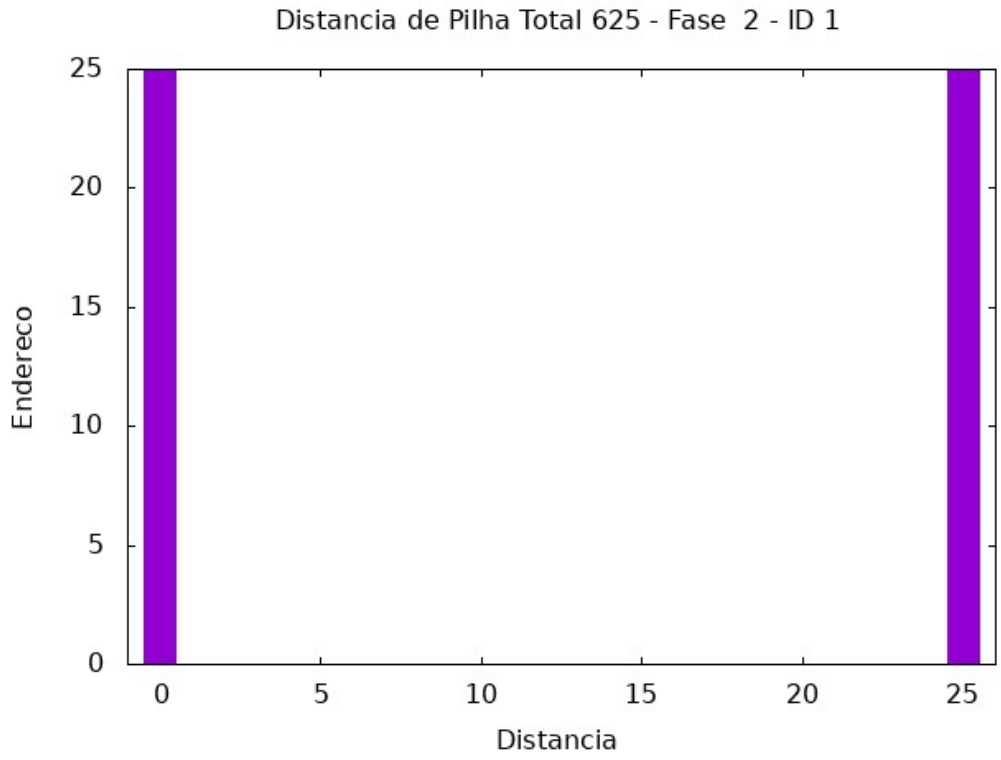

Figura 21 - Distância de pilha no uso da matriz B durante a apresentação dos resultados

6. Conclusão

Neste trabalho foi desenvolvido um programa capaz de executar as três operações - multiplicação, soma e transposição - em matrizes bidimensionais alocadas dinamicamente. Foram implementadas funções para ler as matrizes a partir de arquivos de texto e também para escrever os resultados em arquivos de texto. As opções de execução são feitas por meio de opções de linha de comando, ficando a critério do usuário optar por passar as matrizes em arquivos de texto ou passar as dimensões e deixar o programa criar matrizes aleatórias.

O objetivo do trabalho, que era recordar conceitos fundamentais do desenvolvimento de programas em linguagem C/C++, tipos abstratos de dados e de desempenho e robustez foram alcançados. Além disso, foi possível avaliar e comprovar por meio dos testes de desempenho o comportamento assintótico das operações implementadas. Outro ponto importante que foi aprendido com o desenvolvimento deste trabalho foi o comportamento do programa quanto ao acesso à memória que pode ser visualizado por meio do mapa de acesso à memória e pelo histograma de distância de pilha.

Para um trabalho futuro, seria interessante fazer uma análise da operação de multiplicação quanto à localidade de referência utilizando o produto externo como estratégia de multiplicação de matrizes e verificar se é uma abordagem mais vantajosa.

Bibliografia

[1] Pappa, Gisele L., Meira Jr., Wagner. Slides virtuais da disciplina de Estrutura de Dados 2021/2. Disponibilizado via Moodle. DCC. Universidade Federal de Minas Gerais

[2] William., Stallings (2010). Computer organization and architecture: designing for performance (8th ed.). Upper Saddle River, NJ: Prentice Hall. ISBN 9780136073734. OCLC 268788976

[3] Kecheng Ji, Ming Ling, Longxing Shi. Using the first-level cache stack distance histograms to predict multi-level LRU cache misses. Microprocessors and Microsystems, v. 55, 2017, p55-69, ISSN 0141-9331, https://doi.org/10.1016/j.micpro.2017.10.001.

Apêndice A

Instruções de compilação e execução

A.1 Compilação

O programa pode ser compilado utilizando o Makefile presente na raiz do projeto, utilizando os seguintes comandos:

make all - compila todo o projeto e executa as três operações para matrizes aleatórias registrando o acesso de memória na pasta tmp/;

make test - compila todo o projeto e executa todos os testes para as três operações com matrizes aleatórias;

make bin/matop - compila todo o projeto gerando o executável na pasta bin;

A.2 Execução

Cada operação pode ser executada separadamente utilizando a linha de comando:

Operações com matrizes lidas de arquivos:

./bin/matop <*operação*> **-p** <*log*> **-1** <*matriz A*> **-2** <*matriz B*> **-o** <*matriz C*>

onde: <*operação*> pode ser **-m** para multiplicação, **-s** para soma ou **-t** para transposição;

<*log*> é o nome do arquivo de registro de desempenho e acesso à memória;

<*matriz A*> nome do arquivo contendo a matriz A;

<*matriz B*> nome do arquivo contendo a matriz B. Caso a operação **-t** seja escolhida, esta opção não é necessária.

<matriz C> nome do arquivo onde o resultado da operação será gravado.

Operações com matriz aleatórias:

./bin/matop <*operação*> **-p** <*log*> **-x** <*linhas*> **-y** <*colunas*> **-o** <*matriz C*>

onde: <linhas> é o número de linhas para as matrizes;

<colunas> é o número de colunas para as matrizes.

Registro de acesso

Para fazer o registro de acesso à memória é necessário acrescentar a flag **-l** no comando.