Trabalho Prático 0:

Operações com Matrizes Alocadas Dinamicamente

Erick Sunclair Santos Batista

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

ericksunclair@ufmg.br

1. Introdução

Matrizes são estruturas de dados largamente utilizadas na computação. Porém, como outras estruturas de dados, as matrizes podem ter grandes variações de tamanhos o que pode tornar o seu manuseio muito dispendioso para o computador. Esse trabalho busca avaliar diferentes operações com matrizes nas linguagens C/C++ tendo em vista conceitos como desempenho, robustez e abstração.

Para isso foi desenvolvido um Tipo Abstrato de Dados Matriz com três operações muito importantes: a soma, a multiplicação e a transposição de matrizes. Em seguida foram feitos testes de desempenho com esse TAD buscando avaliar o custo temporal dessas operações para o computador e também fazer a análise do acesso a posições de memória ao longo do tempo.

A seção 2 aborda as características do código implementado e as configurações usadas para testar o programa. A seção 3 explica como acessar e executar o programa. A seção 4 apresenta uma análise de complexidade de tempo e espaço do programa. Por fim, a seção 5 possui a conclusão do trabalho desenvolvido.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilado pelo compilador G++ da GNU Compiler Collection e os testes foram realizados em um computador com Windows 10, intel core i5 3.4 GHz e 8GB de memória RAM.

2.1. Matriz

O programa teve como base uma matriz alocada dinamicamente, ou seja, em que a memória é alocada em tempo de execução, o que significa que nós podemos executar o programa escolhendo o tamanho exato da matriz que desejamos criar e a quantidade correta de memória necessária será alocada, evitando desperdício de espaço de memória e alocação desnecessária e indevida e aumentando o desempenho do programa e o seu reúso.

No arquivo “mat” foi criada uma classe “Matrix” de tamanho variável e escolhido pelo operador. Essa matriz classe possui um construtor que recebe as duas dimensões desejadas para a matriz e aloca ela na memória. O destrutor desaloca o espaço ocupado. Essa classe também tem métodos para inicializar a matriz apenas com valor zero e para inicializa-la com valores aleatórios dentro de um limite. Você também pode modificar um elemento da matriz ou ler um elemento, imprimir a matriz inteira para o terminal e copiar uma

matriz para a outra, mas os métodos mais importantes desenvolvidos para este trabalho são os de soma, de multiplicação e de transposição de matrizes, que terão seu desempenho analisado pela aplicação.

2.2. Registro de Acessos

A biblioteca de registro de acessos, “memlog”, desenvolvida pelo professor Wagner Meira Jr., consiste de um clock modo relógio para registro de tempo, um contador de acessos e algumas funções responsáveis por controlar e registrar o acesso a posições de memória seja para leitura, seja para escrita de dados no nosso Tipo Abstrato de Dados Matriz. Esse programa é usado pelo nosso programa principal “matop” e pela própria implementação do nosso TAD “mat” para retornar o momento exato em que houveram cada uma das operações de escrita ou de leitura na nossa matriz, bem como uma contagem de quantas vezes essas operações ocorreram, qual posição da memória foi acessada e o tamanho dessa posição.

Na própria implementação dos métodos da classe matriz sempre que existe uma operação de escrita ou leitura na matriz é chamada uma função da biblioteca de registro relacionada a escrita ou a leitura, respectivamente. A biblioteca de registro de acessos recebe da nossa classe matriz a posição de memória em que houve a operação e o tamanho dessa posição de memória, seja para registro de leitura seja para registro de escrita, e escreve em um arquivo de registro uma linha com as informações sobre esse acesso (momento do acesso, tipo do acesso, etc.). Ao fim da execução este arquivo terá todas as operações de escrita e leitura executadas com as suas devidas informações.

O programa principal faz teste com uma das três operações: soma, multiplicação e transposição de matrizes, para que possa ser gerado o arquivo de registro de acessos. O operador ao executar o programa principal “matop” envia para esse programa qual das três operações deseja executar, se deseja registrar os acessos à memória, em qual arquivo deseja salvar os registros e as dimensões das matrizes nas quais deseja executar as operações. Ao fim da execução o arquivo com todas as informações é gerado.

3. Instruções de Compilação e Execução

Para executar o programa recomenda-se usar o Makefile disponibilizado na matriz do diretório do programa pois ele já compila, linka e executa um programa para cada uma das três operações analisadas. Para isso deve-se seguir os seguintes passo:

- Utilizando um terminal acesse o diretório “/Erick_Sunclair_2020026877/TP”
- Execute o arquivo Makefile utilizando o seguinte comando: “make all”
- No próprio terminal aparecerão as matrizes resultado da soma, da multiplicação e da transposição, respectivamente, mas o que realmente nos interessa são os arquivos “somalog.out”, “multlog.out” e “transplog.out” que serão gerados no diretório “/tmp”. Nele estão os registros de acesso à memória que serão usados para analisar o desempenho do nosso TAD Matriz
- Para apagar os objetos criados na compilação do programa basta executar no terminal outro comando do arquivo Makefile: “make clean”
- E podemos executar novamente o programa.

Uma observação importante é que podemos editar as linhas 26, 27 e 28 do arquivo Makefile para desligar o registro de acesso à memória (retirando o -l das linhas desejadas), além de editar as dimensões das matrizes nas quais queremos executar as operações mudando os valores após -x e -y (as duas dimensões da matriz).

4. Análise de Complexidade

4.1. Complexidade Temporal

Com relação à complexidade temporal o nosso TAD Matriz possui um construtor e um destrutor de ordem O(tx), onde tx é o número de linhas da matriz que será alocada, já que o programa vai iterar sobre esse valor para fazer a alocação dessas linhas. Assim como a função que copia uma matriz para outra. Já as funções para inicializar a matriz como nula ou com valores aleatórios e a função de imprimir a matriz iteram sobre tx e em seguida sobre ty, sendo esses o número de linha e o número de colunas da matriz, respectivamente, então a matriz é de ordem O(tx * ty). As funções para escrever ou ler um elemento da matriz só fazem operações básicas como comparações, atribuição ou retorno, então são funções de ordem O(1).

Falando agora das operações de maior interesse, a função de soma possui, além das operações de ordem O(1), duas iterações de ordem O(tx) e uma de ordem O(tx * ty), portanto a ordem de complexidade temporal dela é O(tx * ty). A de multiplicação também tem a mesma ordem pela mesma lógica de soma. A função de transposição, por outro lado, possui duas iterações de ordem O(tx * ty) e uma de ordem O(ty), o que acaba por deixa-la com a mesma ordem de complexidade das funções de soma e de multiplicação.

4.2. Complexidade Espacial

Analisando com relação à complexidade espacial nós temos as funções construtora, de cópia, de soma, de multiplicação e de transposição com ordem O(tx * ty), pois todas essas alocam um espaço de memória de tamanho tx * ty. Todas as demais funções são de ordem O(1) com relação à complexidade espacial criam apenas variáveis simples ou nem novas variáveis na memória.

5. Análise dos Resultados

Os testes realizados com o nosso programa nos possibilitaram tirar conclusões sobre sua eficiência temporal e também com relação ao seu comportamento no acesso à memória.

5.1. Com Relação ao Tempo

Foram executados diversos testes temporais utilizando o GNU Profiler (gprof). Nesse caso o teste foi feito com o registro de acesso à memória desligado pois ele poderia interferir no tempo de execução das funções. Os resultados infelizmente não foram muito conclusivos pois os tempos medidos foram baixíssimos. Veja na Imagem 5.1 abaixo o resultado dos tempos totais de execução de cada função para a execução do programa com matrizes de dimensão 500 X 500. Como podemos ver, mesmo com uma matriz de dimensão grande, o gprof não registrou valores muito conclusivos de tempo de uso das funções, talvez devido à velocidade do computador onde foram feitos os testes, mesmo ele sendo de configuração mediana. Portanto, podemos notar a quantidade enorme de vezes que as funções de registro de escrita ou de leitura na nossa matriz são chamadas. Isso se dá pois, mesmo essas funções não sendo executadas, pois o registro está desativado, elas ainda assim são chamadas, um problema que pode influenciar o desempenho do programa sem necessidade e pode ser melhorado para atualizações futuras do programa.

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00   750000     0.00     0.00  escreveMemLog(long, long)
  0.00      0.00     0.00   500000     0.00     0.00  leMemLog(long, long)
  0.00      0.00     0.00   250000     0.00     0.00  std::setprecision(int)
  0.00      0.00     0.00        2     0.00     0.00  Matrix::~Matrix()
  0.00      0.00     0.00        1     0.00     0.00  parse_args(int, char**)
  0.00      0.00     0.00        1     0.00     0.00  iniciaMemLog(char*)
  0.00      0.00     0.00        1     0.00     0.00  desativaMemLog()
  0.00      0.00     0.00        1     0.00     0.00  finalizaMemLog()
  0.00      0.00     0.00        1     0.00     0.00  Matrix::imprimeMatriz()
  0.00      0.00     0.00        1     0.00     0.00  Matrix::transpoeMatriz()
  0.00      0.00     0.00        1     0.00     0.00  Matrix::inicializaMatrizAleatoria()
  0.00      0.00     0.00        1     0.00     0.00  Matrix::Matrix(int, int)
```

Imagem 5.1 – Desempenho temporal de cada função para matrizes 500 X 500

Em seguida foi feita um teste com matrizes de mesma dimensão 500 X 500 mas com o registro de acesso à memória ligado e o resultado já tem valores de tempo mais visíveis (Figura 5.2), ilustrando como essas funções tem um custo temporal significativo para o programa, e devem ser usadas apenas quando está sendo analisado o comportamento da função e como ela acessa às posições de memória.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.00      0.01     0.01  1250000     0.00     0.00  clkDifMemLog(timespec, timespec, timespec*)
 50.00      0.02     0.01                             clock_gettime
  0.00      0.02     0.00   750000     0.00     0.00  escreveMemLog(long, long)
  0.00      0.02     0.00   500000     0.00     0.00  leMemLog(long, long)
  0.00      0.02     0.00   250000     0.00     0.00  std::setprecision(int)
  0.00      0.02     0.00        2     0.00     0.00  Matrix::~Matrix()
  0.00      0.02     0.00        1     0.00     0.00  parse_args(int, char**)
  0.00      0.02     0.00        1     0.00     0.00  ativaMemLog()
  0.00      0.02     0.00        1     0.00     0.00  iniciaMemLog(char*)
  0.00      0.02     0.00        1     0.00     0.00  finalizaMemLog()
  0.00      0.02     0.00        1     0.00     0.00  Matrix::imprimeMatriz()
  0.00      0.02     0.00        1     0.00     6.00  Matrix::transpoeMatriz()
  0.00      0.02     0.00        1     0.00     2.00  Matrix::inicializaMatrizAleatoria()
  0.00      0.02     0.00        1     0.00     0.00  Matrix::Matrix(int, int)
```

Imagem 5.2 – Desempenho temporal de cada função para matrizes 500 X 500 com registro de acesso ligado

5.2. Com Relação à Memória

Ligando o registro de acessos à memória recebemos arquivos de saída que registram o comportamento do programa e seus acessos a cada posição de memória para cada operação (soma, multiplicação e transposição).

Para uma matriz de dimensão 5 temos 162 eventos de acesso à memória para soma, 502 acessos para multiplicação e 102 para transposição, o que mostra que a multiplicação é a operação que mais acessa a memória. Além disso, ao aumentar as dimensões das matrizes para 20 o número de eventos, para a soma, aumentou aproximadamente 16 vezes, para 2562 eventos. Para a transposição houve aumento também de 16 vezes, para 1602 eventos, e para a multiplicação o aumento foi de 52 vezes, para 26002, mostrando que o aumento do número de acessos no caso da multiplicação também é muito maior.

6. Conclusão

Neste trabalho foi criado em C++ um Tipo Abstrato de Dados do tipo Matriz e foram feitos teste de desempenho de tempo nesse programa, analisando separadamente suas funções. Também foram registrados e analisados os acessos desse programa a diferentes posições de memória, analisando o momento em que cada posição de memória é acessada e quantos acessos existem.

Com os testes realizados foi possível notar que a multiplicação de matrizes é a operação mais custosa tanto em termos de gasto de memória quanto em termos de gasto temporal. Além disso pudemos perceber também que as próprias operações de registro de acesso à memória possuem um custo enorme ao equipamento, podendo afetar o desempenho do próprio programa.

Com esse trabalho foi possível aprender ou se aprofundar em conceitos importantíssimos como Tipos Abstratos de Dados, Alocação Dinâmica, desempenho e robustez de programas, localidade de referência, bem como praticar os conceitos de complexidade temporal e espacial e suas ordens de grandeza. Também foram encontradas algumas dificuldades no processo, principalmente no uso de novas ferramentas, como o GNU Profiler, "gnuplot" e a própria biblioteca de registro de acesso à memória, sendo esta mais trabalhosa apenas para a compreensão inicial, ao contrário do "gprof" e "gnuplot" que, devido ao uso difícil ou talvez mal funcionamento acabou por impossibilitar melhor análise do problema. Outra questão difícil, porém muito engrandecedora ao aluno, foi a diferença entre a programação no terminal no Linux e no Windows, o que gerou alguns conflitos ao fazer testes com o Makefile disponibilizado no ambiente do Windows, mas que puderam ser circundados.

Referências

EMBARCADOS. **Análise de desempenho com GNU Profiler gprof**. Disponível em: https://www.embarcados.com.br/desempenho-gnu-profiler-gprof/. Acesso em: 11 nov. 2021.

EMBARCADOS. **Introdução ao Makefile**. Disponível em: https://www.embarcados.com.br/introducao-ao-makefile/. Acesso em: 11 nov. 2021.

INTELLECTUALE. **Alocação Dinâmica em C**. Disponível em: http://linguagemc.com.br/alocacao-dinamica-de-memoria-em-c/. Acesso em: 11 nov. 2021.