Trabalho Prático de Algoritmos e Estruturas de Dados

Lucas O. Silvestre[1]

[1]Sistemas de Informação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG – Brazil

lsilvs@dcc.ufmg.br

1. Objetivo inicial

O presente trabalho tem como objetivo familiarizar o aluno com algumas primitivas básicas da linguagem C, assim como os padrões de documentação e codificação esperados ao longo da disciplina.

Para isso, o aluno deverá implementar um algoritmo que calcule o Produto de Kronecker. O produto de Kronecker consiste em uma operação entre duas matrizes de dimensões arbitrárias que resulta em uma matriz em bloco.

Ou seja,

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix},$$

A figura a seguir exemplifica o produto desejado:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1\cdot 0 & 1\cdot 5 & 2\cdot 0 & 2\cdot 5 \\ 1\cdot 6 & 1\cdot 7 & 2\cdot 6 & 2\cdot 7 \\ 3\cdot 0 & 3\cdot 5 & 4\cdot 0 & 4\cdot 5 \\ 3\cdot 6 & 3\cdot 7 & 4\cdot 6 & 4\cdot 7 \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}.$$

As matrízes devem ser alocadas e desalocadas dinamicamente utilizando os comandos *malloc* e *free*.

2. Modelagem e Solução proposta

A modelagem implementada consiste na leitura de um arquivo de entrada especificado na chamada do programa. A primeira linha do arquivo de entrada representa o número de instâncias que o arquivo contém. A linha seguinte representa as dimensões da matriz que virá a seguir. As linhas seguintes são os elementos dessa matriz. Esse padrão se repete para o número de instâncias.

A solução proposta é iniciar a leitura do arquivo de entrada salvando os dados em variáveis. Observe que a consistência do padrão do arquivo de entrada é de extrema importância para o funcionamento do programa. Partindo do ponto que o arquivo de entrada segue o modelo especificado, o algoritmo proposto lê o número de instâncias e itera sobre ele. A cada iteração do número de instâncias, o par de matrizes é lido do arquivo e, em seguida, o produto de Kronecker é aplicado.

Para realizar o produto de Kronecker precisamos iterar sobre o par de matrizes de cada instância e fazer a mutiplicação conforme proposto.

```
// Preenche a matriz resultante (Produto de Kronecker)
for (i = 0; i < numLinhasA; ++i) {
    for (j = 0; j < numColunasA; ++j) {
        for (k = 0; k < numLinhasB; ++k) {
            for (l = 0; l < numColunasB; ++l) {
                matrizResult[i * numLinhasB + k][j * numColunasB + l] = matrizA[i][j] * matrizB[k][l];
            }
        }
    }
}
```

Como iniciativa de modularização, foi criado um arquivo de funções para auxilizar o desenvolvimento e evitar retrabalho. As funções modularizadas são as de alocação de matriz, desalocação de matriz e preenchimento da matriz.

2.1. Aloca Matriz

A função para alocar a matriz recebe o número de linhas e colunas como parâmetro e retorna um ponteiro para a matriz alocada.

```
int **aloca_matriz(int linhas, int colunas) {
    // Aloca a matriz
    int ** matriz;
    int i;

    matriz = (int **) malloc(linhas * sizeof(int *));

    for(i = 0; i < linhas; i++) {
        matriz[i] = (int *) malloc(colunas * sizeof(int));
    }
    return(matriz);
}
```

2.2. Preenche Matriz

A função que preenche a matriz recebe o arquivo a ser lido e número de linhas e colunas da matriz. Nesse caso, a própria função preenche_matriz utiliza a função aloca_matriz para alocar uma nova matriz antes de preenchê-la.

Assim como a função aloca_matriz, preenche_matriz retorna um ponteiro para a matriz preenchida.

```
int **preenche_matriz(FILE *arquivo, int linhas, int colunas) {
    // Aloca a matriz
    int ** matriz;
    int i, j;

    matriz = aloca_matriz(linhas, colunas);

    // Preenche a matriz
    for(i = 0; i < linhas; i++) {
        for(j = 0; j < colunas; j++) {
            fscanf(arquivo, "%d", &matriz[i][j]);
        }
    }
    return(matriz);
}
```

2.3. Desaloca Matriz

Por fim, a função que desaloca_matriz recebe a matriz e o número de linhas como parâmetro e libera cada linhas e em seguida a própria matriz. Essa função não possui retorno.

```
void desaloca_matriz(int ** matriz, int linhas) {
    // Desaloca a matriz
    int i;
    for(i = 0; i < linhas; i++) {
        free(matriz[i]);
    }
    free(matriz);
}
```

Para cada iteração sobre o número de instâncias é realizado a alocamento de três matrizes (par de matrizes do arquivo + matriz resultante), o produto de Kronecker, a escrita da matriz resultando no arquivo de saída e o desalocamento das três matrizes alocadas ao final da iteração.

Após finalizar o processamento de todos os pares de matrizes, os arquivos de entrada e saída são fechados.

3. Análise de Complexidade

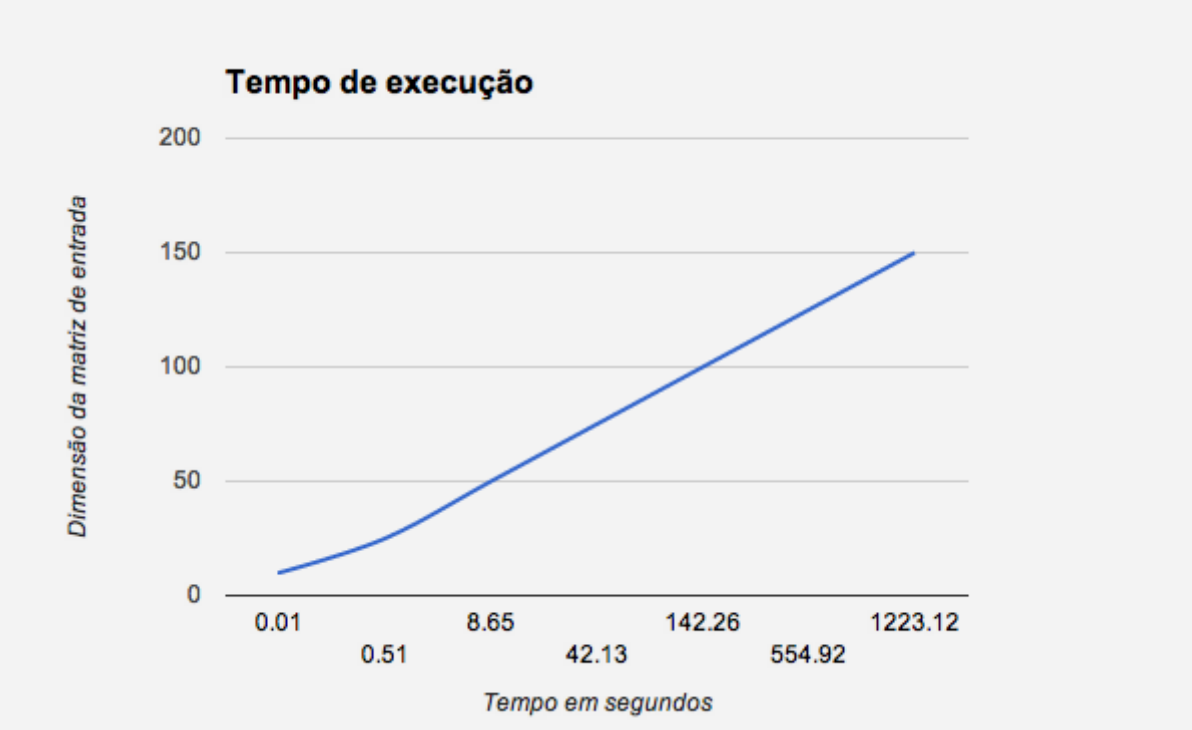
Como já foi citado anteriomente, para calcular o produto de Kronecker se faz necessário iterar sobre toda as duas matrizes de entrada. Logo, precisamos de quatro for's alinhados. Por isso, a ordem de complexidade deste algoritmo é O(m * n * p * q), onde m é o número de linhas da primeira matriz, n o número de colunas da primeira matriz, p o número de linhas da segunda matriz e q o número de colunas da segunda matriz.

Como a maior complexidade é a do produto de Kronecker, a complexidade final do algoritmo é o citado acima.

4. Experimentos

Para realizar os testes de desempenho, foi criado um algoritmo para gerar matrizes e popular o arquivo de entrada. Com esse programa, conseguimos gerar quantas instâncias forem necessárias para realizar testes de desempenho e consumo de memória.

O gráfico a seguir demonstra o tempo de execução do programa de acordo com a dimensão da matriz do arquivo de entrada



5. Especificação

Todo o desenvolvimento do código foi realizado utilizando um MacBook Air com as seguintes especificações:

Processador: Intel Core i5 1,7 GHz

Memoria: 4 GB 1333 MHz DDR3

Sistema Operacional: OS X 10.8.2

IDE: Sublime Test 2

GCC: i686-apple-darwin11-llvm-gcc-4.2 (GCC) 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)

6. Conclusão

Em linhas gerais, a maior dificuldade da implementação foi gerar a matriz resultando conforme especificado pelo produto de Kronecker. Com a modularização em funções, reduzimos o número de linhas de código e atingimos um nível de reuso ideal para o projeto em questão. Ficou evidente também que quanto maior a dimensão da matriz de entrada, maior o arquivo de saída gerado, podendo chegar a gigas de espaço em hd.