Trabalho Prático: Operações com matrizes alocadas dinamicamente

Gabriel Teixeira Carvalho

1. Introdução

Esta documentação lida com o problema de implementar um Tipo Abstrato de Dados (TAD) `matriz` alocando memória dinamicamente. O objetivo desta tarefa é implementar matrizes de uma maneira mais poderosa e eficiente do que as matrizes alocadas estaticamente, que possuem um alto grau de desperdício de memória e um limite fixo de tamanho. Além disso, outro foco importante foi a revisão de conceitos de Tipos Abstratos, desempenho e robustez de código, sendo muito aplicados ao longo da implementação. Para resolver o problema citado, uma implementação de matrizes estáticas oferecida como exemplo foi adaptada para uma implementação dinâmica, com funções e métodos atualizados para o novo escopo.

A seção 2 desta documentação trata mais a fundo sobre como o programa foi implementado, enquanto na seção 3 é apresentada uma análise da complexidade temporal e espacial do trabalho. Já na seção 4, são apresentados aspectos de robustez e proteção contra erros de execução. A seção 5 destrincha o comportamento do programa em termos de tempo e espaço através de diversos experimentos realizados após a implementação do programa. Por fim, a seção 6 conclui e sumariza tudo que será falado nesta documentação, sendo seguida por referências bibliográficas utilizadas na confecção do código e por um manual de compilação e execução do programa.

2. Método

2.1. Ambiente de desenvolvimento

O programa foi implementado na linguagem C, compilado pelo GCC, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL2), com um processador Intel Core i7-6500U (2.50GHz, 4 CPUs) e 8192 MB RAM.

2.2. Organização, Arquivos e Funcionamento

O código está organizado em 4 diretórios na pasta raiz:

- `obj`: Esta pasta contém os arquivos `.c` da pasta `src` traduzidos para linguagem de máquina pelo compilador GCC e com a extensão `.o` (object).
- `bin`: Contém o arquivo executável `matop`, que consiste nos arquivos .o da pasta `obj` compilados em um só arquivo pelo GCC.

- `include`: Contém os arquivos de declaração (contrato) dos Tipos Abstratos de Dados utilizados no programa, com seus atributos e métodos especificados dentro de arquivos da extensão `.h`, que serão incluídos e implementados nos arquivos de extensão `.c`:
    - `mat.h`: Define o TAD `mat_tipo` que representa uma matriz com diversos métodos importantes para a implementação dessas estruturas matemáticas, tais como criação, inicialização, impressão e as operações básicas (soma, multiplicação e transposição), além de declarar seus atributos.
    - `memlog.h`: Define um TAD que gerencia o registro dos acessos à memória e o registro do desempenho, através de métodos utilizados sempre que há leitura ou escrita em memória.
    - `msgassert.h`: Define macros que conferem condições necessárias para o funcionamento correto do programa, abortando a execução ou imprimindo um aviso caso estas condições sejam descumpridas.

- `src`: Contém as implementações dos TADs declarados nos arquivos `.h` da pasta `include` em arquivos `.c`; Nestes arquivos estão programadas as regras, a lógica e a robustez contra erros que é o alicerce de todo o programa:
    - `mat.c`: Neste arquivo está toda a implementação dos métodos definidos no arquivo `mat.h`. Métodos mais importantes:
        - `criaMatriz`: Alicerce do trabalho, faz uso de ponteiros para ponteiros para implementar matrizes bidimensionais que alocam memória dinamicamente de acordo com o número de linhas e colunas especificado. Cada matriz criada possui um identificador único para ajudar na análise experimental.
        - `imprimeMatrizNoArquivo`: Imprime num arquivo escolhido pelo usuário as dimensões e elementos da matriz especificada.
        - `somaMatrizes`: Soma duas matrizes passadas como parâmetro e coloca o resultado em uma terceira.
        - `multiplicaMatrizes`: Multiplica duas matrizes passadas como parâmetro e coloca o resultado em uma terceira
        - `transpoeMatriz`: Transpõe uma matriz passada como parâmetro.
        - `acessaMatriz`: Percorre todos os elementos da matriz, “aquecendo o cache”, permitindo uma análise mais clara da performance.
        - `destroiMatriz`: desaloca o espaço utilizado pela matriz, liberando espaço no heap e evitando vazamentos de memória.

- `memlog.c`: Neste arquivo está presente a implementação do TAD que registra os acessos de memória e desempenho de tempo. Métodos mais importantes:
  - `leMemLog`: Registra em um arquivo uma operação de leitura de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
  - `escritaMemLog`: Registra em um arquivo uma operação de escrita de um elemento realizada, com informações como tempo, endereço acessado, tamanho do elemento.
  - `defineFaseMemLog`: Separa as fases de execução do programa, fundamental para a análise experimental.

- `matop.c`: Este arquivo implementa a função principal do programa, responsável por controlar o fluxo de execução do programa, além de outras duas funções fundamentais:
  - `parseArgs`: Função que recebe da linha de comando as operações a serem realizadas e arquivos a serem utilizados para preenchimento das matrizes e para saída de resultados de operações e performance.
  - `leMatrizDoArquivo`: Função que preenche matriz a partir de dados recebidos de um arquivo especificado. Essa função foi deixada de fora do TAD `mat_tipo` para separar as operações e funcionamento das matrizes da lógica de entrada de dados.

Além disso, há um arquivo `makefile`, responsável pela compilação modularizada do programa, permitindo compilar somente os módulos necessários, ao invés de recompilar todos os arquivos a cada mudança.

2.3. Detalhes de implementação

A estrutura de dados fundamental para o trabalho, utilizada para representar as matrizes, foi um Tipo Abstrato de Dados. O objetivo de conseguir alocar matrizes dinamicamente é alcançado através de um ponteiro para ponteiro do tipo `double`. Primeiramente, é alocado no heap um espaço equivalente ao número de linhas da matriz, seguido de uma outra alocação de memória referente às colunas. Os elementos da matriz podem ser acessados normalmente usando o símbolo de dereferência de vetores, []. Outra estrutura importante é o TAD `memlog_tipo`, que guarda informações necessárias para o registro de performance.

As operações que o usuário pode executar no programa e suas opções de linha de comando são as seguintes:

`'-s'`	Soma matrizes passadas pela linha de comando
`'-m'`	Multiplica matrizes passadas pela linha de comando
`'-t'`	Transpõe matriz passada pela linha de comando

`'-1 <arquivo>'`	Define o caminho do arquivo que contém os dados da matriz 1 a ser utilizada na operação
`'-2 <arquivo>'`	Define o caminho do arquivo que contém os dados da matriz 2 a ser utilizada na operação (Opcional em caso de soma ou multiplicação)
`'-o <arquivo>'`	Define o caminho do arquivo onde será impresso o resultado da operação escolhida
`'-p <arquivo>'`	Define o caminho do arquivo onde será impresso o registro de desempenho (opcional)
`'-l'`	Define se deve ser registrados todos os acessos a memória feitos pelas funções `leMemLog` e `escreveMemLog` no arquivo de registro de desempenho.

Por fim, cabe ressaltar que todas as passagens de matrizes por parâmetro para as funções e métodos é feita por referência, passando o endereço ao invés de copiar a matriz em questão. Isto economiza grandes quantidades de memória a cada chamada de função.

3. Análise de Complexidade

Nesta seção será analisada a complexidade tanto de tempo como de espaço das funções e métodos descritos acima. Sendo M = número de linhas da matriz, P = número de colunas da matriz, e assumindo que as funções de alocar e desalocar memória sejam O(1) em complexidade de tempo:

Função `criaMatriz` – complexidade de tempo:

Essa função executa a função de alocação de espaço `malloc` M vezes – Θ(M*P) – e, para cada linha, executa `malloc` P vezes – Θ(P). Portanto, a complexidade assintótica de tempo é Θ(M*P).

Função `criaMatriz` – complexidade de espaço:

Essa função aloca memória no heap para M linhas – Θ(M) – e, para cada linha, aloca memória no heap para as colunas – Θ(P). Portanto, a complexidade assintótica de espaço é Θ(M*P).

Função `imprimeMatrizNoArquivo` - complexidade de tempo:

Essa função itera por toda a matriz e imprime cada elemento, portanto, sua complexidade de tempo é Θ(M*P).

Função `imprimeMatrizNoArquivo` - complexidade de espaço:

Essa função aloca memória para novas variáveis e recebe uma matriz passada por referência como parâmetro. Assim, sua complexidade de espaço é Θ(1).

Função `somaMatrizes` - complexidade de tempo:

Essa função itera por toda a matriz A – Θ(M*P) – e por toda a matriz B – Θ(M*P) – de mesmas dimensões, somando cada valor na matriz de resultado C, em tempo Θ(M*P). Portanto, sua complexidade de tempo é Θ(M*P).

Função `somaMatrizes` - complexidade de espaço:

Nenhuma memória adicional é alocada durante a execução dessa função e todas as matrizes são passadas por referência, portanto, sua complexidade de espaço é Θ(1).

Função `multiplicaMatrizes` - complexidade de tempo:

Essa função multiplica o i-ésimo elemento de cada linha da matriz A com o i-ésimo elemento de cada coluna da matriz B e soma o valor resultante ao elemento correspondente da matriz C. Seja K = número de colunas de A = número de linhas de B, então, para cada linha e para cada coluna, essa função realiza K multiplicações. Portanto, sua complexidade é Θ(M*P*K).

Função `multiplicaMatrizes` - complexidade de espaço:

Nenhuma memória adicional é alocada durante a execução dessa função e todas as matrizes são passadas por referência, portanto, sua complexidade de espaço é Θ(1).

Função `transpoeMatriz` - complexidade de tempo:

Essa função cria uma matriz auxiliar – Θ(M*P) –, itera em cada elemento da matriz original para transpô-la – Θ(M*P) – e, por fim, copia os elementos da matriz auxiliar para a original – Θ(M*P). Portanto, sua complexidade é Θ(M*P).

Função `transpoeMatriz` - complexidade de espaço:

Essa função cria uma matriz auxiliar com as dimensões da matriz a ser transposta, mas trocadas. Portanto, sua complexidade de tempo é Θ(M*P).

Função `acessaMatriz` - complexidade de tempo:

Essa função acessa cada elemento da matriz exatamente uma vez, portanto, sua complexidade de tempo é Θ(M*P).

Função `acessaMatriz` - complexidade de espaço:

Essa função requer uma quantidade constante de memória independentemente da entrada e a matriz é passada por referência como argumento, logo, é Θ(1) em espaço.

Função `destroiMatriz` - complexidade de tempo:

Essa função libera o espaço alocado em cada linha em tempo Θ(M), portanto, sua complexidade de tempo é $\Theta(M^2)$.

Função `destroiMatriz` - complexidade de espaço:

Essa função requer uma quantidade constante de memória independentemente da entrada e a matriz é passada por referência como argumento, logo, é Θ(1) em espaço.

Função `leMemLog` - complexidade de tempo:

Essa função realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Função `leMemLog` - complexidade de espaço:

Essa função requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Função `escritaMemLog` - complexidade de tempo:

Essa função realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Função `escritaMemLog` - complexidade de espaço:

Essa função requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Função `defineFaseMemLog` - complexidade de tempo:

Essa função realiza uma quantidade constante de comandos independentemente da entrada, logo, sua complexidade assintótica de tempo é Θ(1).

Função `defineFaseMemLog` - complexidade de espaço:

Essa função requer uma quantidade constante de memória independentemente da entrada, logo, sua complexidade assintótica de espaço é Θ(1).

Função `parseArgs` - complexidade de tempo:

Essa função sempre precisa copiar o nome da matriz 1 passado pela linha de comando para uma variável caractere por caractere, porém, como a variável tem um tamanho máximo de 100 caracteres, a complexidade assintótica de tempo dessa função é limitada a 100 iterações antes de gerar algum erro caso o nome seja grande demais. Assim, complexidade é O(K), K = tamanho do nome do arquivo e K < 100.

Função `parseArgs` - complexidade de espaço:

Essa função requer uma quantidade constante de memória independentemente da entrada, é Θ(1) em espaço.

Função `leMatrizDoArquivo` - complexidade de tempo:

Essa função chama a função `criaMatriz` – Θ(M*P) – e precisa acessar cada elemento da matriz passada como parâmetro para preenchê-la, portanto, sua complexidade é Θ(M*P).

Função `leMatrizDoArquivo` - complexidade de espaço:

Essa função chama a função `criaMatriz`, que tem complexidade de espaço Θ(M*P), portanto, essa função também possui complexidade assintótica de espaço Θ(M*P).

Analisando as complexidades assintóticas das funções acima, percebe-se que em todas as operações, a complexidade de espaço será Θ(M*P) pois precisamos criar as matrizes, porém, a complexidade de tempo pode variar. No caso da soma e da transposição, todas as operações realizadas são O(M*P) e as próprias funções que realizam a operação desejada (`somaMatrizes` e `transpoeMatriz`) são Θ(M*P) e, portanto, essas operações possuem complexidade Θ(M*P). Já no caso da multiplicação, a complexidade de tempo é Θ(M*P*K). No caso de as matrizes serem quadradas, a complexidade de espaço é $\Theta(N^2)$ e a de tempo é $\Theta(N^2)$ (soma e transposição) ou $\Theta(N^3)$ (multiplicação).

Caso façamos uso da opção `'-l'` do programa, a qual ativa o registro de acesso, a cada operação de leitura ou escrita será realizada uma operação de impressão de complexidade O(1). Isso não muda a ordem de complexidade da operação escolhida, porém, aumenta a constante que multiplica a função de complexidade da operação. Portanto, essa opção pode reduzir bastante a performance do código.

4. Estratégias de Robustez

Em todo o código, são utilizadas as macros `avisoAssert` e `erroAssert`. Estas macros asseguram certas condições importantes para o programa e podem gerar uma mensagem de aviso no caso do `avisoAsser`t ou gerar uma mensagem e abortar a execução do programa no caso do `erroAssert`. Os mecanismos de programação defensiva e robustez utilizados foram:

- Asserção de escolha de operação – aborta o programa se nenhuma operação for escolhida pelo usuário.
- Asserção de tamanho de nome de arquivo indefinido – aborta o programa se o nome de um arquivo necessário não for definido.
- Asserção de número de operações escolhidas – emite um aviso caso sejam escolhidas mais de uma operação para o programa.
- Asserção de limite de caracteres de nome de arquivo – aborta o programa se o nome de um arquivo for grande demais.
- Asserção de abertura de arquivo – aborta o programa se o arquivo especificado não for aberto corretamente.
- Asserção de dimensão de matriz – aborta o programa se uma das dimensões passadas para uma matriz for inválida.
- Asserção de alocação de ponteiro – aborta o programa se um ponteiro não for alocado corretamente.
- Asserção de elemento inválido – aborta o programa se índices para acesso de um elemento de uma matriz forem inválidos.

- Asserção de dimensões incompatíveis para soma – aborta o programa se dimensões das matrizes a serem somadas forem inválidas com a operação.
- Asserção de dimensões incompatíveis para multiplicação – aborta o programa se dimensões das matrizes a serem multiplicadas forem inválidas com a operação.
- Asserção de destruição de matriz – emite um aviso se matriz a ser destruída já tiver sido destruída.

5. Análise Experimental

Essa seção compila uma série de experimentos de diferentes tipos que analisam a performance do programa por meio dos registros do TAD `memlog_tipo`.

5.1. Análise de tempo

Esse experimento tem como objetivo medir o tempo de execução ao performar as operações implementadas no código com diferentes tamanhos de matrizes.

O tempo é obtido ao subtrair o tempo inicial do tempo final, os quais estão presentes no arquivo de registro de desempenho. Foi feita a mesma bateria de testes para cada operação: calcula-se o tempo para realizar uma dada operação com 5 matrizes diferentes, repetindo cada procedimento 4 vezes e tirando a média dos resultados obtidos. Foram utilizadas matrizes de tamanhos 5x5, 10x10, 100x100, 250x250, 500x500 e 1000x1000, porém, os resultados das matrizes de dimensão menores que 100 foram retiradas do conjunto de dados final, pois estavam com tempos menores que 0.01s, abaixo do limite mínimo de precisão do processador.

Ao analisar os dados obtidos e traçando um gráfico com a ajuda do programa GNUPLOT, percebe-se aquilo que já foi constatado nas análises de complexidade de tempo, as operações de soma e transposição possuem complexidade parecidas ($\Theta(N^2)$, nesse caso), enquanto a operação de multiplicação tem complexidade de tempo maior, $\Theta(N^3)$, nesse caso.

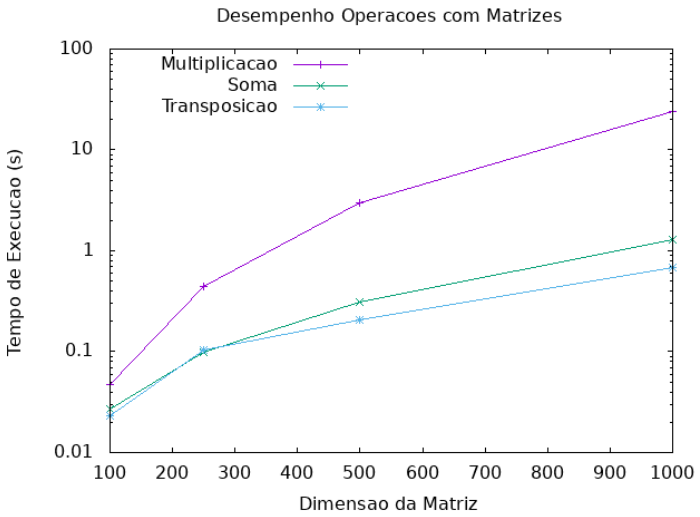

Figura 1: gráfico de desempenho de tempo

Além disso, analisando o perfil de execução do código com o programa GPROF, chega-se a outra conclusão importante: O registro de acesso (`'-l'`) pode aumentar bastante o custo para executar o código, chegando a tomar quase 50% do tempo de execução total.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.15      0.01     0.01                             multiplicaMatrizes
  0.00      0.01     0.00        2     0.00     0.00  leMatrizDoArquivo
  0.00      0.01     0.00        1     0.00     0.00  parse_args

 %         the percentage of the total running time of the
time       program used by this function.
```

Figura 2: perfil de execução do programa sem opção '-l'

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 50.08      0.06     0.06                             multiplicaMatrizes
 25.04      0.09     0.03                             leMemLog
 20.86      0.12     0.03                             clkDifMemLog
  4.17      0.12     0.01                             destroiMatriz
  0.00      0.12     0.00        2     0.00     0.00  leMatrizDoArquivo
  0.00      0.12     0.00        1     0.00     0.00  parse_args
```

Figura 3: perfil de execução do código com a opção '-l'

5.2. Localidade de referência

Outra forma de estudar a performance do programa é fazendo um Mapa de Acesso à Memória, o qual permite enxergar claramente como são feitos os acessos à memória do computador durante a execução das operações, tanto no tempo quanto no espaço.

Esse tipo de observação leva em conta os princípios de Localidade de Referência Espacial e Temporal, que dizem que um acesso ao endereço X no tempo T implica que acessos ao endereço X + dX no tempo T + dT se tornam mais prováveis à medida que dX e dT tendem a zero.

Para facilitar essa análise, as funções registradoras (`leMemLog` e `escreveMemLog`) registram uma fase e um ID em todo registro:

- Um ID identifica a qual matriz o registro se refere e é atribuído à matriz no momento de sua criação. A matriz A possui ID 0, a matriz B possui ID 1, a matriz C possui ID 2 e a matriz auxiliar criada pela função `transpoeMatriz` possui ID 3.
- Uma fase representa um intervalo de tempo no qual as operações realizadas são semelhantes, o que permite separar o registro em fases com diferentes análises. As fases são definidas através da função `defineFaseMemLog`, dentro de cada operação de matrizes. A fase 0 corresponde à leitura da matriz de um arquivo. A fase 1 corresponde ao método `acessaMatriz` e à execução

da operação. A fase 2 corresponde ao método `acessaMatriz` e à impressão da matriz resultante.

Esse estudo foi feito a partir da geração de gráficos com tempo no eixo X e endereço de memória no eixo Y, após análises dos registros gerados com a opção `'-l'` pelas 3 operações principais ao passar matrizes de tamanho 5x5 como parâmetro.

5.2.1. Soma

Na soma, em cada uma das matrizes A e B são realizadas 25 operações de escrita durante a leitura da matriz do arquivo, 25 operações de leitura por conta da função `acessaMatriz` e mais 25 operações de leitura dos elementos que serão somados em C por conta da operação de soma.

No caso da matriz C, são feitas 25 operações de escrita durante sua inicialização, 25 operações de leitura pela função `acessaMatriz` na fase 1, 25 operações de escrita dos resultado das somas dos elementos de A e B nos respectivos elementos de C, 25 operações de leitura pela função `acessaMatriz` na fase 2 e 25 operações de leitura pela impressão da matriz.

Podemos ver pelos gráficos que ilustram os registros listados acima que a operação de soma possui boa localidade de referência, já que todos seus acessos à memória são feitos em posições de endereço próximos uns aos outros numa mesma fase de execução, não havendo grandes saltos entre endereços, apenas pequenos saltos esporádicos. Isto ocorre porque a memória alocada para as linhas está em posições com endereços contíguos, com um pouco a mais de memória alocada para cada linha (isto é um mecanismo da função `malloc` que grava quão grande deve ser o pedaço de memória alocada nesse espaço extra). Em todas as funções e métodos listados, as matrizes são percorridas linha após linha e, dessa forma, os endereços acessados estão muito próximos uns dos outros. Quando se troca de linha, há um pequeno salto no número do endereço, devido ao espaço extra garantido pelo `malloc`.

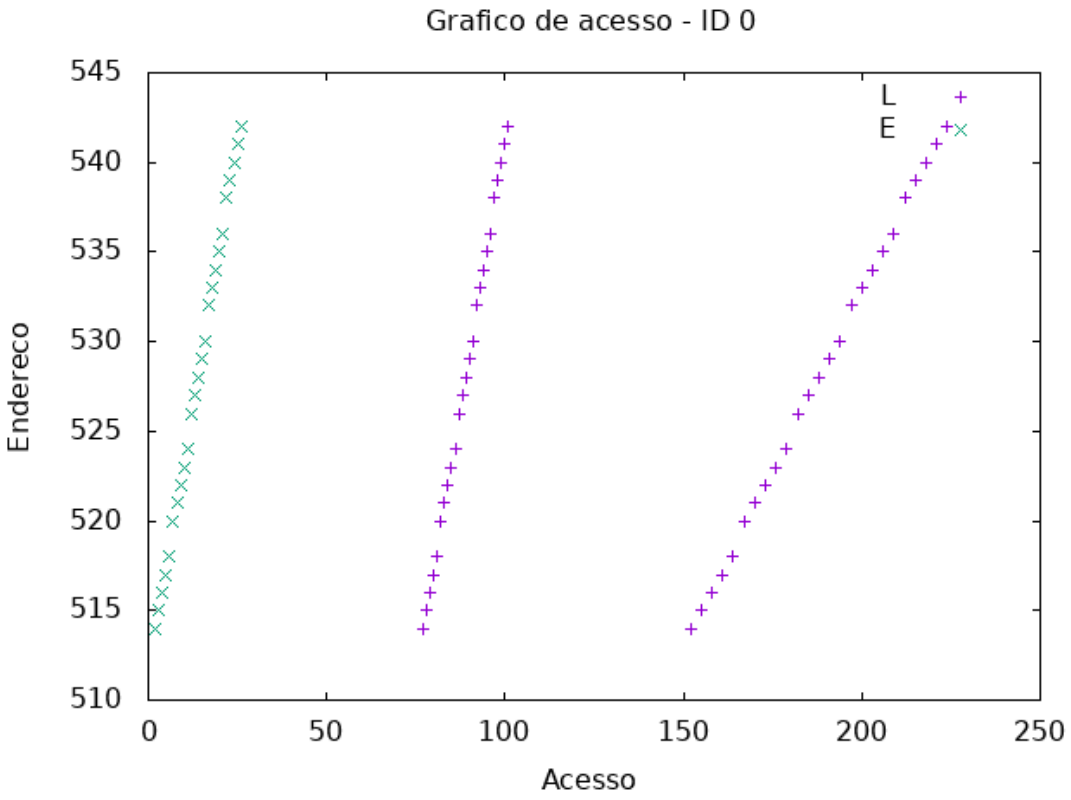

Figura 4: Mapa de acesso matriz A - Soma

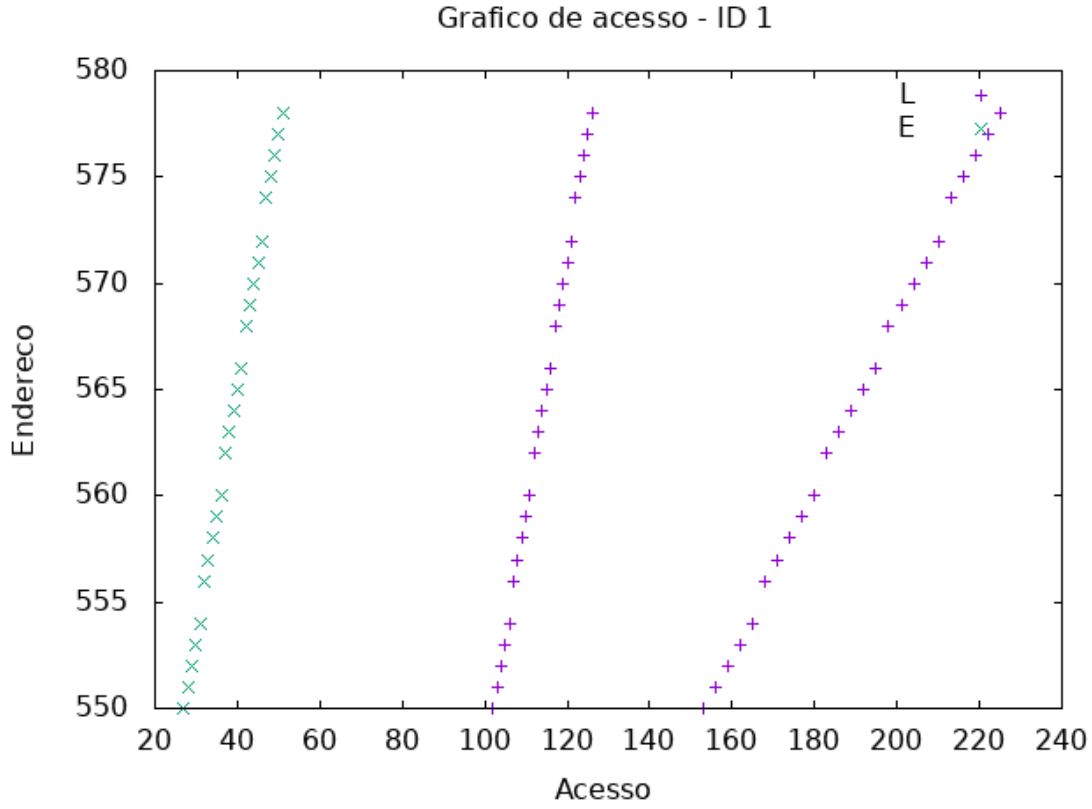

Figura 5: Mapa de acesso matriz B - Soma

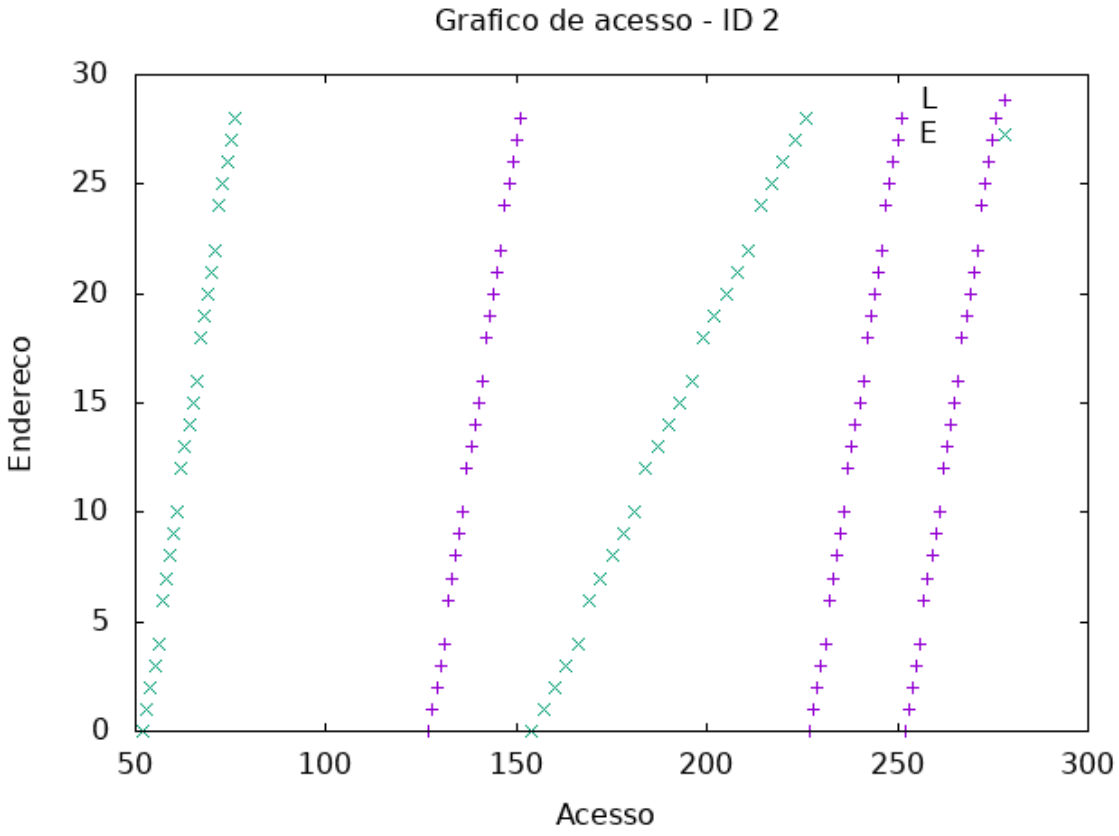

Figura 6: Mapa de acesso matriz C - Soma

5.2.2. Multiplicação

Na multiplicação, em cada uma das matrizes A e B são realizadas 25 operações de escrita durante a leitura da matriz do arquivo, 25 operações de leitura por conta da função `acessaMatriz` e mais 125 operações de leitura para realizar a multiplicação.

No caso da matriz C, são feitas 25 operações de escrita durante sua inicialização, 25 operações de leitura pela função `acessaMatriz` na fase 1, 125 operações de escrita (5 para cada elemento de C pois são feitas repetidas somas ao fazer o produto interno entre uma linha de A e uma coluna de B), 25 operações de leitura pela função `acessaMatriz` na fase 2 e 25 operações de leitura pela impressão da matriz.

Podemos ver nos gráficos da multiplicação que as matrizes A e C possuem boas localidades de referências, pois os acessos que são feitos são todos em posições de memória próximas umas das outras, enquanto a matriz B não segue tão bem os princípios de localidade explicados acima.

Isso acontece porque os elementos são percorridos linha após linha tanto na matriz A quanto na matriz C e portanto, aproveita-se do fato da memória ser alocada em blocos contíguos de acordo com as linhas. Já na matriz B, para realizar o produto interno de uma linha de A com uma de suas colunas, é preciso percorrê-la coluna após coluna, o que gera um mapa de acesso com saltos grandes entre cada acesso.

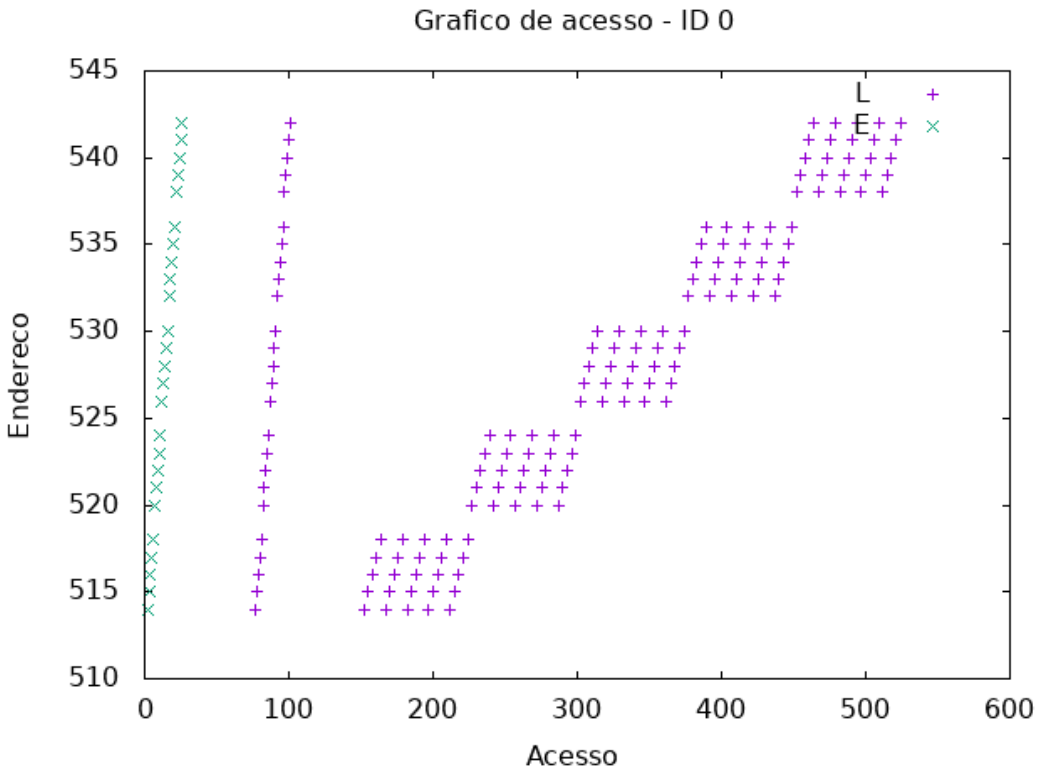

Figura 7: Mapa de acesso matriz A - Multiplicação

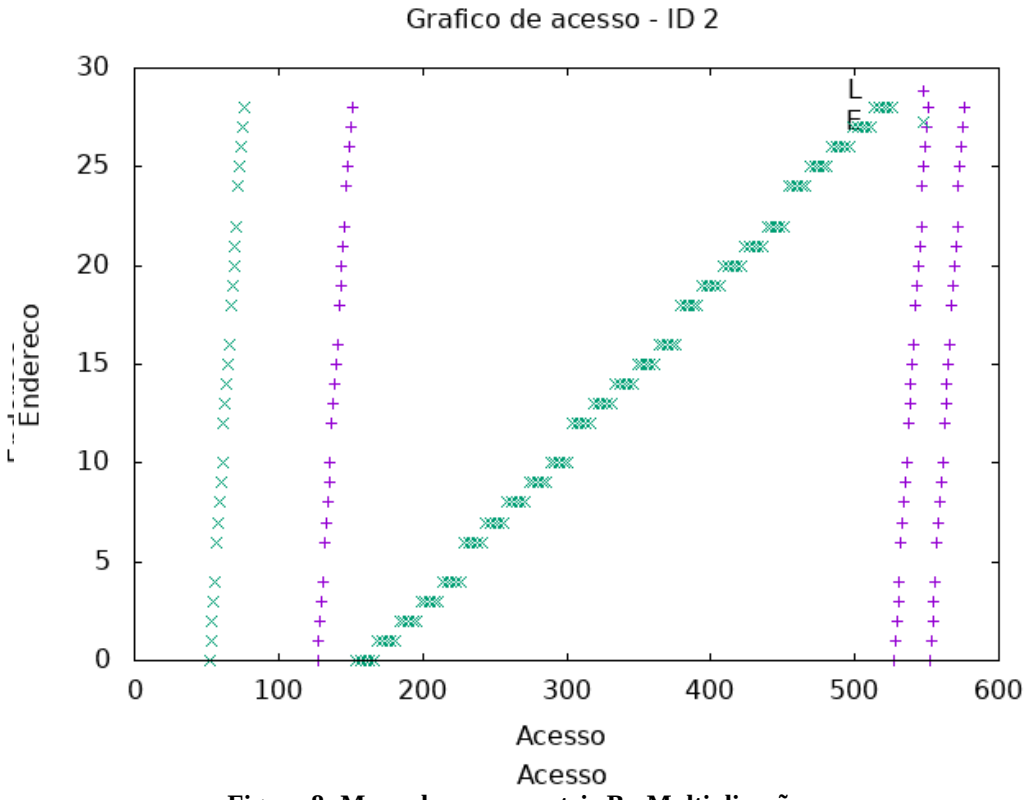

Figura 8: Mapa de acesso matriz B - Multiplicação

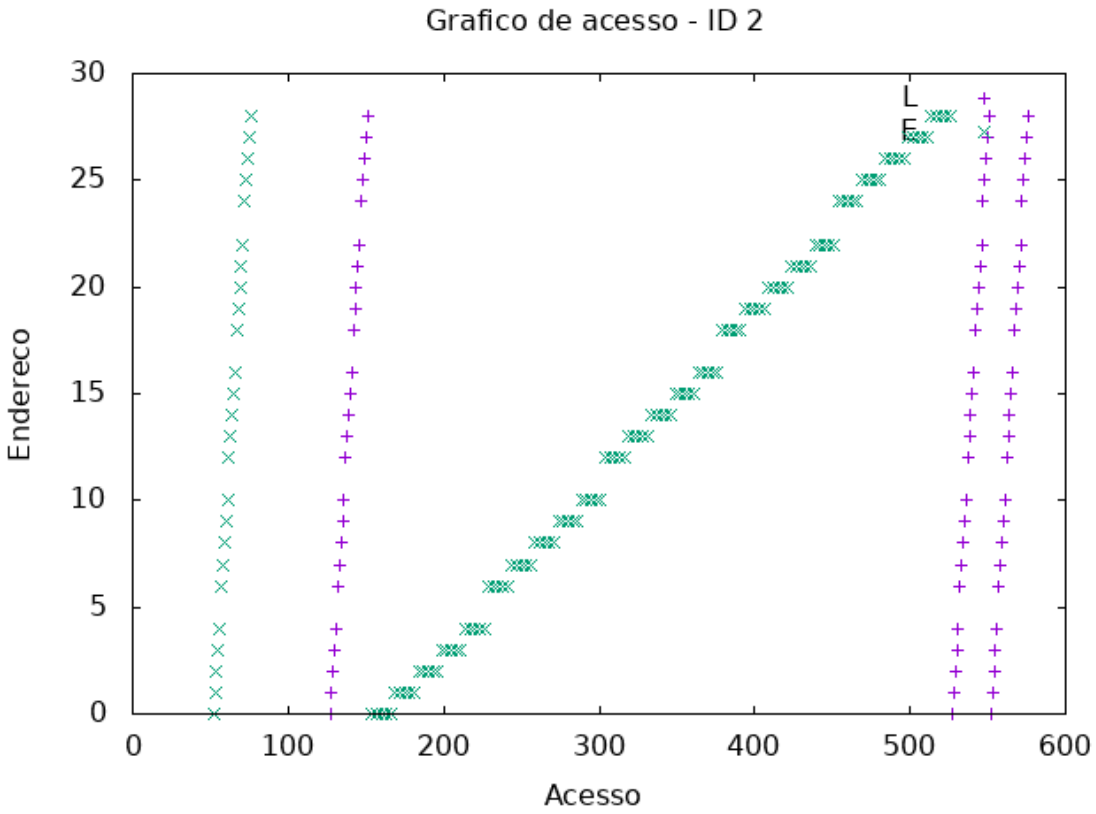

Figura 9: Mapa de acesso matriz C – Multiplicação

5.2.3. Transposição

Na transposição, na matriz A são realizadas 25 operações de escrita durante a leitura da matriz do arquivo, 25 operações de leitura pela função `acessaMatriz` na fase 1, 25 operações de leitura para criar uma matriz auxiliar com os mesmos elementos de A mas com dimensões opostas, 25 operações de escrita para reinicializar A com dimensões opostas, 25 operações de escrita para copiar os elementos da matriz auxiliar para A, 25 operações de leitura pela função `acessaMatriz` na fase 2 e 25 operações de leitura pela impressão da matriz.

No caso da matriz auxiliar, são feitas 25 operações de escrita para sua inicialização, 25 operações de escrita para a cópia dos elementos de A para a matriz auxiliar e 25 operações de leitura para a cópia dos elementos da matriz auxiliar para a matriz A.

Pelos gráficos, é possível ver que a operação possui boa localidade de referência, pois os acessos são feitos linha após linha em posições de memória próximas umas das outras, tanto na matriz A quanto na matriz auxiliar.

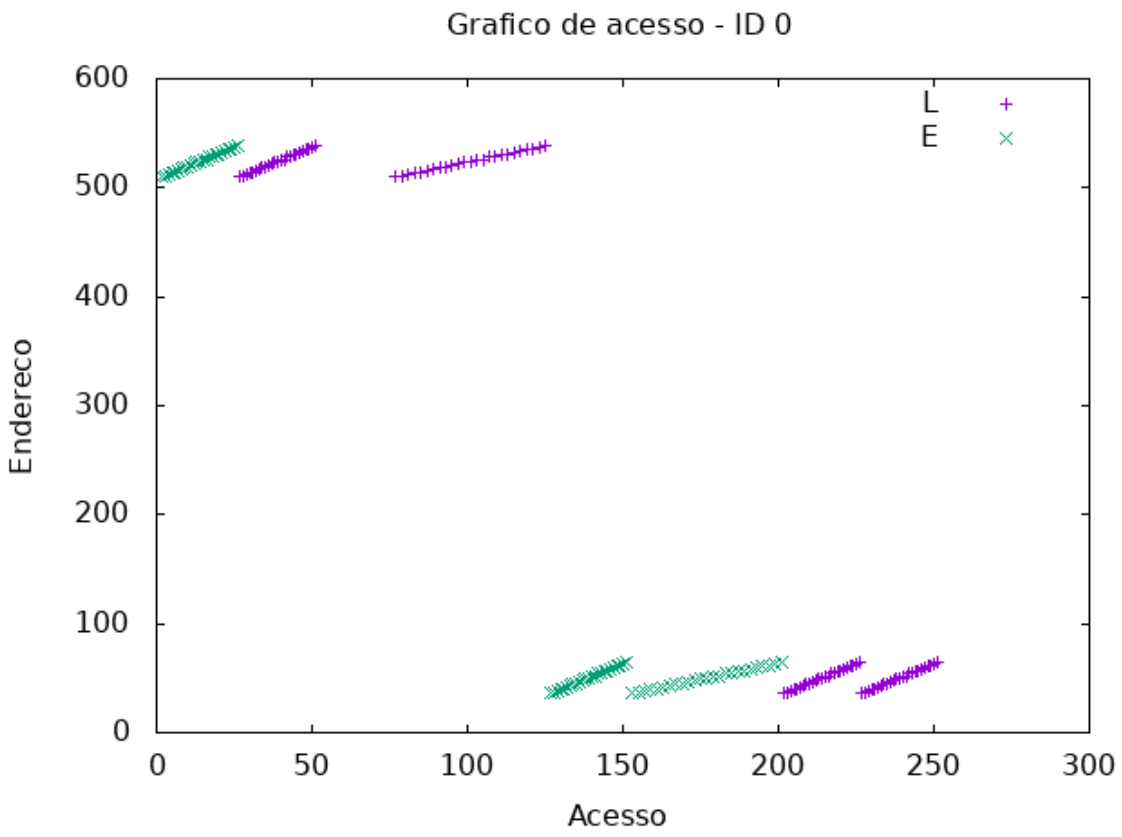

Figura 10: Mapa de acesso matriz A - Transposição

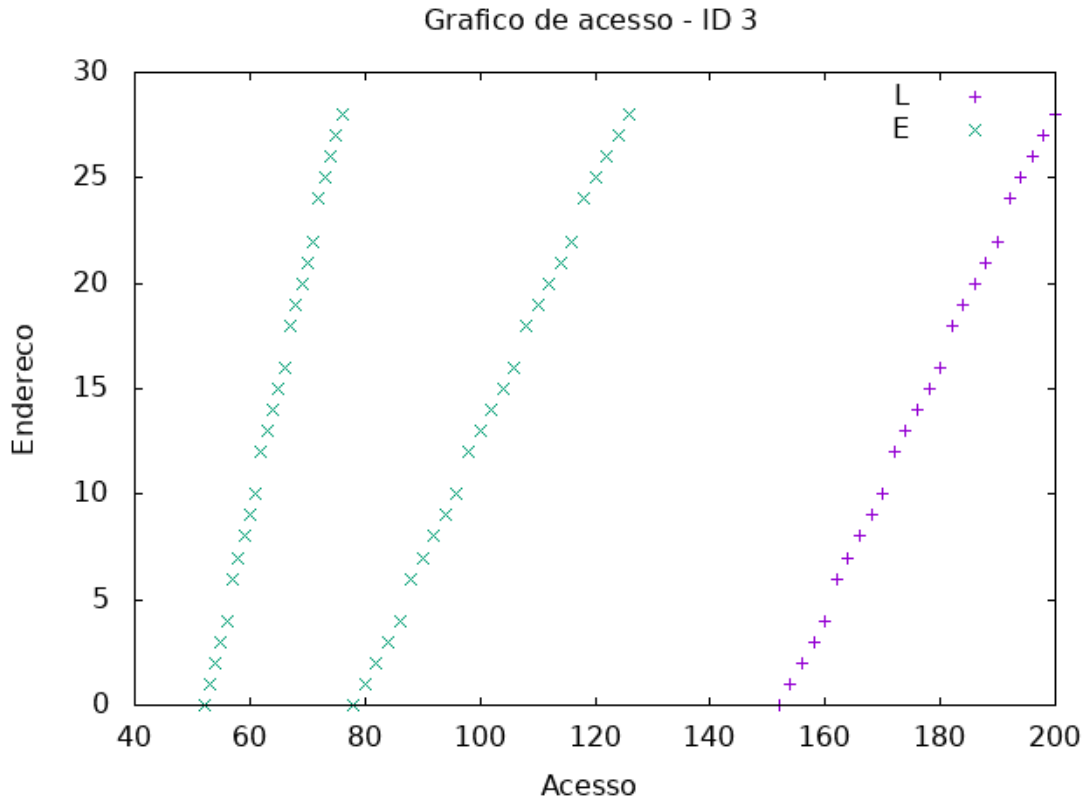

Figura 11: Mapa de acesso matriz Auxiliar – Transposição

5.3. Distância de pilha

Outra maneira de analisar a performance do programa e que corrobora com o que já foi analisado anteriormente é a distância de pilha (DP). Cada vez que um endereço é acessado, ele é colocado no topo de uma pilha inicialmente vazia. Quando o endereço é acessado novamente, sua posição na pilha é a distância de pilha.
Fazendo uma soma ponderada de todas as distâncias de pilha com suas frequências de um conjunto de procedimentos, chega-se à distância de pilha desse conjunto e quanto maior ele seja, pior é a localidade de referência da operação. Isso se dá, pois, um mesmo endereço demora mais a ser acessado novamente se sua distância de pilha é alta, ferindo o princípio da localidade de referência. Analisando esses aspectos, é possível tirar algumas conclusões a respeito das diferentes operações.

5.3.1. Soma

Na fase 1 da soma, as 3 matrizes possuem distância de pilha 625. Isto acontece porque todos os 25 elementos de cada uma são percorridos uma única vez pelo método acessaMatriz (DP 0) e depois as matrizes são lidas ou escritas para ser feita a soma. Como cada endereço só é acessado novamente depois de 25 acessos, todos os 25 endereços das matrizes possuem DP 25 (Por exemplo, o endereço do 4º elemento da matriz A é acessado pelo método acessaMatriz e só é acessado novamente depois de todos os outros 21 elementos serem percorridos e os 3 primeiros serem lidos para fazer a soma na matriz C, ou seja, sua DP será 25).
Na fase 2, o mesmo se aplica com a matriz C e a impressão.
Portanto, a soma das distâncias de pilha para a Soma é 25*0 + 25* 25 = 625.
Analisando os histogramas das distâncias de pilha da soma, percebe-se aquilo que já foi constatado nos mapas de acesso: a operação da soma possui uma boa localidade de referência, tendo uma distância de pilha relativamente baixa, igual a 625.

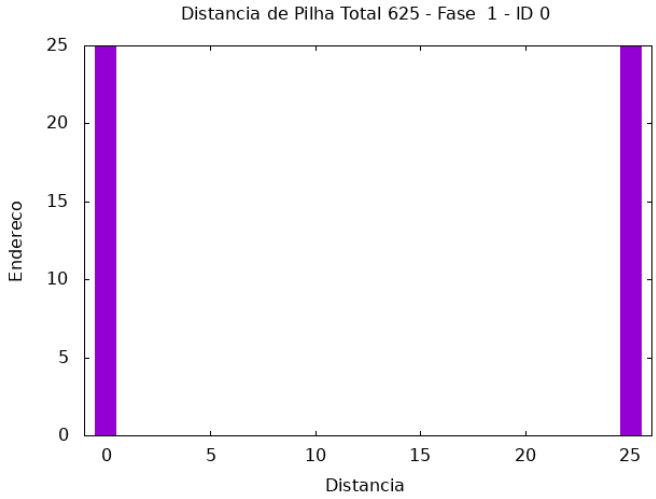

Figura 12: Histograma de distância de pilha Matriz A na fase 1 – Soma

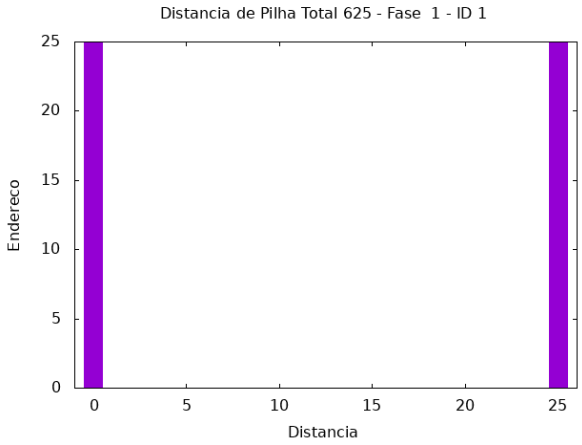

Figura 13: Histograma de distância de pilha Matriz B na fase 1 – Soma

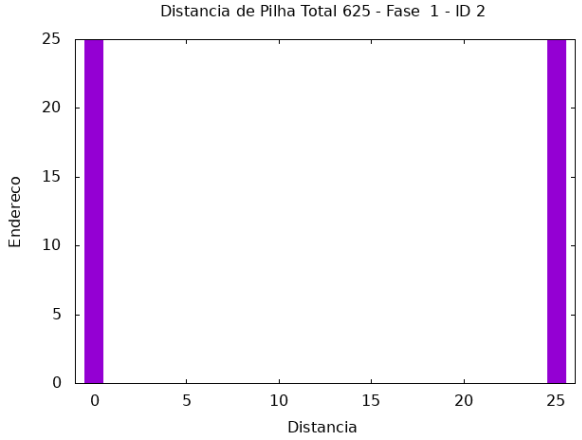

Figura 14: Histograma de distância de pilha Matriz C na fase 1 – Soma

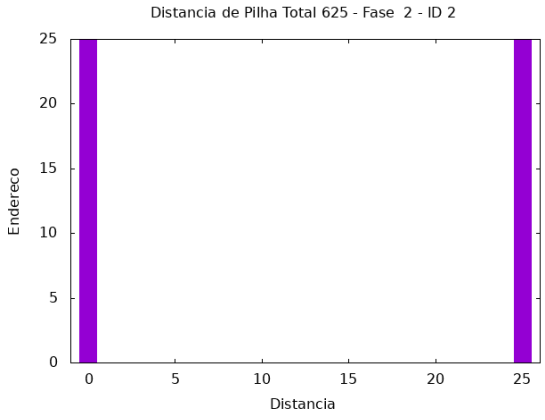

Figura 15: Histograma de distância de pilha Matriz C na fase 2 – Soma

5.3.2. Multiplicação

Na fase 1 da multiplicação, a matriz A possui DP 1125 (25* 0 do método `acessaMatriz` + 100*5 do produto interno dos 5 elementos das linhas de A com as colunas de B + 25*25 da primeira vez que cada elemento é acessado depois do acessa), a matriz B possui DP 3025 (25*0 do método `acessaMatriz` + variadas frequências e distâncias de pilha de quando troca-se de coluna + cerca de 100*25 do caminhamento pelas colunas) e a matriz C possui DP 725 (25*0 do método `acessaMatriz` + 100*1 das somas do produto interno feitas várias vezes em sequência no mesmo elemento + 25*25 da primeira vez que cada elemento é acessado depois do acessa).
Na fase 2, a mesma distância da matriz C na impressão da soma se aplica.
Analisando os histogramas das distâncias de pilha da multiplicação, torna-se claro o impacto que o caminhamento sobre as colunas tem na localidade de referência do programa, pois a matriz B tem uma distância de pilha muito maior que as demais.

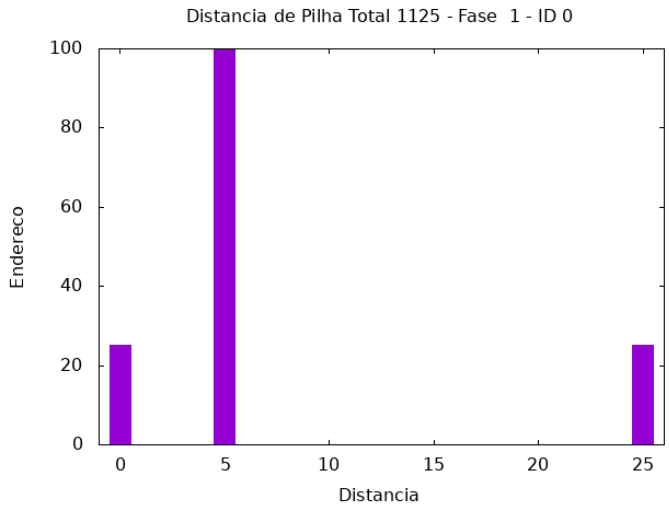

Figura 16: Histograma de distância de pilha Matriz A na fase 1 – Multiplicação

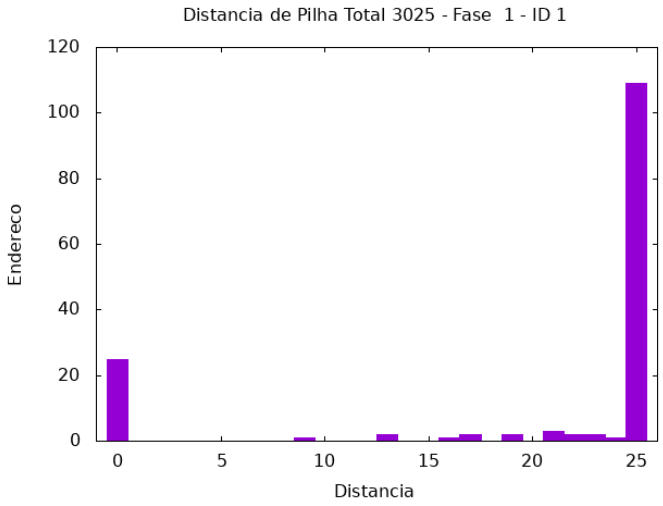

Figura 17: Histograma de distância de pilha Matriz B na fase 1 – Multiplicação

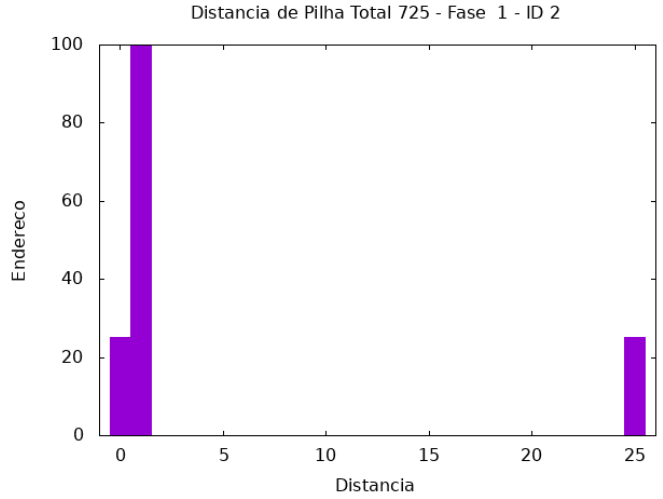

Figura 18: Histograma de distância de pilha Matriz C na fase 1 – Multiplicação

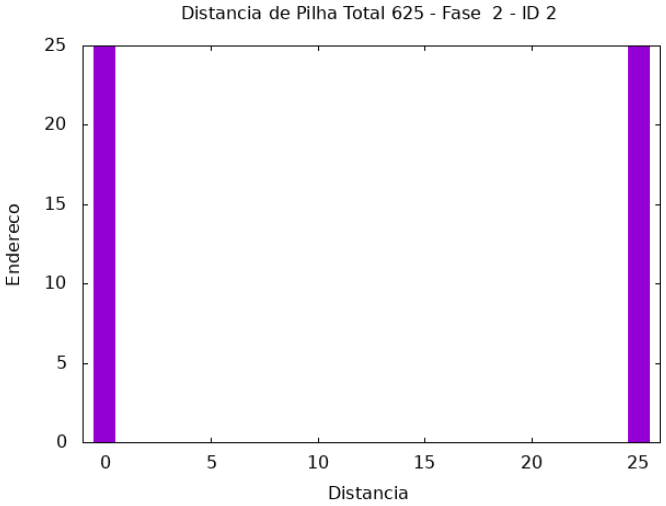

Figura 19: Histograma de distância de pilha Matriz C na fase 2 – Multiplicação

5.3.3. Transposição

Na fase 1 da transposição, a matriz A possui DP 1250 (25*0 do método `acessaMatriz` + 25*25 da cópia dos elementos para a matriz auxiliar + 25*0 da inicialização da nova matriz A em um conjunto de endereços diferente + 25*25 da escrita dos elementos da matriz auxiliar na nova matriz A) e a matriz auxiliar possui DP 1250 (25*0 do método da sua inicialização + 25*25 da cópia dos elementos de A para a matriz auxiliar + 25*25 da cópia dos elementos da matriz auxiliar para a nova matriz A).
Na fase 2, a mesma distância da matriz C na impressão da soma se aplica à matriz A. Analisando os histogramas das distâncias de pilha da transposição, percebe-se uma distância de pilha mais próxima à da soma, relativamente baixa comparada à multiplicação.

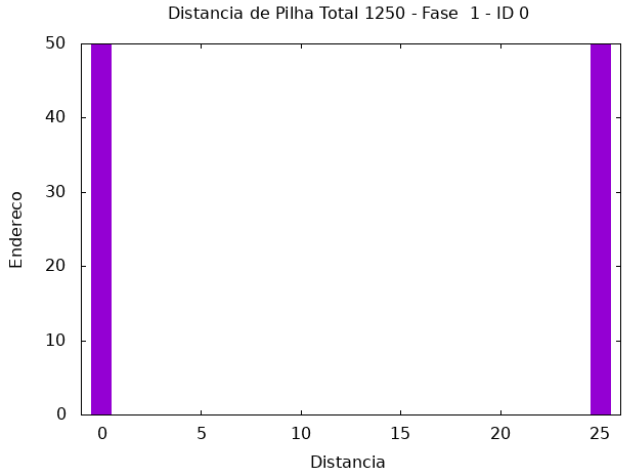

Figura 20: Histograma de distância de pilha Matriz A na fase 1 – Transposição

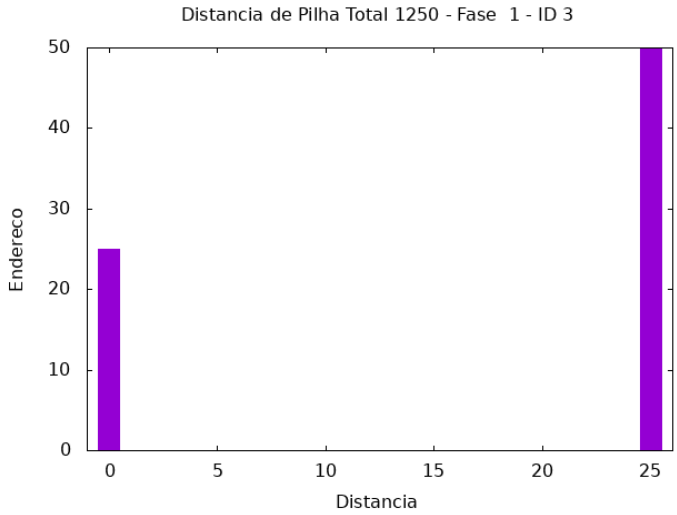

Figura 21: Histograma de distância de pilha Matriz auxiliar na fase 2 – Transposição

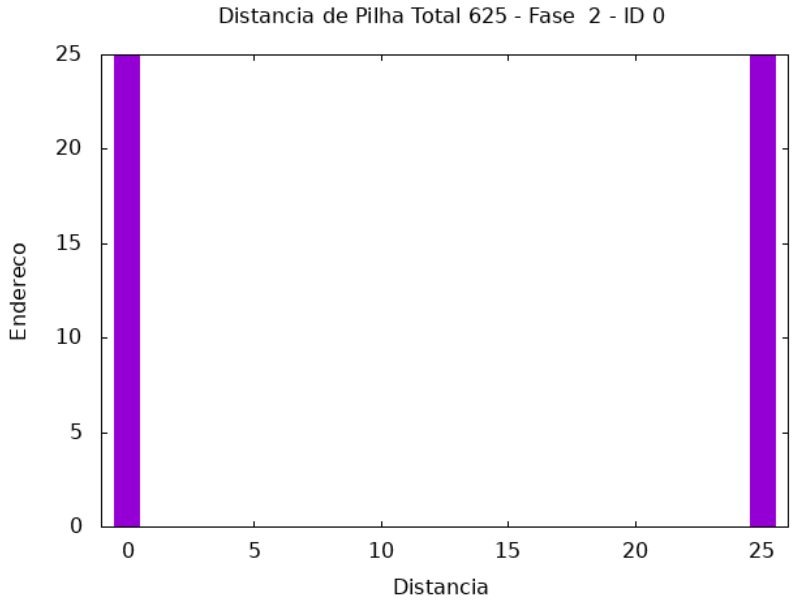

Figura 22: Histograma de distância de pilha Matriz A na fase 2 – Transposição

6. Conclusão

Este trabalho lidou com o problema de transformar uma implementação estática de matrizes em uma implementação dinâmica, na qual a abordagem utilizada para a resolução foi a aplicação de ponteiros e alocação dinâmica num tipo abstrato de dados que representa uma matriz bidimensional. Com a solução adotada, pode-se verificar que uma matriz dinamicamente alocada é mais versátil, pois não tem um limite de tamanho tão rígido quanto o da implementação estática e também é mais eficiente porque tem menos desperdícios de memória não utilizada.

Por meio da resolução desse trabalho, foi possível aplicar e revisar conceitos relacionados a alocação de memória, análise de complexidade de tempo e espaço, programação defensiva e performance de programas.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo, entender o código original, aprender a receber opções pela linha de comando, aprender a utilizar programas de traçar gráficos, análise e criação dos gráficos, organização do projeto, definição das frentes de trabalho a seguir e a criação de uma documentação completa.

7. Bibliografia

Meira, W. and Pappa, G. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

How does free know how much to free?. Stack Overflow, 2013. Disponível em: < https://stackoverflow.com/questions/1518711/how-does-free-know-how-much-to-free>. Acesso em: 16 de nov. de 2021.

Krzyzanowski, P. Processing the command line. people.cs.rutgers.edu, 2019. Disponível em: < https://people.cs.rutgers.edu/~pxk/rutgers/index.html >. Acesso em: 02 de nov. de 2021.

Implement strcpy() function in C. Techie Delight, 2021. Disponível em: < https://www.techiedelight.com/implement-strcpy-function-c/#:~:text=The%20time%20complexity%20of%20the,length%20of%20the%20source%20string.>. Acesso em: 16 de nov. de 2021.

gnuplot Tutorial 1: basic Plotting tips & tricks, errorbars, png output. MathAndPhysics, 2019. Disponível em: < https://www.youtube.com/watch?v=9QUtcfyBFhE&ab_channel=MathAndPhysics> . Acesso em: 10 de nov. de 2021.

C++ POINTERS (2020) - What is a dynamic two-dimensional array? (MULTIDIMENSIONAL dynamic arrays). CodeBeauty, 2020. Disponível em: < https://www.youtube.com/watch?v=mGl9LO-je3o&ab_channel=CodeBeauty >. Acesso em: 02 de nov. de 2021.

Instruções para compilação e Execução

1 – Extraia o arquivo `.zip` na pasta desejada.

2 – Execute o comando "`cd TP`" no terminal

3 – Execute o comando "`make all`" no terminal para compilar os módulos do programa

4 – Execute o programa `matop` da pasta `bin` com a opção desejada (existem matrizes de exemplo de diferentes dimensões na pasta `matrizes`)