UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Projeto de Programação I

Multiplicação de Matrizes

Objetivo

O objetivo deste trabalho é verificar se é válido ou não o senso comum de que algoritmos iterativos geralmente apresentam um melhor desempenho com relação a tempo de execução quando comparados a suas respectivas versões recursivas. Para tal, serão implementadas as versões iterativa e recursiva de um algoritmo e realizados testes para diferentes cargas (*workloads*). Através da análise dos resultados em termos de tempo de execução registrado em cada um dos testes, será possível concluir acerca da hipótese inicialmente dada.

1 Introdução

Na Matemática, uma *matriz* é um arranjo retangular de números, símbolos ou expressões, chamados de *entradas* ou *elementos*. Tais elementos são dispostos de forma horizontal em *linhas* e de forma vertical em *colunas*. O número de linhas e colunas de uma matriz determina a sua *dimensão*: se uma determinada matriz possui m linhas e n colunas ($m, n \in \mathbb{N}^+$), diz-se que ela é uma matriz m por n, ou possui dimensão $m \times n$. Por exemplo, diz-se que a matriz

$$\begin{pmatrix} 5 & 13 & -8 \\ 24 & 9 & -2 \end{pmatrix}$$

é uma matriz 2 por 3 (dimensão 2×3) pelo fato de ter duas linhas e três colunas. Um dado elemento de uma matriz A de dimensão $m \times n$ é tipicamente denotado a_{ij}, referindo-se a um elemento localizado na linha i e coluna j da matriz, sendo $1 \leq i \leq m$ e $1 \leq j \leq n$.

Além da própria Matemática, matrizes podem ser encontradas em diversas áreas do mundo científico, tais como Física, Computação, Economia e Estatística. Tanto na teoria quanto na prática, matrizes são utilizadas principalmente como estruturas para a representação de fenômenos, organização de dados, relações entre variáveis, equações em sistemas lineares, etc. Em várias linguagens de programação, matrizes são representadas como *arranjos multidimensionais* em que cada linha e cada coluna é identificada por um número inteiro chamado *índice*, em umas iniciando em 0 e em outras em 1.

2 Multiplicação de matrizes

Diversas operações podem ser realizadas sobre matrizes a fim de modificar os valores de seus elementos ou mesmo as suas dimensões, tais como adição, subtração, transposição, etc. Dentre essas operações, a *multiplicação de matrizes* destaca-se pelas suas diversas aplicações teóricas e práticas na Ciência pelo fato de diversos problemas poderem ser representados e solucionados através de uma multiplicação entre matrizes. O assunto é de tamanha importância que, até o momento, encontrar o algoritmo mais rápido para realizar a multiplicação de matrizes é um problema em aberto na Ciência da Computação[1]. Diversas pesquisas têm sido desenvolvidas desde a década de 1960 no intuito de propor algoritmos que consigam computar a multiplicação de matrizes da forma mais eficiente possível, principalmente para matrizes de grandes dimensões. Com os avanços nos últimos anos em *hardware* e programação paralela e concorrente, passou-se a também investigar formas de multiplicar matrizes que fossem eficientes tanto com relação a de tempo de execução quanto a consumo de recursos computacionais.

2.1 Algoritmo iterativo

A operação de multiplicação entre duas matrizes pode ser enunciada da seguinte forma:

> O produto de uma matriz A com dimensão $m \times p$ por outra matriz B com dimensão $p \times n$ é uma matriz C com dimensão $m \times n$ (sendo $m, n, p \in \mathbb{N}^+$) em que cada elemento c_{ij} é obtido pela soma dos produtos dos elementos correspondentes da i-ésima linha da matriz A pelos elementos da j-ésima coluna da matriz B:
>
> $$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$
>
> sendo $1 \leq i \leq m$, $1 \leq j \leq n$ e $i, j, k \in \mathbb{N}^+$.

Uma restrição importante a ser observada diz respeito ao fato que as duas matrizes A e B precisam ser necessariamente *compatíveis* entre si, isto é, o número de colunas da matriz A deve ser igual ao número de linhas da matriz B.

O procedimento iterativo para realizar a multiplicação entre duas matrizes compatíveis A e B fornecidas como entrada envolve a execução de três laços aninhados. O Algoritmo 1 esquematiza na forma de pseudocódigo esse procedimento, que resulta na matriz produto C:

[1] Outros problemas ainda sem solução na Ciência da Computação são listados na Wikipedia em https://en.wikipedia.org/wiki/List_of_unsolved_problems_in_computer_science

Algoritmo 1 Algoritmo iterativo para multiplicação de matrizes

```
1: variável A : Matriz(m, p)        // A possui dimensão m × p
2: variável B : Matriz(p, n)        // B possui dimensão p × n
3: variável C : Matriz(m, n)        // C possui dimensão m × n
4: procedimento MultiplicaMatrizesIterativo(A, B, C : Matriz, m, n, p : Inteiro)
5:     variável i, j, k : Inteiro
6:     para i ← 1 até m faça
7:         para j ← 1 até n faça
8:             variável soma : Inteiro
9:             soma ← 0
10:            para k ← 1 até p faça
11:                soma ← soma + A[i, k] * B[k, j]
12:            fim para
13:            C[i, j] ← soma
14:        fim para
15:    fim para
16: fim procedimento
```

Para exemplificar, considere as seguintes matrizes A e B, ambas de dimensão 2×2:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} -1 & 3 \\ 4 & 2 \end{pmatrix}$$

Seguindo o Algoritmo 1, a computação da matriz produto C, também de dimensão 2×2, é feita da seguinte forma:

$$C = A.B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} -1 & 3 \\ 4 & 2 \end{pmatrix} = \begin{pmatrix} 1.(-1) + 2.4 & 1.3 + 2.2 \\ 3.(-1) + 4.4 & 3.3 + 4.2 \end{pmatrix} = \begin{pmatrix} 7 & 7 \\ 13 & 17 \end{pmatrix}$$

2.2 Algoritmo recursivo

A estratégia de *divisão e conquista* (D&C) consiste em particionar recursivamente um problema de maior complexidade em dois ou mais sub-problemas similares, porém de menor complexidade, até que se tornem simples o suficiente para serem resolvidos de maneira trivial. As soluções dos sub-problemas vão sendo então combinadas até que se tenha uma solução completa para o problema original.

Fazendo uso desse raciocínio, foram desenvolvidos algoritmos recursivos que fazem uso da estratégia D&C no intuito de encontrar uma forma mais eficiente de se multiplicar duas matrizes. Para que isso seja possível, é **mandatório** que as matrizes sejam quadradas (isto é, possuam o mesmo número de linhas e colunas) e que a dimensão $n \times n$ seja uma potência de base 2, ou seja, $n \in \{1, 2, 4, 8, \ldots, 2^k\}$ para algum $k \in \mathbb{N}$. Cada uma das matrizes de entrada e a matriz produto são *particionadas* em quatro sub-matrizes de dimensão $n/2 \times n/2$, de modo que o produto pode ser expresso em termos dessas submatrizes como se cada uma delas fosse um elemenento individual da matriz. Essas operações

vão sendo realizadas recursivamente até que o tamanho de cada submatriz seja igual a 1, quando a multiplicação pode ser feita de forma direta entre os elementos individuais.

O Algoritmo 2 esquematiza na forma de pseudocódigo a multiplicação de duas matrizes quadradas de dimensão $n \times n$ A e B, que resulta em uma matriz quadrada C, também de dimensão $n \times n$. Nas linhas 7 a 10, são feitas as quatro partições das matrizes A, B e C, todas de dimensão $n/2 \times n/2$, além da criação de mais oito matrizes quadradas de dimensão $n/2 \times n/2$ (P_1 a P_8) que irão armazenar temporariamente os resultados das oito multiplicações de submatrizes a serem realizadas (linhas 11 a 18). Em seguida, são feitas quatro operações de soma entre essas oito matrizes produto menores (linhas 19 a 22) para compor as quatro partes da matriz produto final C, o que é feito na linha 23.

Algoritmo 2 Algoritmo recursivo para multiplicação de matrizes quadradas de dimensão $n \times n$

```
1:  variável A, B, C : Matriz(n, n)
2:  procedimento MultiplicaMatrizesRecursivo(A, B, C : Matriz, n : Inteiro)
3:      se n = 1 então
4:          C[1,1] ← A[1,1] * B[1,1]
5:          retorne
6:      senão
7:          variável A11, A12, A21, A22 : Matriz(n/2, n/2)          // Partições da matriz A
8:          variável B11, B12, B21, B22 : Matriz(n/2, n/2)          // Partições da matriz B
9:          variável C11, C12, C21, C22 : Matriz(n/2, n/2)          // Partições da matriz C
10:         variável P1, P2, P3, P4, P5, P6, P7, P8 : Matriz(n/2, n/2)  // Sub-matrizes para produtos
11:         MultiplicaMatrizesRecursivo(A11, B11, P1, n/2)
12:         MultiplicaMatrizesRecursivo(A12, B21, P2, n/2)
13:         MultiplicaMatrizesRecursivo(A11, B12, P3, n/2)
14:         MultiplicaMatrizesRecursivo(A12, B22, P4, n/2)
15:         MultiplicaMatrizesRecursivo(A21, B11, P5, n/2)
16:         MultiplicaMatrizesRecursivo(A22, B21, P6, n/2)
17:         MultiplicaMatrizesRecursivo(A21, B12, P7, n/2)
18:         MultiplicaMatrizesRecursivo(A22, B22, P8, n/2)
19:         C11 ← SomaMatrizes(P1, P2)
20:         C12 ← SomaMatrizes(P3, P4)
21:         C21 ← SomaMatrizes(P5, P6)
22:         C22 ← SomaMatrizes(P7, P8)
23:         // Agrupar as quatro sub-matrizes C11, C12, C21 e C22 na matriz produto C
24:     fim se
25: fim procedimento
```

Para exemplificar, considere novamente as seguintes matrizes A e B, ambas de dimensão 2×2:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} -1 & 3 \\ 4 & 2 \end{pmatrix}$$

Seguindo o Algoritmo 2, a computação da matriz produto C, também de dimensão 2×2, seria feita através dos seguintes passos:

1) Particionamento das matrizes A e B em quatro submatrizes de dimensão 1×1 cada:

$$\begin{array}{cccc} A_{11} = (1) & A_{12} = (2) & A_{21} = (3) & A_{22} = (4) \\ B_{11} = (-1) & B_{12} = (3) & B_{21} = (4) & B_{22} = (2) \end{array}$$

2) Multiplicação das submatrizes, equivalendo à multiplicação direta dos elementos:

$$\begin{array}{rcrcrcl} P_1 & = & (A_{11}.B_{11}) & = & (1).(-1) & = & (-1) \\ P_2 & = & (A_{12}.B_{21}) & = & (2).(4) & = & (8) \\ P_3 & = & (A_{11}.B_{12}) & = & (1).(3) & = & (3) \\ P_4 & = & (A_{12}.B_{22}) & = & (2).(2) & = & (4) \\ P_5 & = & (A_{21}.B_{11}) & = & (3).(-1) & = & (-3) \\ P_6 & = & (A_{22}.B_{21}) & = & (4).(4) & = & (16) \\ P_7 & = & (A_{21}.B_{12}) & = & (3).(3) & = & (9) \\ P_8 & = & (A_{22}.B_{22}) & = & (4).(2) & = & (8) \end{array}$$

3) Soma das matrizes produto que calculadas:

$$\begin{array}{rcrcrcl} C_{11} & = & (P1 + P2) & = & ((-1)) + (8) & = & (7) \\ C_{12} & = & (P3 + P4) & = & (3) + (4) & = & (7) \\ C_{21} & = & (P5 + P6) & = & (-3) + (16) & = & (13) \\ C_{22} & = & (P7 + P8) & = & (9) + (8) & = & (17) \end{array}$$

4) Composição da matriz produto final C:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} 7 & 7 \\ 13 & 17 \end{pmatrix}$$

3 Tarefas

As tarefas a serem realizados neste exercício de programação consistem em basicamente (1) implementar os algoritmos para multiplicação de matrizes anteriormente descritos, (2) realizar sucessivas execuções dos algoritmos a fim de fazer uma análise comparativa entre eles, e (3) elaborar um relatório descrevendo as atividades que foram realizadas, os resultados observados e as conclusões obtidas.

3.1 Implementação dos algoritmos para multiplicação de matrizes

Você deverá implementar os Algoritmos 1 e 2, que são respectivamente a versão iterativa e a versão recursiva da multiplicação de matrizes. Para permitir o emprego de matrizes cujos elementos sejam números inteiros ou decimais, as funções que implementam cada algoritmo devem fazer uso de *templates* de função, obedecendo **estritamente** às seguintes assinaturas:

```
/**
 * @brief   Funcao que multiplica duas matrizes quadradas de dimensao n x n de forma iterativa
 * @param   A Matriz de entrada
 * @param   B Matriz de entrada
 * @param   n Dimensao das matrizes de entrada
 * @return  Matriz produto resultante da multiplicacao
 */
T** multiplicaI(T** A, T** B, int n);

/**
 * @brief   Funcao que multiplica duas matrizes quadradas de dimensao n x n de forma recursiva
 * @param   A Matriz de entrada
 * @param   B Matriz de entrada
 * @param   n Dimensao das matrizes de entrada
 * @return  Matriz produto resultante da multiplicacao
 */
T** multiplicaR(T** A, T** B, int n);
```

3.2 Entrada de dados e execução do programa

As operações de multiplicação serão feitas sobre duas matrizes quadradas, cada uma representada por um arquivo de texto que deverá ser lido pelo programa. Os arquivos são nomeados pelo nome da matriz seguido da sua dimensão e são estruturados da seguinte forma: (i) a primeira linha contém dois números inteiros separados por um espaço simples, representando as dimensões da matriz, e (ii) as linhas subsequentes contêm os valores dos elementos de cada linha da matriz, também separados por um espaço simples. A operação de multiplicação deverá ser feita entre as matrizes cujos nomes de arquivo iniciam com o o caractere `A` pelas matrizes cujos nomes de arquivo iniciam com o caractere `B`. Um exemplo de arquivo de entrada teria o nome `A2x2.txt` e o seguinte contéudo:

```
2 2
1 2
3 4
```

Neste projeto, será utilizado como entrada um conjunto de vinte arquivos representando matrizes quadradas de diferentes dimensões, sendo dez para as matrizes `A` e dez para as matrizes `B`. Os arquivos estão disponíveis para *download* através da Turma Virtual do SIGAA.

O programa principal deverá ser executado via linha de comando da seguinte forma:

```
$ ./multimat 2 4 8 16 32 64 128 256 512 1024
```

em que uma sequência de dez números inteiros (separados por espaço simples) representando as dimensões das matrizes quadradas que serão tratadas pelo programa é passada como argumentos de linha de comando. Como o algoritmo recursivo para multiplicação de matrizes trabalha apenas com

matrizes quadradas cujas dimensões são potências de base 2, então todos os números nessa sequência devem atender a essa restrição. Para cada uma das dimensões de matrizes i, o programa deverá:

1) ler o arquivo de entrada `Aixi.txt` referente à primeira matriz e armazenar seus elementos como uma matriz **alocada dinamicamente**;
2) ler o arquivo de entrada `Bixi.txt` referente à segunda matriz e armazenar seus elementos como uma matriz **alocada dinamicamente**;
3) chamar a função `multiplicaI` para executar o algoritmo iterativo para multiplicação das matrizes, medindo o tempo despendido para tal execução;
4) chamar a função `multiplicaR` para executar o algoritmo recursivo para multiplicação de matrizes, medindo o tempo despendido para tal execução;
5) gravar a matriz produto resultante da multiplicação em um arquivo `Cixi.txt`, que será uma das saídas a serem geradas pelo programa.

3.3 Experimentação

Para medir o tempo de execução de cada função, você poderá fazer uso da *Chrono Time Library*, uma biblioteca presente no padrão C++11. Documentação sobre essa biblioteca pode ser consultada na Internet através dos links http://www.cplusplus.com/reference/chrono/ e http://en.cppreference.com/w/cpp/chrono. Um bom exemplo do uso dessa biblioteca também pode ser encontrado na Internet, no endereço https://www.guyrutenberg.com/2013/01/27/using-stdchronohigh_resolution_clock-example/. É importante lembrar que, para que seja possível utilizar essa biblioteca, você deverá adicionar a diretiva de compilação `-std=c++11` ao compilar o seu programa.

No intuito de tornar os experimentos representativos do ponto de vista estatístico, será necessário executar cada um dos algoritmos de multiplicação em um total de vinte vezes. Os tempos de execução de cada algoritmo em cada uma dessas vinte execuções deverá ser registrado a fim de calcular os valores *máximo*, *mínimo* e *médio* dos tempos nas vinte execuções, além do *desvio padrão* observado. Para calcular o desvio padrão σ dos tempos de execução desses algoritmos, você poderá utilizar a seguinte equação:

$$\sigma = \sqrt{\frac{1}{20}\sum_{i=1}^{20}(t_i - t_M)^2}$$

em que t_i é o tempo registrado na i-ésima execução do algoritmo em questão e t_M é o tempo médio das vinte execuções. Um baixo desvio padrão indica que os pontos dos dados tendem a estar próximos da média do, enquanto que um alto desvio padrão indica que os pontos dos dados estão espalhados por uma ampla gama de valores.

Além do arquivo de saída referente à matriz produto resultante da multiplicação, o programa também deverá gerar automaticamente outros dois arquivos, a saber, `stats-ite.dat` e `stats-rec.dat`,

que contêm as estatísticas para as vinte execuções dos algoritmos iterativo e recursivo, respectivamente. Cada linha nesses arquivos corresponde a uma dimensão de matriz e cada uma das estatísticas, separados por espaço. Uma vez gerados esses arquivos, você deverá utilizar o programa `gnuplot` (http://www.gnuplot.info/) para gerar automaticamente **um único gráfico de linha** que mostrará o crescimento do tempo médio de execução dos algoritmos em função das dimensões das matrizes. Para possibilitar a geração do gráfico pelo `gnuplot`, você deverá criar um *script* de configuração próprio que deverá ser fornecido como entrada quando da execução do `gnuplot`, que utilizará os dados dos dois arquivos `stats-ide.dat` e `stats-rec.dat`. Consulte a documentação oficial disponível em http://gnuplot.info/documentation.html, bem como bons exemplos de uso e demonstração em http://alvinalexander.com/technology/gnuplot-charts-graphs-examples e http://gnuplot.sourceforge.net/demo/.

3.4 Controle sobre a pilha de execução (*stack*)

Para o compilador, a *pilha de execução* (também conhecida como *stack*) é uma região para armazenamento de dados temporários, como retornos de função e variáveis locais, sendo implementada como uma estrutura de dados do tipo pilha (daí o nome). A pilha de execução pode ser implementada através de registradores específicos na CPU (também chamada de *hardware stack*) ou pode ser implementada em RAM. Em qualquer dos casos, ela é bastante limitada.

O inimigo número #1 da pilha recursiva são as funções recursivas. Dependendo da função e da quantidade de chamadas, a pilha de execução pode ser consumida de forma muito rápida. Enquanto a chamada recursiva continua, a memoria é consumida ate sua exaustão, situação em que é lançado um erro de *stack overflow* (estouro de pilha). Por esse motivo, a recursão tradicional é na maioria das vezes menos eficiente que a iteração.

No caso da pilha de execução ser insuficiente para a execução do algoritmo recursivo, uma solução para contornar o problema (*workaround*) é aumentar do tamanho da pilha de chamadas:

1) No sistema operacional Windows, deve ser adicionada a diretiva de compilação `-Wl,-stack,<valor>`, de modo que a pilha de chamadas será alterada para o valor informado (em *kilobytes*) ao executar o programa. Por exemplo, para aumentar o tamanho da pilha para 1 GB, a diretiva de compilação seria `-Wl,-stack,1048576`.

2) No sistema operacional Linux, embora seja possível alterar a pilha de execução via programação através do uso de bibliotecas específicas, o mais prático é realizar a alteração diretamente sobre a variável de ambiente do sistema operacional que controla o tamanho da pilha de execução. Para isso, digite no Terminal do sistema Linux o comando

```
$ ulimit -s unlimited
```

e em seguida rode o programa. Ao término da execução do programa, digite no Terminal o comando

```
$ ulimit -s 8192
```

para fazer com que a pilha de execução volte ao valor padrão de 8 MB.

3.5 Relato

Uma vez realizada a tarefa de implementação e experimentação, você deverá elaborar um relatório técnico contendo, no mínimo, as seguintes seções:

Introdução Explicar o propósito do relatório.

Detalhes de implementação Descrever como foi feita a sua implementação em termos de arquivos, funções, etc. e como o programa funciona de uma maneira geral.

Metodologia Indicar o método adotado para realizar os experimentos com os algoritmos e analisar os resultados obtidos. Por exemplo, você deve apresentar a caracterização técnica do computador utilizado (processador, sistema operacional, quantidade de memória RAM), a linguagem de programação e a versão do compilador empregados, os cenários considerados, entre outras informações. Você também deverá descrever qual o procedimento adotado para gerar os resultados, como a comparação entre os algoritmos foi feita, etc.

Resultados Apresentar os resultados obtidos na forma de um gráfico de linha **e** tabelas. Para cada algoritmo deverá ser apresentado uma tabela contendo os tempos mínimo, médio e máximo obtidos para cada dimensão de matriz. Por sua vez, o gráfico de linha deverá exibir apenas os tempos médios obtidos nas execuções de cada algoritmo.

Discussão Discutir os resultados, ou seja, o que foi possível concluir através dos resultados obtidos através dos experimentos.

4 Orientações gerais

Você deverá atentar para as seguintes observações gerais no desenvolvimento deste trabalho:

1) Apesar da completa compatibilidade entre as linguagens de programação C e C++, seu código fonte **não** deverá conter recursos da linguagem C nem ser resultante de mescla entre as duas linguagens, o que é uma má prática de programação. Dessa forma, deverão ser utilizados **estritamente** recursos da linguagem C++.

2) Durante a compilação do seu código fonte, você deverá habilitar a exibição de mensagens de aviso (*warnings*), pois elas podem dar indícios de que o programa potencialmente possui problemas em sua implementação que podem se manifestar durante a sua execução.

3) Aplique boas práticas de programação. Codifique o programa de maneira legível (com indentação de código fonte, nomes consistentes, etc.) e documente-o adequadamente na forma de comentários. Anote ainda o código fonte para dar suporte à geração automática de documentação utilizando a ferramenta Doxygen (http://www.doxygen.org/). Consulte o documento extra disponibilizado na Turma Virtual do SIGAA com algumas instruções acerca do padrão de documentação e uso do Doxygen.

4) Busque desenvolver o seu programa com qualidade, garantindo que ele funcione de forma correta

e eficiente. Pense também nas possíveis entradas que poderão ser utilizadas para testar apropriadamente o seu programa e trate adequadamente possíveis entradas consideradas inválidas.

5) Lembre-se de aplicar boas práticas de modularização, em termos da implementação de diferentes funções e separação entre arquivos cabeçalho (`.h`) e corpo (`.cpp`).

6) A fim de auxiliar a compilação do seu projeto, você deverá **obrigatoriamente** construir um `Makefile` que faça uso da estrutura de diretórios apresentada anteriormente em aula.

7) Garanta o uso consistente de alocação dinâmica de memória. Para auxilia-lo nesta tarefa, você pode utilizar o Valgrind (http://valgrind.org/) como ferramenta de apoio para verificar se o seu programa apresenta vazamentos, *overflow* e outros problemas relacionados ao gerenciamento incorreto da memória.

5 Autoria e política de colaboração

Este trabalho deverá ser realizado individualmente ou em equipe de **no máximo** dois integrantes. O trabalho em cooperação entre estudantes da turma é estimulado, sendo admissível a discussão de ideias e estratégias. Contudo, tal interação não deve ser entendida como permissão para utilização de (parte de) código fonte de colegas, o que pode caracterizar situação de plágio. Trabalhos copiados em todo ou em parte de outros colegas ou da Internet serão sumariamente rejeitados e receberão nota zero.

Independentemente de o trabalho ser realizado individualmente ou em equipe, você deverá utilizar o sistema de controle de versões Git no desenvolvimento. Ao final, todos os arquivos de código fonte do repositório Git local deverão estar unificados em um repositório remoto Git hospedado em algum serviço da Internet, a exemplo do GitHub, Bitbucket, Gitlab ou outro de sua preferência. A fim de garantir a boa manutenção de seu repositório, configure corretamente o arquivo `.gitignore` em seu repositório Git.

6 Entrega

Você deverá submeter um único arquivo compactado no formato `.zip` contendo todos os códigos fonte resultantes da implementação deste exercício, sem erros de compilação e devidamente testados e documentados, **até as 23h59 do dia 1º de maio de 2017** através da opção *Tarefas* na Turma Virtual do SIGAA. Juntamente com os códigos fonte, o arquivo compactado deverá também conter o relatório escrito, preferencialmente em formato PDF. Por fim, você deverá ainda informar, no campo Comentários do formulário de submissão da tarefa, o endereço do repositório Git utilizado.

7 Avaliação

O trabalho será avaliado sob os seguintes critérios: (i) utilização correta dos conteúdos vistos anteriormente e nas aulas presenciais da disciplina; (ii) a corretude da execução do programa implementados, que deve apresentar saída em conformidade com a especificação e as entradas de dados fornecidas; (iii) a aplicação correta de boas práticas de programação, incluindo legibilidade, organização e documentação de código fonte; (iv) a qualidade do relatório técnico produzido, e; (v) a utilização correta do repositório Git, no qual deverá estar registrado todo o histórico da implementação por meio de *commits*. O trabalho possuirá nota máxima de 3,00 (três) pontos para a 1ª Unidade da disciplina, distribuídos de acordo com a seguinte composição:

Item avaliado	**Nota máxima**
Modularização adequada	0,20
Recursividade e iteratividade	
– Implementação da versão recursiva do algoritmo de multiplicação de matrizes	0,50
– Implementação da versão iterativa do algoritmo de multiplicação de matrizes	0,50
Uso correto do `Makefile`	0,15
Uso adequado de controle de versão com Git	0,15
Uso consistente de alocação dinâmica de memória	0,50
Qualidade do relatório escrito	1,00
Total	**3,00**

Por sua vez, o não cumprimento de algum dos critérios anteriormente especificados poderá resultar nos seguintes decréscimos, calculados sobre a nota total obtida até então:

Falta	**Decréscimo**
Programa apresenta erros de compilação, não executa ou apresenta saída incorreta	−70%
Falta de comentários no código fonte e/ou de documentação gerada com Doxygen	−10%
Implementação na linguagem C ou resultante de mistura entre as linguagens C e C++	−30%
Programa compila com mensagens de aviso (*warnings*)	−50%
Plagiarismo	−100%