UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS

Caio Eloi Campos - 2020031498

TRABALHO PRÁTICO:
OPERAÇÕES COM MATRIZES ALOCADAS DINAMICAMENTE

BELO HORIZONTE, 2022

1. INTRODUÇÃO

O trabalho prático 0 proposto pelos professores da disciplina tinham como foco a implementação de um programa de matrizes com implementação dinâmica. As matrizes são criadas partindo de valores vindo de um arquivo de texto. Na parte da implementação das matrizes, foi levado em consideração algumas operações matriciais básicas: Adição, subtração, multiplicação e transposição.

O trabalho também contará com análises experimentais com o intuito de observar o desempenho do algoritmo criado, tais como tempo e espaço utilizado. Por fim, tópicos importantes como análise de localidade e referência, e também estratégias de robustez de software são descritas no documento.

2. IMPLEMENTAÇÃO

2.2 ESPECIFICAÇÕES

O programa foi desenvolvido em suma na linguagem C + +, utilizando o compilador GNU. A máquina utilizada durante o desenvolvimento, tais como testes realizados, conta com 16Gb de memória RAM, processador Intel(R) Core(™) i7-1165G7 @ 2.80GHz 1.69GHz, rodando o sistema operacional Windows 10, porém, todo o desenvolvimento do código e seus testes foram feitos dentro do software Visual Studio Code, utilizando a máquina virtual WSL, versão 2.

2.3 ORGANIZAÇÃO DO CÓDIGO

O projeto segue todas as especificações requisitadas pelos professores da disciplina, tais como *headers* na pasta *include,* arquivos de código na pasta *src,* etc…

Primeiramente, foram utilizados poucos pacotes para implementação do código, três são padrões da linguagem C + +, que são *o "**iomanip**", o "**string**" e o "**regex**"*, que são utilizados para funções de entrada / saída de dados, de manipulação de strings e lidar com expressões regulares, respectivamente. Outro arquivo utilizado foi o *"**msgassert.h**"* que foi utilizado para realização de testes para maior robustez do código. Por último, o arquivo *"**memlog.h**",* utilizado para análises de performance.

2.4 ESTRUTURA DE DADOS

A principal estrutura de dados presente no programa é a *classe* ***mat_tipo****,* com cabeçalho na pasta "***include***" e implementação na pasta "***src****"*. Esta estrutura de dados possui dois inteiros, que representam o número de linhas e colunas da matriz, um inteiro como identificador e a matriz de fato, implementada como um *double* ponteiro de ponteiro, ex.: ***double **m***.

2.5 PRINCIPAIS FUNCIONALIDADES E PROCEDIMENTOS

Foram implementadas 11 funções relacionadas a esta estrutura, sendo elas:

```
mat_tipo(const int &tx, const int &ty);
void inicializaMatrizNula();
void imprimeMatriz() const;
mat_tipo transpoeMatriz() const;
double acessaMatriz() const;
~mat_tipo();
double getElement(const int &x, const int &y) const;
double *getAddress(const int &x, const int &y) const;
void setElement(const int &x, const int &y, const double &v);
mat_tipo operator+(const mat_tipo &mat);
mat_tipo operator*(const mat_tipo &mat);
```

Os nomes, como uma boa prática de programação, representam o que cada função realiza de fato no algoritmo. Sobre operações de matrizes, é preciso inferir que todas elas possuem verificações que garantem que as atividades podem ser realizadas, como por exemplo na multiplicação precisamos que o número de linhas da matriz "a" seja igual ao número de colunas da matriz "b". Primeiramente, temos a função *"****mat_tipo operator+****"*, que irá realizar a soma dos elementos entre duas matrizes distintas e salvar o resultado em uma terceira matriz "c". A "***mat_tipo operator****" fará por sua vez a multiplicação entre duas matrizes, também armazenando o resultado em uma matriz "c". Por último a função "***transpoeMatriz****"* que recebe uma matriz "a" e retorna sua forma transposta.

As outras funções apresentadas possuem um aspecto mais técnico de implementação e para análise de desempenho. Primeiro, a função construtora da classe "***mat_tipo****"* realiza a alocação dinâmica o elemento que representa uma matriz dentro da estrutura de dados e também atribui os valores às outras variáveis

dela. Já a função "***inicializaMatrizNula****",* tem um papel de segurança na implementação das matrizes, inicializando todas as posições com o valor 0 para evitar erros no momento em que for necessário realizar qualquer operação sob as estruturas. A função "***acessaMatriz***" tem a função de retornar dados computacionais para a análise que será discutida futuramente no projeto. A operação "***imprimeMatriz***" realiza a impressão dos valores de uma matriz na tela, colocando também os índices (linha e coluna) e um espaçamento entre eles. As funções "***setElement***", "***getElement***" e "***getAddress***" possuem uso de ler a matriz de um arquivo, pegar os elementos da matriz e registro de memória ao ler a matriz de um arquivo, respectivamente. Por último, a função destrutora "***~mat_tipo****"* que faz a desalocação das estruturas utilizando a função padrão da linguagem C + + chamada "***delete****"*.

O programa principal conta com 8 funções. Além da função main e de uma função de uso, as funções no geral ou avaliam a corretude da entrada ou são auxiliares para tal fim. Para garantir a corretude da linha de execução, foi usado o getopt da biblioteca <unistd.h>. A corretude do arquivo, por outro lado, exigiu um pouco mais de engenhosidade (que será descrita na seção apropriada). Para a linha de execução ser correta, ela deve conter pelo menos uma operação (-s ou -t ou -m), um arquivo de registro (-p <arq>), um arquivo de saída (-o <arq>) e os arquivos com as matrizes (-1 <arq> e -2 <arq>). O número mínimo de matrizes varia de acordo com a operação: para transposição apenas uma matriz é necessária, enquanto que para as outras operações é preciso fornecer ambos os argumentos. Opcionalmente podem ser usados os parâmetros -l e -h para ativar o registro de memória (necessário para a impressão) e imprimir uma mensagem de uso, respectivamente. Assumindo uma entrada correta, o fluxo é o seguinte: memlog é iniciado; matrizes são construídas; operação realizada; matriz resultante é impressa. Alguns detalhes do registro de memória foram omitidos.

3. ANÁLISE DE COMPLEXIDADE

Aqui será analisada a complexidade de cada método citado anteriormente, levando em consideração apenas custos de **TEMPO** e **ESPAÇO DE MEMÓRIA**.

3.1 ANÁLISE DOS MÉTODOS

- **construtor**: função que realiza operações constantes em tempo O(1), com laço iterado n vezes. Possui complexidade assintótica de tempo é Θ(n). Sua complexidade assintótica de espaço é Θ(1).

- ***destrutor***: mesma descrição do construtor.

- ***acessaMatriz()***: essa função realiza operações constantes em tempo O(1) e possui 2 laços aninhados, cada um itera por uma dimensão da matriz (de tamanhos m e n). Assim, sua complexidade assintótica de tempo é Θ(mn). Essa função realiza suas operações considerando estruturas auxiliares unitárias O(1), então sua complexidade assintótica de espaço é Θ(1).

- ***inicializaMatrizNula()***: essa função realiza operações constantes em tempo e possui 2 laços aninhados, cada um itera por uma dimensão da matriz, logo sua complexidade assintótica de tempo é Θ(mn). Não é declarada nenhuma estrutura auxiliar e também não são declarados parâmetros, e desse modo sua complexidade assintótica de tempo é Θ(1).

- ***transpoeMatriz()***: essa função realiza operações constantes em tempo e possui 2 laços aninhados, cada um itera por uma dimensão da matriz. Há uma chamada para a função da análise anterior também. Desse modo, sua complexidade assintótica de tempo é Θ(mn). Dessa vez, é declarada uma estrutura auxiliar de tamanho nm. Assim, a complexidade assintótica de espaço dessa função é Θ(mn).

- ***operator+()***: esse operador realiza operações constantes em tempo, possui 2 laços aninhados (cada um itera por uma dimensão das matrizes, que têm

dimensões iguais) e chama o método "***inicializaMatrizNula***". Assim, sua complexidade assintótica de tempo é Θ(mn). É declarada uma estrutura auxiliar de tamanho mn, um parâmetro é passado por referência, também de tamanho mn. Sua complexidade assintótica de espaço é Θ(mn).

- ***operator*()***: esse operador realiza operações constantes em tempo, possui 3 laços aninhados e chama o método "***inicializaMatrizNula***". Aqui os laços aninhados são um pouco mais complicados: a matriz base tem dimensões mk e a matriz do parâmetro tem dimensões kn. As operações dos loops são, então, realizadas mnk vezes. Assim, a complexidade assintótica de tempo do operador é Θ(mnk). Sua análise de complexidade de espaço é mais simples: a matriz resultante possui dimensões mn. Logo, a complexidade assintótica de espaço é Θ(mn).

- **imprimeMatriz()**: essa função realiza operações constantes de tempo e possui 2 laços aninhados, que possuem uma condicional (para não imprimir um espaço no fim de cada linha). Sua complexidade assintótica de tempo é Θ(mn), como as outras funções que iteram por 2 loops. São criadas algumas estruturas auxiliares unitárias O(1), e não há parâmetros, então sua complexidade assintótica de espaço é Θ(1)

- ***getElement()***: essa função realiza operações constantes, em tempo O(1). Sua complexidade assintótica de tempo é Θ(1). Essa função só recebe parâmetros unitários por referência, portanto sua complexidade de espaço também é Θ(1).

- ***getAddres()***: mesma análise anterior.

- **setElement():** mesma análise anterior.

4. ESTRATÉGIAS DE ROBUSTEZ

Em suma, o programa possui diversos pontos de verificação, que param o programa ao invés de corrigir os problemas. Todos eles estão usando a função "***erroAssert()***" do pacote "***msgassert.h***".

Alguns pontos principais e mais utilizados envolvem problemas de alocação, entrada de dimensões inválidas e operação inválida para as matrizes "a" e "b"

```
// inicializa as dimensoes da matriz
this->tamx = m;
this->tamy = n;
this->m = new double *[this->tamx];
erroAssert(this->m != nullptr, "Erro ao alocar matriz!");
for (int i = 0; i < this->tamx; i++)
{
    this->m[i] = new double [this->tamy];
    erroAssert(this->m[i] != nullptr, "Erro ao alocar matriz!");
}
```

```
//verificando dimensões da matriz
erroAssert(m > 0, "Dimensão Inválida");
erroAssert(n > 0, "Dimensão Inválida");
```

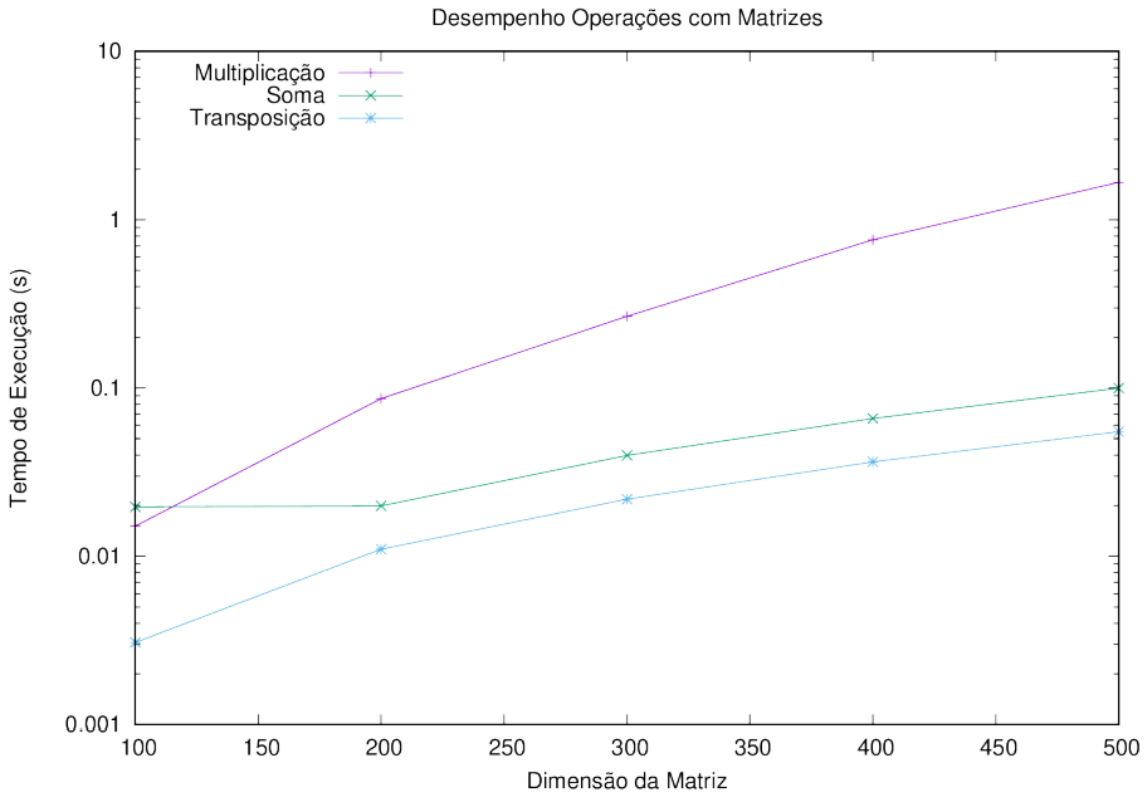
```
// Descricao: multiplica as matrizes a e b e armazena o resultado em c
// Entrada: a,b
// Saida: c
// verifica a compatibilidade das dimensoes
erroAssert(this->tamy==b.tamx,"Dimensoes incompativeis");
```

5. ANÁLISE EXPERIMENTAL

Esta etapa do trabalho consiste em analisar o desempenho computacional do algoritmo e também análise de localidade e referência, frente a vários testes diferentes.

5.1 DESEMPENHO COMPUTACIONAL

Primeiramente, foram realizados testes na máquina descrita no início do arquivo utilizando matrizes quadradas (X = Y) de tamanhos de 100 a 500, com saltos de 100 entre cada teste. O gráfico abaixo mostra o tempo (em segundos) que a máquina demora para realizar as 3 operações principais descritas com os valores impostos para teste.



Como esperado e visto no tópico sobre complexidade assintótica das funções, a função mais trabalhosa em termos de custo, é a de multiplicar matrizes, tendo um desempenho muito mais lento se comparado com as outras duas de soma e transposição.

5.2 ANÁLISE DE MEMÓRIA

Aqui é apresentado todos os testes envolvendo análise de localidade e referência, como a eficiência de acesso à memória do programa. Assim como o teste acima, foi utilizado uma matriz quadrada, mas com tamanho 5x5 e elementos sequenciais de 1 à 25, assim como apresentado nas aulas anteriormente.

O programa é dividido em 3 fases:

FASE 0: inicialização da(s) matriz(es) de entrada.

FASE 1: realização da operação em questão

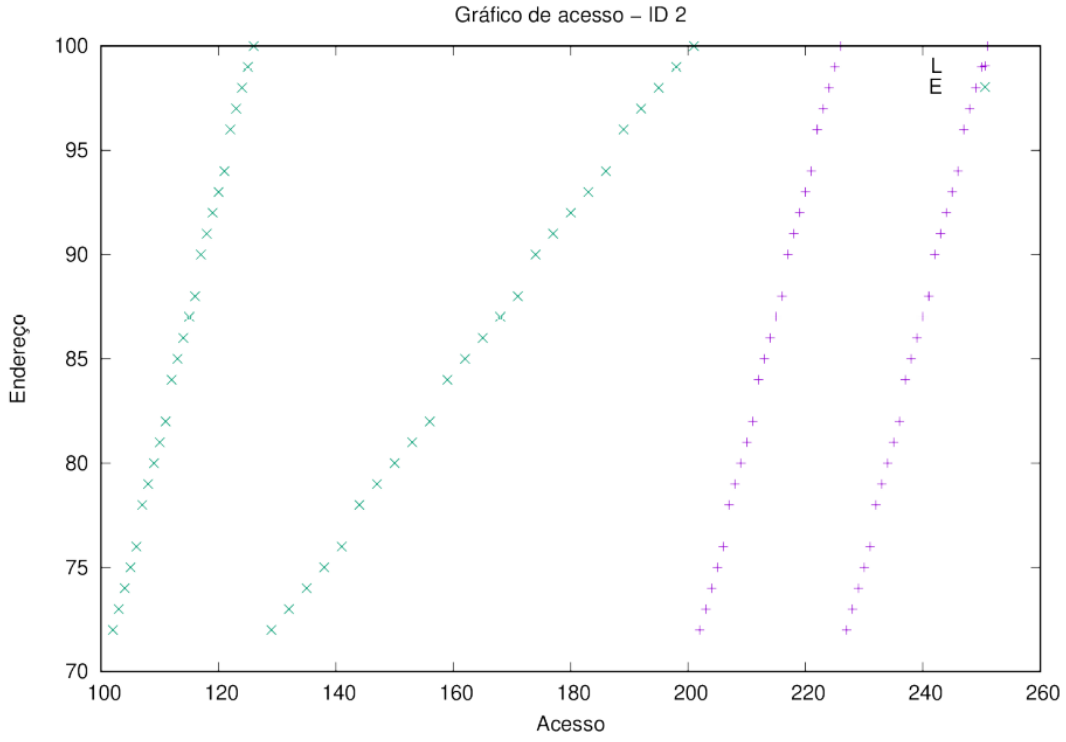
FASE 2: impressão.

No começo das fases 1 e 2, as matrizes utilizadas durante a fase são acessadas com o método "***acessaMatriz()***". Isso é importante pois é necessário ter um parâmetro para realizar a análise de pilha: sem um parâmetro não há comparação com relação ao seu último uso, tornando-a sem sentido.

Vou separar no arquivo os 3 tipos de operações e suas respectivas análises.

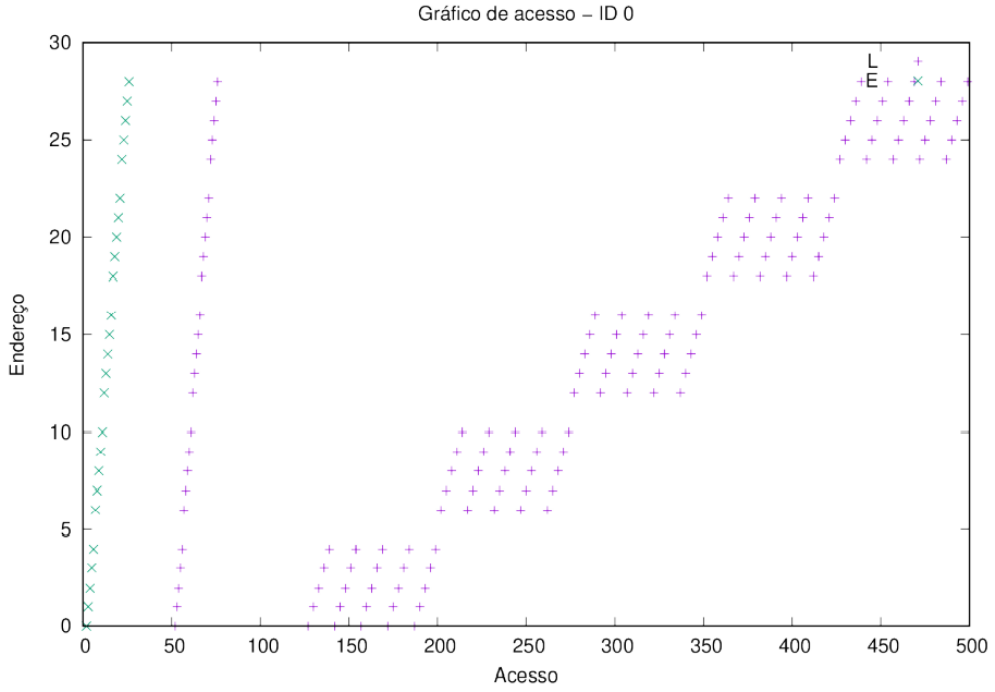
5.2.1 ADIÇÃO

Na adição, são lidas duas matrizes, e no fim resultam em uma terceira.

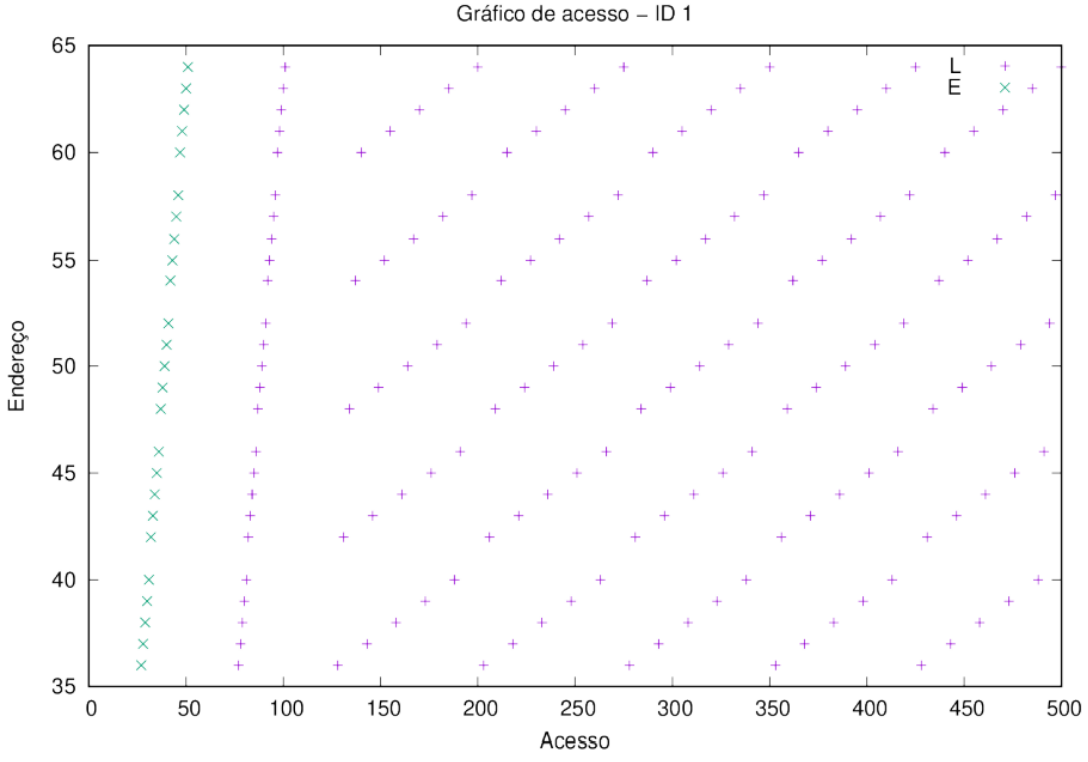


5.2.2 MULTIPLICAÇÃO

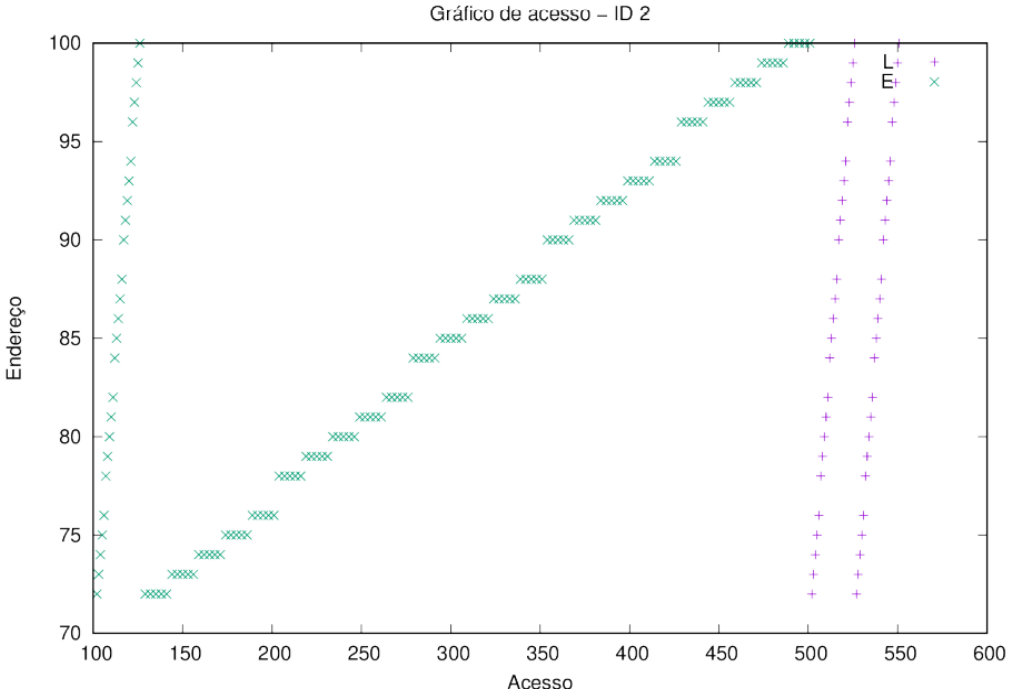
A multiplicação funciona bem diferente da adição, logo os resultados apresentados ficam nitidamente diferentes. O primeiro acesso visto abaixo é da matriz “a”:



Abaixo, é notável a ineficiência de acesso, visto que os valores acessados pela coluna, e a memória é organizada em linhas.

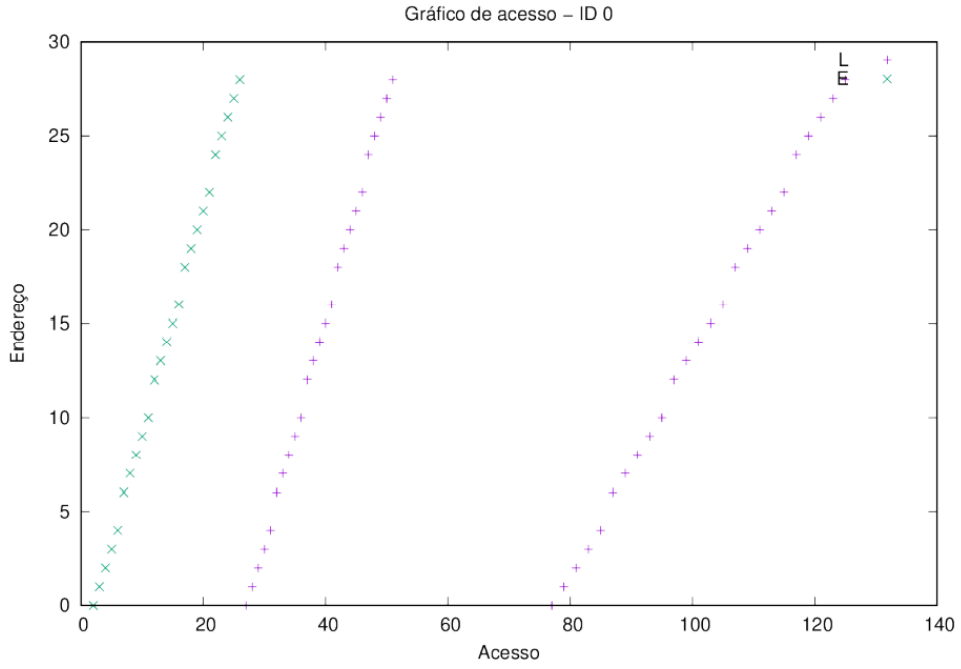


Por último, a matriz “c” resultante é acessada. É possível observar aqui primeiro a atividade de deixar nulo os valores da matriz e depois o acesso sequencial.



5.2.3 TRANSPOSIÇÃO

A transposição, por último, lê dados apenas para uma matriz, porém o resultado é salvo em uma matriz “c” e depois retornado, por isso no gráfico abaixo recebemos dois acessos similares.



6. CONCLUSÃO

O trabalho desenvolvido entregou todas as especificações pedidas pelos professores, indo de implementações de práticas de programação (código limpo, robustez, etc..) à análises de complexidade de funções, desempenho computacional e análises de memória. Foi muito interessante fazer todas as análises e entender melhor como funciona um computador ao criar um código, e como é importante sempre estar atento à complexidade dos algoritmos no desenvolvimento de um software.

7. BIBLIOGRAFIA

Wikipédia. Localidade de referência. **Wikipédia**, 2022. **Disponível em:** https://pt.wikipedia.org/wiki/Localidade_de_referência**.**
Acesso em: 2 mai, 2022.