Trabalho Prático 1
Operações com matrizes alocadas dinamicamente

Guilherme Mota Bromonschenkel Lima - 2019027571

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

guilhermebkel@ufmg.br

1. Introdução

O problema proposto foi implementar um programa que realiza algumas das operações mais populares envolvendo matrizes, como: soma, multiplicação e transposição. Com base em cada uma dessas operações, podemos avaliar seu custo computacional à medida em que variamos as dimensões das matrizes e também o padrão de acesso à memória e localidade de referência.

O programa recebe alguns argumentos no início de sua execução e com base nesses argumentos, decide qual operação executar envolvendo as matrizes que são fornecidas através de arquivos de texto de configuração.

Durante a execução das operações, fazemos uso de um módulo chamado **memlog** para fazer diversos registros de acesso de memória. Além disso, para realizar a análise experimental foi necessário utilizar a biblioteca **analisamem**, que é responsável por pegar os dados gerados pela biblioteca **memlog** e tratá-los para uma melhor visualização.

Por fim, fazemos uso também de uma ferramenta chamada **gprof** para verificar métricas de desempenho computacional, como a duração de chamadas de funções do programa.

2. Implementação

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

Os testes do programa foram realizados no sistema operacional Ubuntu (versão 18.04).

2.1. Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de um vetor de vetores do tipo double. A preferência dessa estrutura se deve ao fato de que estamos lidando com matrizes, então buscamos uma estrutura na qual possamos fazer uso da ideia de linhas e colunas de forma simples.

Essa estrutura de dados foi montada através de um tipo de dados denominado **struct**. Através desse tipo de dados, conseguimos armazenar além da estrutura de dados principal, outras informações importantes, como a quantidade de linhas e colunas da matriz.

2.2. Formato dos dados

Durante a execução do programa fazemos o uso de alguns argumentos que nos ajudam a obter os dados de entrada para a execução das regras de negócio do programa.

Nesse sentido, vale ressaltar que recebemos através de argumentos o caminho para o arquivo de definição das matrizes, para então lermos esses arquivos durante a execução do programa e criar as matrizes em memória.

Durante a execução das operações envolvendo as matrizes, salvamos a matriz resultante em um arquivo .txt que pode ter seu caminho especificado também através de argumentos que são fornecidos durante a execução do programa.

2.3. Modularização

Para garantir uma melhor qualidade de código e facilidade de manutenção, temos o programa como um todo dividido em 3 módulos com obrigações diferentes:

O módulo **mat** (é o domínio responsável por realizar as regras de negócio das operações de soma, multiplicação e transposição entre as matrizes).

O módulo **memlog** (realiza a gravação de acessos de memória durante a execução das operações das matrizes no módulo **mat**)

O módulo **matop** (recebe os argumentos de execução do programa, usa esses argumentos para realizar operações em matrizes através do módulo **mat** e gerar os dados de análise no fim de sua execução através do módulo **memlog**).

3. Análise de Complexidade

A análise de complexidade do programa foi realizada utilizando como base o módulo principal que faz todas as execuções de regras de negócio com base na operação escolhida. Nesse sentido, a análise compreende tanto o âmbito de tempo quanto de espaço.

3.1. Tempo

Antes de realizar todas as operações envolvendo as matrizes, precisamos analisar os argumentos $O(n)$, iniciar o módulo **memlog** $O(1)$, ativar ou desativar o módulo de **memlog** $O(1)$.

Ou seja, para todas as operações realizadas, temos inicialmente a seguinte complexidade: $O(n)$.

A partir daí, podemos compreender então qual é a complexidade de acordo com a operação realizada.

Para o caso de soma, precisamos definir a fase do módulo de **memlog** $O(1)$, ler a matrix do primeiro arquivo $O(n^2)$, ler a matrix do segundo arquivo $O(n^2)$, criar uma matriz resultante $O(n)$, tornar a matriz resultante em uma matriz nula $O(n^2)$, definir a fase do módulo **memlog** $O(1)$, acessar a primeira matriz $O(n^2)$, acessar a segunda matriz $O(n^2)$, acessar a matriz resultante $O(n^2)$, somar as matrizes um e dois $O(n^2)$, definir a fase do módulo **memlog** $O(1)$, acessar a matriz resultante $O(n^2)$, salvar a matriz resultante $O(n^2)$,

destruir a primeira matriz $O(1)$, destruir a segunda matriz $O(1)$ e por fim, destruir a matriz resultante $O(1)$. Sendo assim, teremos a seguinte complexidade na operação de soma: $O(n^2)$.

Para o caso de multiplicação, ocorrem as mesmas operações do caso de soma, com exceção de que a regra de negócio principal executada é uma multiplicação de complexidade $O(n^3)$ ao invés da soma que possuía complexidade $O(n^2)$. Ou seja, teremos a seguinte complexidade na operação de multiplicação: $O(n^3)$.

Por fim, para o caso de transposição, precisamos definir a fase do módulo **memlog** $O(1)$, ler a matriz do arquivo 1 $O(n^2)$, definir a fase do módulo **memlog** $O(1)$, acessar a matriz do arquivo 1 $O(n^2)$, transpor a matriz do arquivo 1 $O(n^2)$, definir a fase do módulo **memlog** $O(1)$, acessar matriz do arquivo 1 $O(n^2)$, salvar a matriz transposta $O(n^2)$, destruir a matriz $O(1)$. Sendo assim, teremos a seguinte complexidade na operação de transposição: $O(n^2)$.

Portanto, podemos concluir que nosso pior caso é na operação de multiplicação, com complexidade $O(n^3)$ e, nosso melhor caso, é na operação de soma e transposição, com complexidade $O(n^2)$.

3.2. Espaço

Como estamos lidando com matrizes e a estrutura de dados utilizada se trata de um vetor de vetores, podemos concluir que temos uma complexidade espacial $O(n^2)$. Isso pode ser constatado pelo fato de que temos um vetor de n linhas e, em cada uma dessas linhas, guardamos um vetor de n colunas.

4. Estratégia de Robustez

Podemos visualizar esse programa em duas grandes partes genéricas: entrada de dados e execução de métodos com base nos dados recebidos.

Nesse sentido, para garantirmos a robustez desse programa, foi necessário adicionar validações nessas duas partes.

A título de exemplo, temos validações para garantir que os parâmetros de entrada obrigatórios sejam fornecidos.

Além disso, quando olhamos para a execução de métodos, conseguimos compreender que temos que garantir que as regras de negócio envolvendo o âmbito de matrizes seja validado também. Por exemplo: é um fato matemático que para fazer uma soma de matrizes, a quantidade de linhas e colunas das matrizes precisa ser igual. Sendo assim, quando formos realizar a operação de soma, temos uma validação para garantir que essa operação só pode ser realizada caso essa regra de negócio seja satisfeita. O mesmo é feito para diversas outras regras de negócio envolvendo o contexto de matrizes.

5. Testes

Foi realizada uma série de testes para garantir que todas as operações estão sendo executadas corretamente. Logo abaixo é possível ver quais foram as entradas utilizadas para todas as operações:

- Entrada (Matrix 1):
5 5
1 2 3 4 5
5 7 8 1 23
1 92 28 2 15
4 5 2 9 2
5 5 3 4 1

- Entrada (Matriz 2):
5 5
5 7 82 1 2
9 86 23 2 4
12 42 58 2 1
2 5 9 1 2
9 7 59 2 4

Com as entradas acima, fomos capazes de obter as saídas esperadas.

5.1. Soma

6.000000 9.000000 85.000000 5.000000 7.000000
14.000000 93.000000 31.000000 3.000000 27.000000
13.000000 134.000000 86.000000 4.000000 16.000000
6.000000 10.000000 11.000000 10.000000 4.000000
14.000000 12.000000 62.000000 6.000000 5.000000

5.2. Multiplicação

112.000000 360.000000 633.000000 25.000000 41.000000
393.000000 1139.000000 2401.000000 82.000000 140.000000
1308.000000 9210.000000 4725.000000 273.000000 462.000000
125.000000 601.000000 758.000000 31.000000 56.000000
123.000000 618.000000 794.000000 27.000000 45.000000

5.2. Transposição

1.000000 5.000000 1.000000 4.000000 5.000000
2.000000 7.000000 92.000000 5.000000 5.000000
3.000000 8.000000 28.000000 2.000000 3.000000
4.000000 1.000000 2.000000 9.000000 4.000000
5.000000 23.000000 15.000000 2.000000 1.000000

6. Análise Experimental

Agora, podemos efetuar uma análise de desempenho computacional e eficiência de acesso à memória.

6.1. Desempenho Computacional

Para esse experimento, foram utilizadas matrizes de tamanho 100x100, 200x200, 300x300, 400x400, 500x500.

Vamos usar a métrica de tempo de execução de cada operação como métrica relevante para a avaliação do nosso algoritmo, visto que é uma métrica que nos possibilita entender o custo de cada operação na prática e fazer uma comparação com a análise de complexidade que fizemos anteriormente. Sendo assim, o tempo de cada operação (em segundos) para cada tipo de matriz pode ser visto na tabela abaixo:

Tamanho da matriz	**Soma** *(tempo de execução)*	**Multiplicação** *(tempo de execução)*	**Transposição** *(tempo de execução)*
100x100	0.005025799	0.011903081	0.003577129
200x200	0.021192646	0.067221732	0.015321283
300x300	0.043022308	0.192650687	0.031089808
400x400	0.074337628	0.425288795	0.054513754
500x500	0.112189236	0.931691302	0.083771662

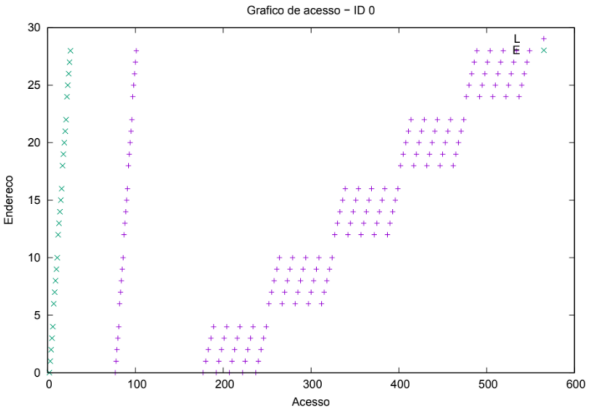
Conforme esperado na análise de complexidade, temos o pior desempenho para a operação de multiplicação e o melhor caso para as operações de soma e transposição.

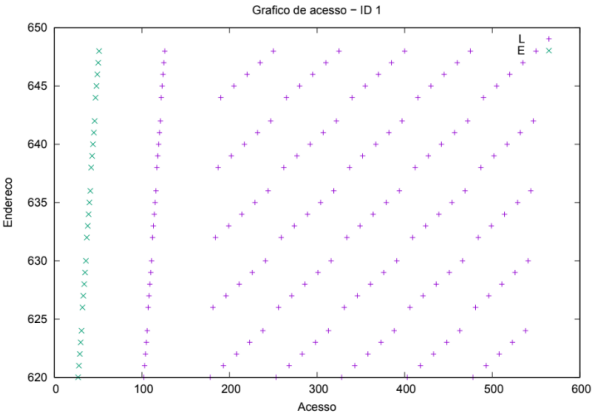
Por fim, ao utilizar o **gprof** para analisar cada operação efetuada em uma matriz de tamanho 500x500, é possível entender que na operação de multiplicação o maior custo de execução advém do método que faz a multiplicação entre as matrizes. Já nas operações de soma e transposição, o maior custo de execução vem do método que faz a leitura das matrizes através do arquivo de texto usado para configurá-las.

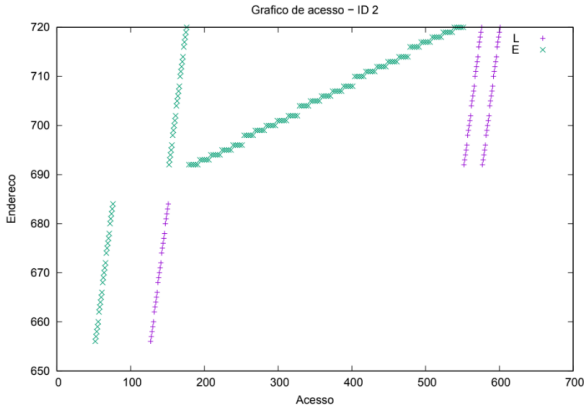
Nesse sentido, pode ser interessante encontrar outra forma de executar a multiplicação das matrizes que diminua o custo de complexidade de $O(n^3)$ para outro mais baixo. No caso das operações de soma e transposição, pode ser interessante encontrar formas mais eficientes de efetuar a leitura dos arquivos de configuração das matrizes.

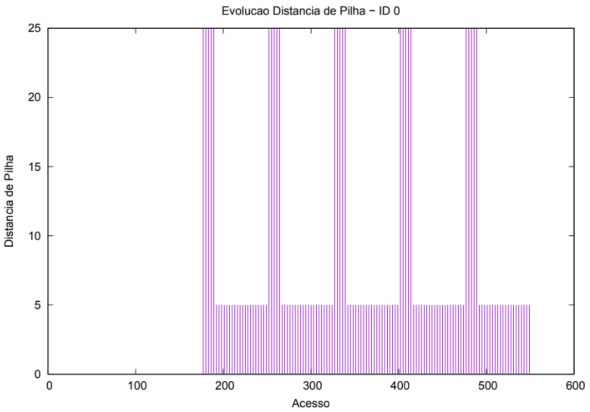
6.2. Análise de padrão de Acesso à Memória e Localidade de Referência

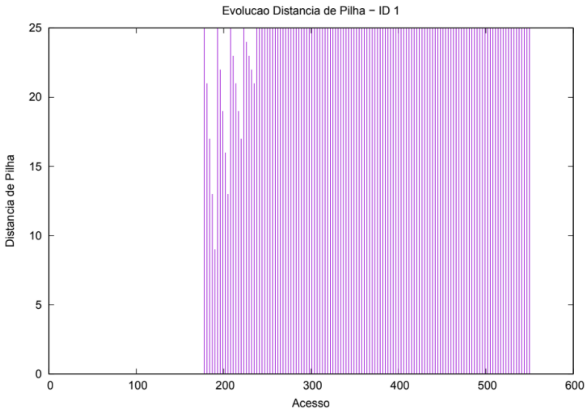
6.2.1. Multiplicação

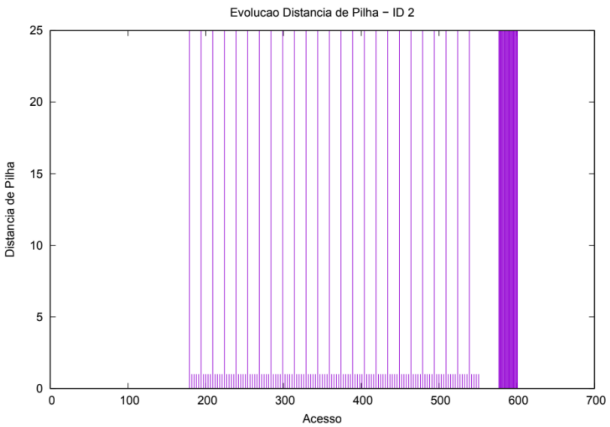


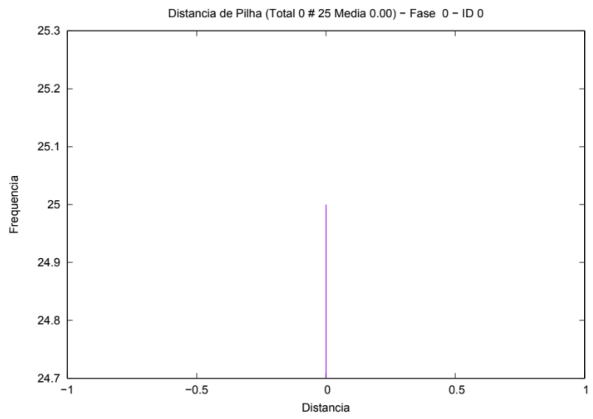


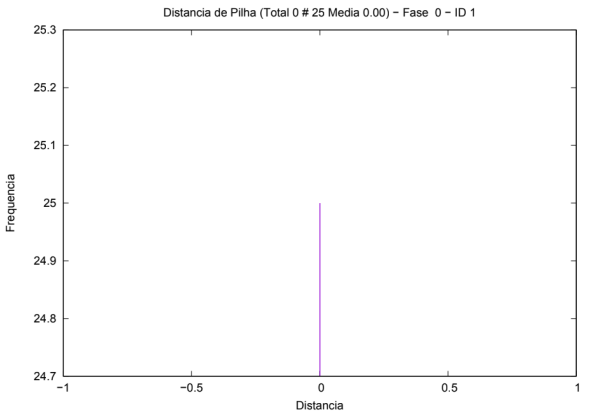


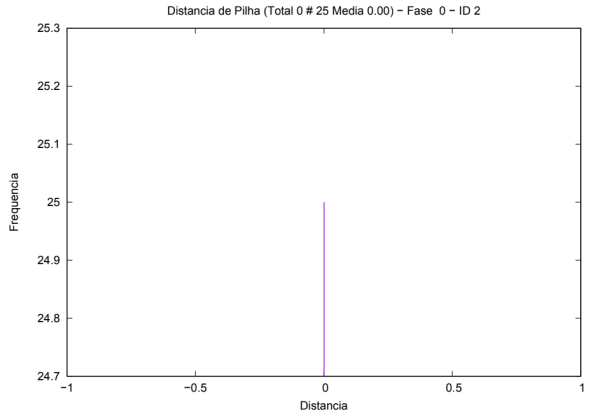


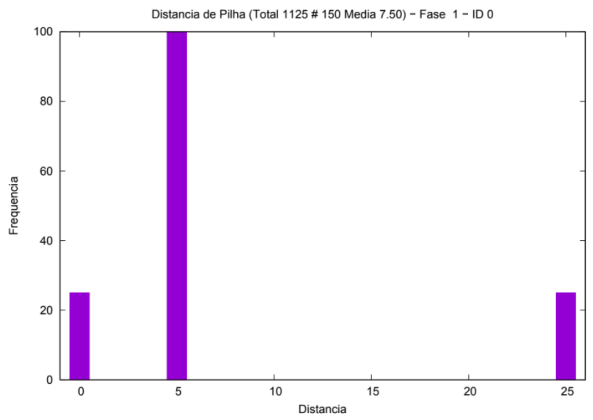


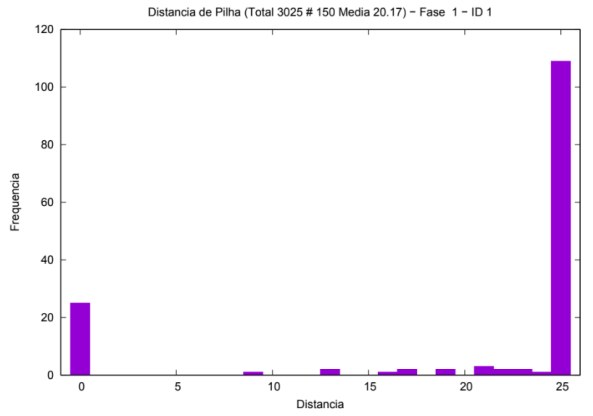


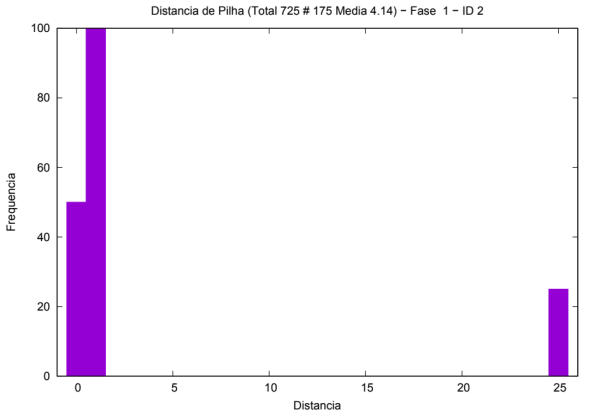

Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 0
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
−1
−0.5
0
0.5
1

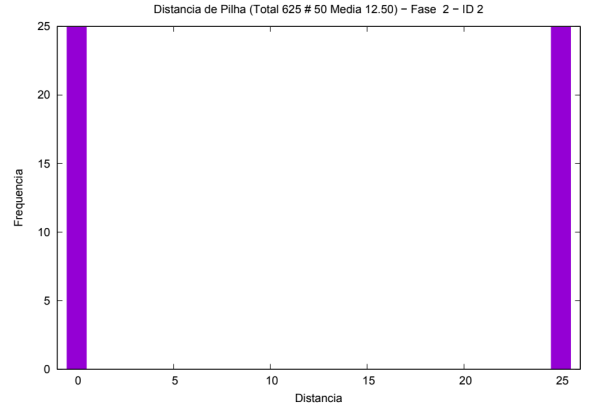

Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 1
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
−1
−0.5
0
0.5
1


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
120
100
80
60
40
20
0
0
5
10
15
20
25


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


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

Ao observar os gráficos de acesso de memória na operação de multiplicação, é possível perceber que como estamos acessando as linhas da matriz 1, os endereços de acesso estão muito próximos, de outra parte, como estamos acessando as colunas da matriz 2, é possível perceber que os endereços de acesso estão menos próximos um dos outros.

Isso era esperado visto que quando fazemos a inicialização das matrizes, a alocação dos dados da matriz é feito num loop que segue o padrão de inicializar a matriz linha por linha. Ou seja, a leitura que for efetuada nesse formato terá o benefício de acessar endereços de memória que estão mais próximos (isso ocorre com a matriz 1).

Nesse sentido, para diminuir a distância dos endereços de memórias da matriz 2, pode ser interessante fazer sua inicialização coluna por coluna, visto que esse é o padrão usado na leitura dessa matriz durante a multiplicação.

De outra parte, ao observar os gráficos de distância de pilha, é possível notar que no caso da matriz 1, temos uma distância de pilha alta apenas nas primeiras 5 leituras de cada conjunto de leitura. Isso ocorre pois dentro de um mesmo conjunto de leitura, o acesso dos endereços de memória seguintes são os mesmos, ou seja, como a pilha já está organizada por conta das leituras iniciais, não há deslocamentos nos itens da pilha.

Ao observar a distância de pilha da matriz 2, é possível observar que seu conjunto de leituras está sempre variando os endereços que estão sendo acessados na matriz, então a pilha não consegue se manter organizada e sempre é lida de uma forma diferente à cada iteração. Isso faz com que ao longo da execução de toda a operação a distância de pilha da matriz 2 mantenha valores extremamente altos.

Por fim, no caso da matriz 3 (resultante), temos um único pico de distância de pilha no início da operação de seu conjunto de escritas, pois no restante das escritas desse conjunto, acessamos literalmente o mesmo endereço de memória.

6.2.1. Soma

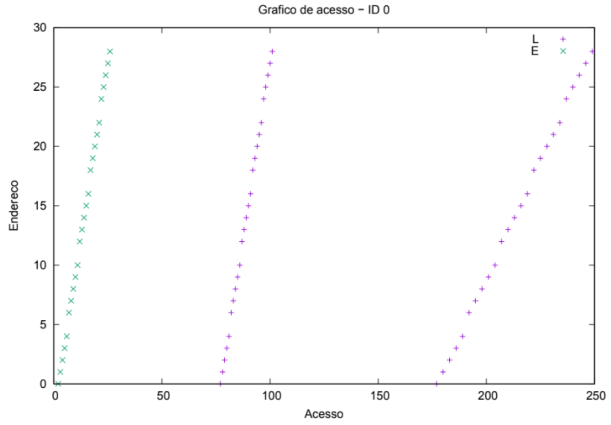


Grafico de acesso − ID 0
Endereco
Acesso
L
E

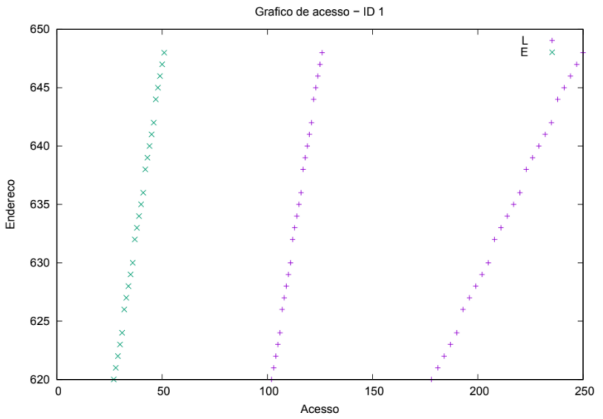


Grafico de acesso − ID 1
Endereco
Acesso
L
E

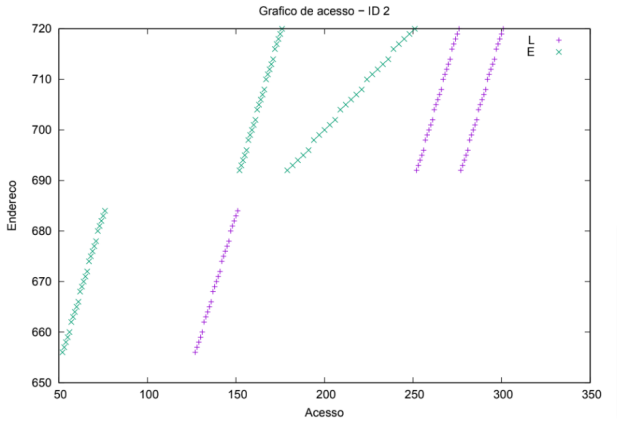
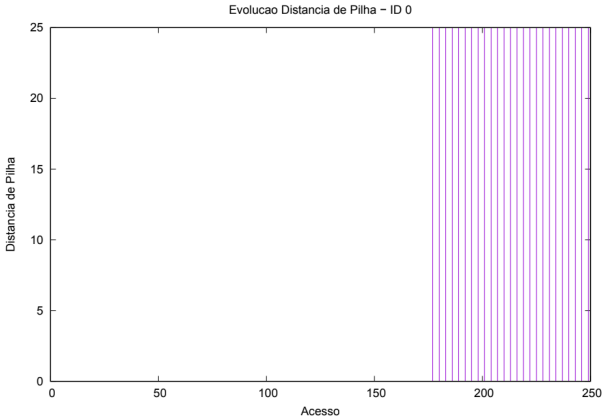
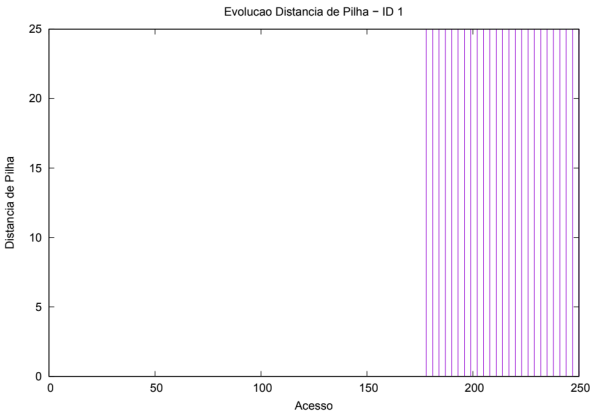
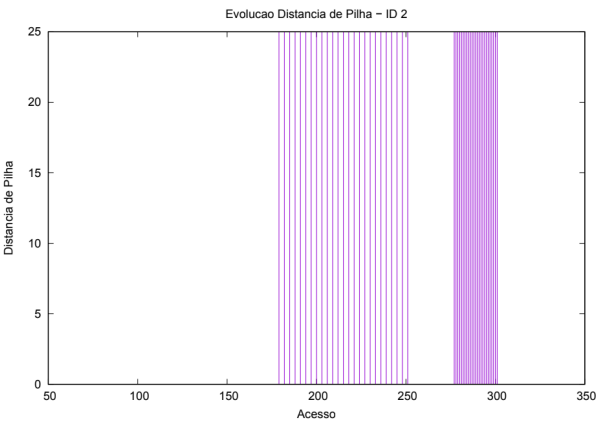
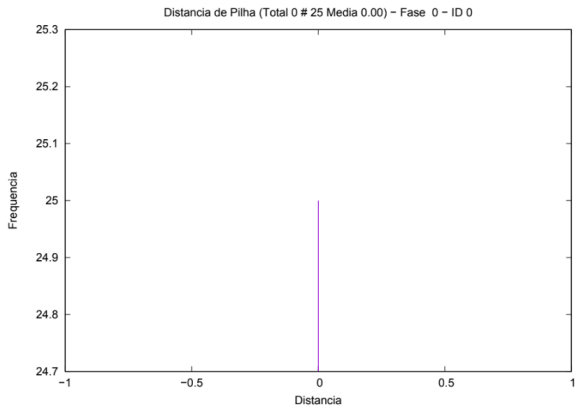


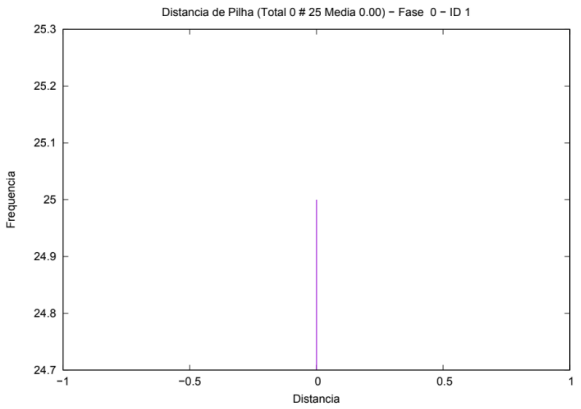
Grafico de acesso − ID 2
Endereco
Acesso
L
E

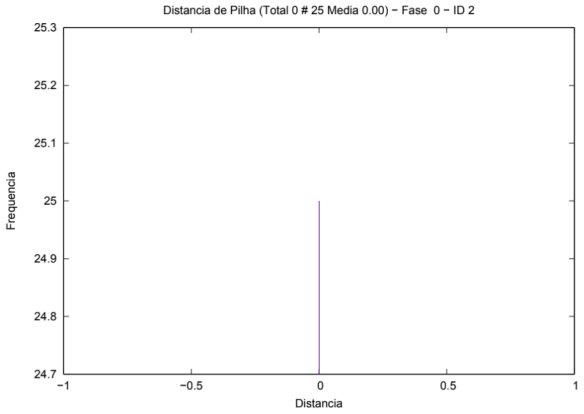

Evolucao Distancia de Pilha − ID 0
Distancia de Pilha
Acesso

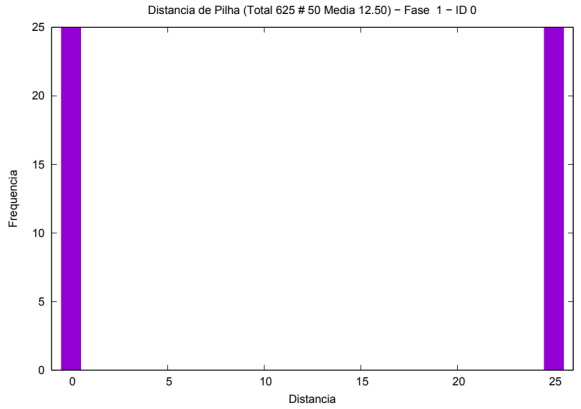

Evolucao Distancia de Pilha − ID 1
Distancia de Pilha
Acesso

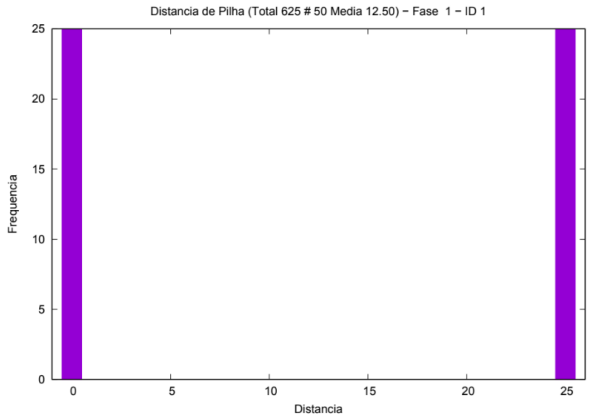

Evolucao Distancia de Pilha − ID 2
Distancia de Pilha
Acesso

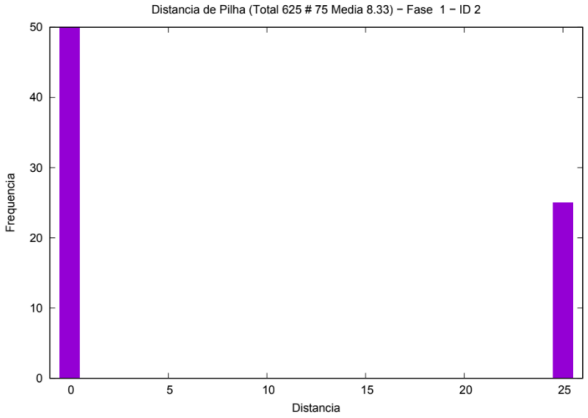

Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 0
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1

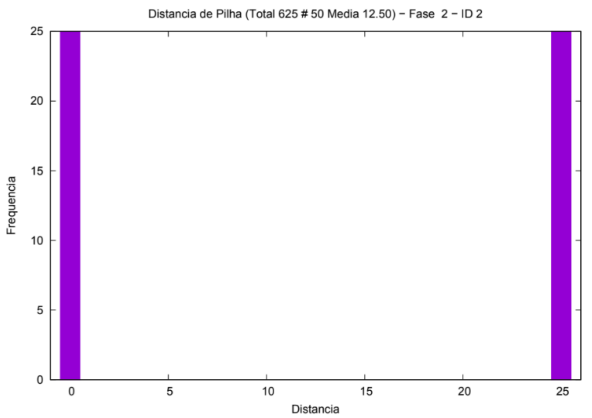

Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 1
Frequencia
Distancia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1


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
25
20
15
10
5
0
0
5
10
15
20
25


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

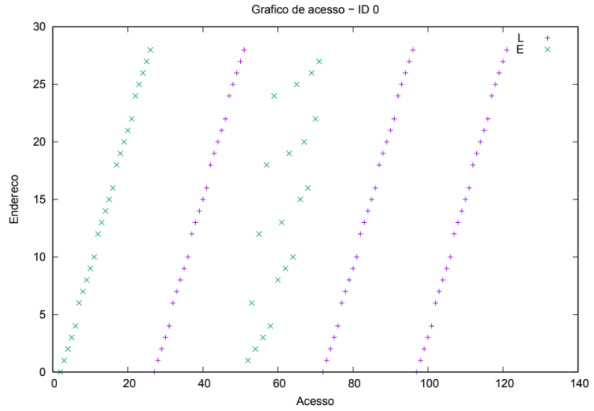

Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 2 - ID 2
Frequencia
Distancia
25
20
15
10
5
0
0
5
10
15
20
25

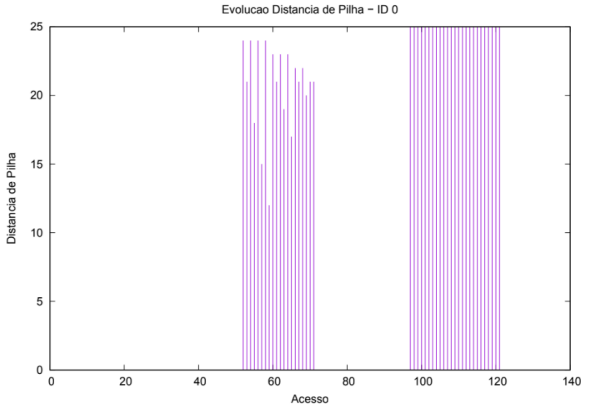
Quando observamos os gráficos de acesso de memória da operação de soma, diferente da operação de multiplicação, notamos um certo padrão na distância dos endereços de memórias que são lidos durante a operação envolvendo as matrizes 1 e 2.

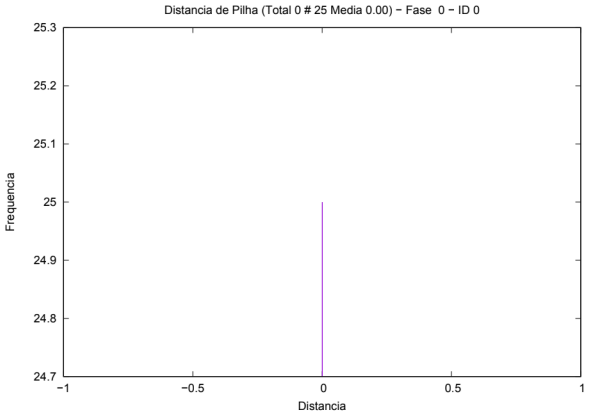
Isso também era algo esperado, visto que no nosso algoritmo, a leitura das matrizes 1 e 2 segue o mesmo padrão, que é o de ler seus dados linha por linha.

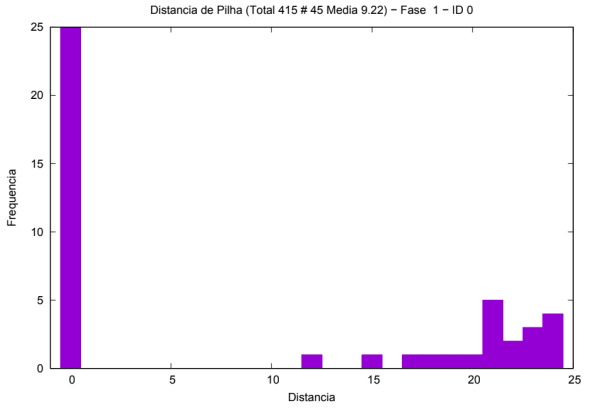
De outra parte, quando analisamos os gráficos de distância de pilha, é possível perceber que para todas as matrizes, a distância de pilha sempre mantém valores altos. Isso ocorre pois durante a execução dessa operação, sempre estamos acessando endereços de memória diferentes em todas as matrizes.

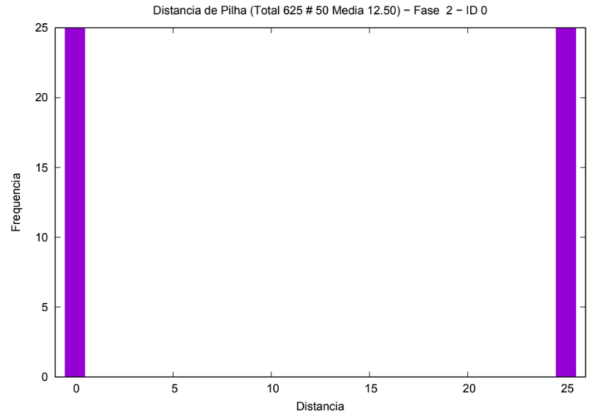
6.2.1. Transposição











No caso da operação de transposição, ao analisar os gráficos de acesso de memória, é possível ver que no início temos a operação de escrita e leitura para fazer a inicialização da matriz.

Em seguida, conforme era esperado, temos a operação de inversão dos valores de linha e coluna que são realizados de 4 em 4. Isso ocorre pois a matriz utilizada nos testes possui dimensões 5x5 e, geralmente quando fazemos uma transposição, o elemento mais central não se modifica, ou seja, de fato o esperado era realizar as operações de 4 em 4.

De outra parte, quando analisamos a distância de pilha, é possível perceber que a distância de pilha começa com um valor mais alto e sofre algumas quedas. Isso ocorre pois como estamos fazendo uma inversão de valores de alguns campos das matrizes, pode ocorrer de acessarmos consecutivamente um mesmo endereço de memória durante a operação, portanto, quando isso ocorre, temos uma diminuição na distância de pilha.

7. Conclusões

Nesse trabalho prático, fizemos um programa para efetuar umas das operações mais comuns envolvendo matrizes, que foram: multiplicação, soma e transposição. Inicialmente fizemos uma análise de complexidade do programa como um todo para entender se durante a execução do programa, o comportamento estaria dentro do esperado. Em seguida, através da execução desse programa, conseguimos gerar dados de acesso de memória e tempo de execução para cada tipo de operação. Por fim, fomos capazes de tratar os dados gerados e analisá-los para entender o comportamento do algoritmo de forma mais aprofundada (por exemplo observando as mudanças de distância de pilha durante a execução de cada operação).

Diante disso, fomos capazes de entender de forma prática como a localidade de referência e análise de complexidade funcionam. Porém, ainda mais importante que melhorar a habilidade com as técnicas utilizadas, foi aprimorar ainda mais nosso pensamento crítico/analítico - que é algo extremamente importante na área da computação.

Por fim, vale ressaltar que, durante a execução do trabalho, a principal dificuldade surgiu mais no sentido de conseguir analisar os dados encontrados e entender o porquê deles se comportarem de tal maneira.

Apêndice

Instruções para compilação e execução

Antes de efetuar a compilação do programa principal, é necessário abrir a pasta **analisamem** e rodar o comando **make**.

Agora, para efetuar a compilação do programa, basta entrar na pasta **TP** e rodar o comando **make** em seu terminal.

Esse comando irá executar as tarefas presentes no arquivo **Makefile** e efetuar tanto a compilação quanto a execução do programa.