# LABORATÓRIO I
# BIBLIOTECA OPENMP

Grupo: **Not Found**

Julia De Souza Dos Santos - 238166
Matheus Sales Oliveira - 203577

# SUMÁRIO

**SUMÁRIO** **2**

**PROBLEMA** **3**

1.1. DEFINIÇÃO 3

1.2. REQUISITOS 3

**OBJETIVO** **4**

**PROGRAMA** **4**

**RESULTADOS E CONCLUSÃO** **7**

**TUTORIAIS** **10**

5.1. COMPILANDO 10

5.1.1 COMPILANDO VIA TERMINAL DO LINUX 10

5.1.2. COMPILANDO VIA SCRIPT 11

5.1.2. COMPILANDO VIA MAKEFILE 11

5.1. EXECUTANDO O PROGRAMA 11

**REFERÊNCIAS** **12**

# 1. PROBLEMA

## 1.1. DEFINIÇÃO

Dadas três matrizes $A_{yxw}$, $B_{wxv}$, $C_{vx1}$, calcule a matriz $D_{yx1}$ tal que $D_{yx1}$= ($A_{yxw}$ X $B_{wxv}$) X $C_{vx1}$. Além disso, calcule a redução pela soma dos elementos na matriz $D_{yx1}$, isto é, a soma de todos os elementos em $D_{yx1}$.

## 1.2. REQUISITOS

- ✓ O programa deve ser escrito em linguagem C e no ambiente Linux.
- ✓ O código fonte deve estar disponível em um repositório no GitHub.
- ✓ O problema deverá ser resolvido utilizando a biblioteca OpenMP.
- ✓ As dimensões das matrizes são definidas pelas variáveis w, v e y (informadas pela linha de comando).
- ✓ Os dados das matrizes devem ser números reais, com até duas casas decimais, entre -10 e +10. Esses números devem ser gerados aleatoriamente.
- ✓ As matrizes devem ser alocadas dinamicamente em uma única etapa.
- ✓ Os valores de cada matriz devem estar armazenados sequencialmente em um arquivo do tipo “dat”.
- ✓ O programa deve ser executado da seguinte forma através do terminal:

```
./programa y w v arqA.dat arqB.dat arqC.dat arqD.dat
```

Onde:

- *./programa* → nome do programa que realiza a multiplicação das matrizes
- *y* → número de linhas da matriz A.
- *w* → número de colunas da matriz A e número de linhas da matriz B.
- *v* → número de colunas da matriz B e número de linhas da matriz C.
- *arqA.dat* → nome do arquivo que contém a matriz A.
- *arqB.dat* → nome do arquivo que contém a matriz B.
- *arqC.dat* → nome do arquivo que contém a matriz C.
- *arqD.dat* → nome do arquivo que contém a matriz resultante B.

# 2. OBJETIVO

O objetivo deste trabalho é avaliar a influência da paralelização de um programa sobre o seu tempo de execução. Nesse cenário, desenvolveu-se um programa na linguagem C que propõe uma solução para o problema apresentado acima utilizando a paralelização. Nesse cenário, o paralelismo foi implementado utilizando a biblioteca OpenMP que possibilita a criação de regiões paralelas através do uso de threads e de uma memória compartilhada.

# 3. PROGRAMA

O programa desenvolvido é composto por 6 funções e, dentre elas, apenas duas fazem o uso de paralelismo, visto que são as responsáveis pelas principais funcionalidades do software: *multiplicar_matrizes* e *reducao_pela_soma*. As outras 4 funções são responsáveis por gerar as matrizes e seus respectivos arquivos e pela coordenação de todas as funcionalidades (*main*).

**a)** `multiplicar_matrizes`: essa função é responsável por multiplicar duas matrizes e armazenar o resultado dessa multiplicação em uma matriz auxiliar que será o retornada pela função. Nela criou-se uma região paralela e definiu-se o número de threads que seriam utilizadas (`num_threads(4)`), quais variáveis seriam privadas (`private(linha, coluna, i, soma)`) e quais seriam compartilhadas entre as threads (`shared(matA, matB, linA, colB, linB)`).

Dentro dessa região foram realizados os *for* aninhados que percorrem as matrizes e realizam a multiplicação das mesmas. Assim, para fazer proveito do paralelismo nessa operação de multiplicação, utilizou-se a cláusula *for* para distribuir as iterações do laço entre as threads daquela região.

```
float* multiplicar_matrizes(float *matA, float *matB, unsigned int linA, unsigned int colA, unsigned int linB, unsigned int colB){
    float *matAux = (float *) malloc(linA * colB * sizeof(float));

    register unsigned int linha, coluna, i;
    register float soma=0;
    CRIA REGIÃO PARALELA
    #pragma omp parallel num_threads(4) private(linha, coluna, i, soma) shared(matA, matB, linA, colB, linB)
    {
        DIVIDE AS ITERAÇÕES DO FOR ENTRE AS THREADS
        #pragma omp for
        for(linha=0; linha<linA; linha++){
            for(coluna=0; coluna < colB; coluna++){
                soma=0;
                for(i=0; i < linB; i++){
                    soma += matA[linha*colA + i] * matB[i*colB + coluna];
                }
                matAux[linha*colB + coluna] = soma;
            }
        }
    }

    return matAux;
}
```

**b)** `reducao_pela_soma`: essa função é responsável por calcular a redução pela soma dos elementos armazenados na matriz resultante $D_{yx1}$. Para isto, a função recebe como parâmetros a matriz $D_{yx1}$ e seu número de linhas e de colunas e utiliza estes dados em um laço de repetição do tipo *for* aninhado para somar cada elemento da matriz a uma variável auxiliar chamada soma. Por fim, o programa retorna o valor armazenado na variável `soma`.

Nessa função foi utilizado o paralelismo no segundo for aninhado onde definiu-se 4 threads (utilizando a cláusula `num_threads(4)`) que, simultaneamente, acessam a variável soma e somam a essa variável o elemento armazenado na respectiva posição da matriz. Para isso, utilizou-se a cláusula `reduction(+:soma)` que permitirá que as threads realizem a operação de soma sobre a variável `soma` de forma simultânea.

```
double reducao_pela_soma(float *matriz, unsigned int linM, unsigned int colM){
    register int linha, coluna;
    register double soma = 0;

    for(linha=0; linha < linM; linha++){
        #pragma omp parallel for num_threads(4) reduction(+:soma)
        for(coluna=0; coluna < colM; coluna++){
            soma += matriz[linha*colM + coluna];
        }
    }

    return soma;
}
```

`OBS`: após realizamos os testes com os arquivos disponíveis no diretório *testes* do professor, vimos que o resultado da soma poderia exceder o limite de dígitos permitidos por um *float* o que, como consequência, fazia com que o

valor da soma fosse aproximado. Então, para evitar esse arredondamento e para que as saídas ficassem equivalentes, a soma foi criada como uma variável do tipo double.

**c)** `gerar_valores_aleatorios`: essa função é responsável por gerar os arquivos de dados de cada matriz. Dessa forma, para cada elemento da matriz, é gerado um número float aleatório entre -10 e +10 e esse é escrito no arquivo da mesma.

```
void gerar_valores_aleatorios (char *nomeArquivo, unsigned int linM, unsigned int colM){
     register unsigned int linha, coluna;
     register float rand_float, rand_limite;

     FILE * arquivo;
     arquivo = fopen(nomeArquivo, "w");

     for (linha=0; linha<linM;linha++){
        for (coluna=0; coluna<colM;coluna++){
            rand_float = (float)rand() / ((float) RAND_MAX + 1);
            rand_limite = (-10) + rand_float * (10 - (-10 + 1));

            fprintf(arquivo, "%.2f\n", rand_limite);
        }
     }

     fclose(arquivo);
}
```

**d)** `gerar_matriz`: essa função é responsável por ler o arquivo que contém todos os elementos da matriz (esse é o arquivo gerado na função anterior) e armazenar cada um dos dados escritos nele na matriz alocada dinamicamente.

```
float *gerar_matriz (char *nomedoArquivo, float *matriz, unsigned int linM, unsigned int colM){
    register unsigned int linha, coluna;
    FILE  *arquivo;
    arquivo= fopen(nomedoArquivo,"r");

    for (linha=0; linha<linM;linha++){
        for (coluna=0; coluna<colM;coluna++){
            fscanf(arquivo,"%f",&matriz[linha*colM +coluna]);
        }
    }

    fclose(arquivo);
    return matriz;
}
```

**e)** `gravar_matriz`: essa função é responsável por escrever o resultado da multiplicação das matrizes no arquivo de saída. Para isso, ela recebe como parâmetro o nome do arquivo que será criado, a matriz que será gravada no arquivo e o número de linhas e colunas desta matriz.

```
void gravar_matriz(char *nomeArquivo, float *matriz, unsigned int linM, unsigned int colM){
    FILE *arquivo;
    arquivo = fopen(nomeArquivo, "w");

    register unsigned int linha, coluna;

    for (linha=0; linha<linM; linha++){
        for (coluna=0; coluna<colM;coluna++){
            fprintf(arquivo,"%.2f\n", matriz[linha*colM +coluna]);
        }
    }

     fclose(arquivo);
}
```

**f)** `main`: essa função é responsável por coordenar todas as demais funções presentes no programa. Nela são feitas as alocações dinâmicas de todas as matrizes, os cálculos do tempo de execução das funções multiplicar_matrizes e reducao_pela_soma, a criação dos arquivos de dados de cada matriz assim como o preenchimento das mesmas. Além disso, é onde as funções que fazem uso da paralelização são chamadas.

Em relação aos detalhes dessa função, o nosso programa faz a alocação de dinâmica de 5 matrizes: matrizA, matrizB, matrizC, matrizD e matrizAB. Essa ultima é alocada com o mesmo número de linhas da matrizA e o mesmo número de colunas da matrizB e gerada a partir da multiplicação das matrizes A e B. Posteriormente, essa matriz AB é multiplicada com a matrizC para gerar a resultante matrizD.

```
matrizA  = (float *) malloc(y * w * sizeof(float));
matrizB  = (float *) malloc(w * v * sizeof(float));
matrizC  = (float *) malloc(v * 1 * sizeof(float));
matrizD  = (float *) malloc(y * 1 * sizeof(float));
matrizAB = (float *) malloc(y * v * sizeof(float));
```

```
matrizAB = multiplicar_matrizes(matrizA, matrizB, y, w, w, v);

matrizD = multiplicar_matrizes(matrizAB, matrizC, y, v, y, 1);
```

# 4. RESULTADOS E CONCLUSÃO

Para avaliar como a paralelização de um programada influencia no seu desempenho, foram realizados 5 testes com o código desenvolvido. Em cada teste, foi

atribuído o mesmo valor para as variáveis y, w, v que representam os números de linhas e colunas de cada matriz. Após definido os valores, o mesmo código foi executado porém, na primeira execução não considerou-se o paralelismo (compilado com o comando: `gcc -o programa programa.c`) mas, já na segunda execução, a paralelização foi considerada (compilado com o comando: `gcc -o programa -fopenmp programa.c`). Para cada execução, os respectivos valores do tempo de execução foram registrados na tabela abaixo.

| y | w | v | Tempo de Execução (sem paralelização) | Tempo de Execução (com paralelização) | Diferença % dos Tempos de Execução |
|---|---|---|---|---|---|
| 10 | 10 | 10 | 0,000005 | 0,000141 | 2720,00% |
| 100 | 100 | 100 | 0,002201 | 0,001190 | -45,93% |
| 1000 | 1000 | 1000 | 2,575367 | 1,044124 | -59,46% |
| 1500 | 1500 | 1500 | 11,134252 | 4,536605 | -59,26% |
| 2000 | 2000 | 2000 | 32,206737 | 11,202972 | -65,22% |
| | | | | | **-57,47%** |

*Figura 1 - Tabela de Análise*

Após o registro dos tempos de execução, foi calculado a diferença percentual entre os tempos obtidos executando o programa sem paralelização e executando o programa com paralelização. Esse cálculo percentual foi feito utilizando-se a seguinte fórmula:

$$\left(\frac{Tempo\ de\ Execução\ com\ Paralelização - Tempo\ de\ Execução\ sem\ Paralelização}{Tempo\ de\ Execução\ sem\ Paralelização}\right) \times 100$$

Através desse cálculo, foi possível notar que o programa paralelizado realizou as mesmas operações (com exceção para o valores de y=w=v=10) de forma muito mais rápida, obtendo um tempo de execução muito menor quando comparado com o tempo de execução do mesmo programa não paralelizado. Se realizamos uma média da porcentagem de melhora da performance para valores de y, w e v entre 100 e 2000, nota-se que houve uma diminuição de cerca de 57,47% no tempo de execução, ou seja, o programa paralelizado executa o mesmo código duas vezes mais rápido que o programa não-paralelizado.

Abaixo, encontram-se três gráficos que demonstram essa diferença entre os tempos de execução. Como há uma diferença considerável entre os tempos de execução para y=w=v=10 e y=w=v=100 em relação aos demais valores, foram criados gráficos exclusivos

para esses valores de x, y e w e, posteriormente, foi criado um gráfico que inclui todos os valores testados.

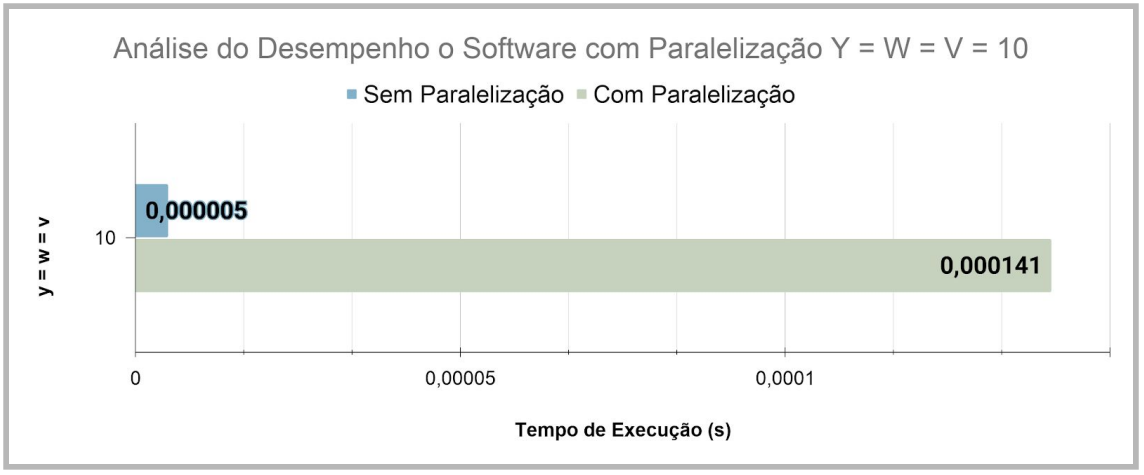


*Figura 2 - Gráfico de desempenho para y = w = v =10*

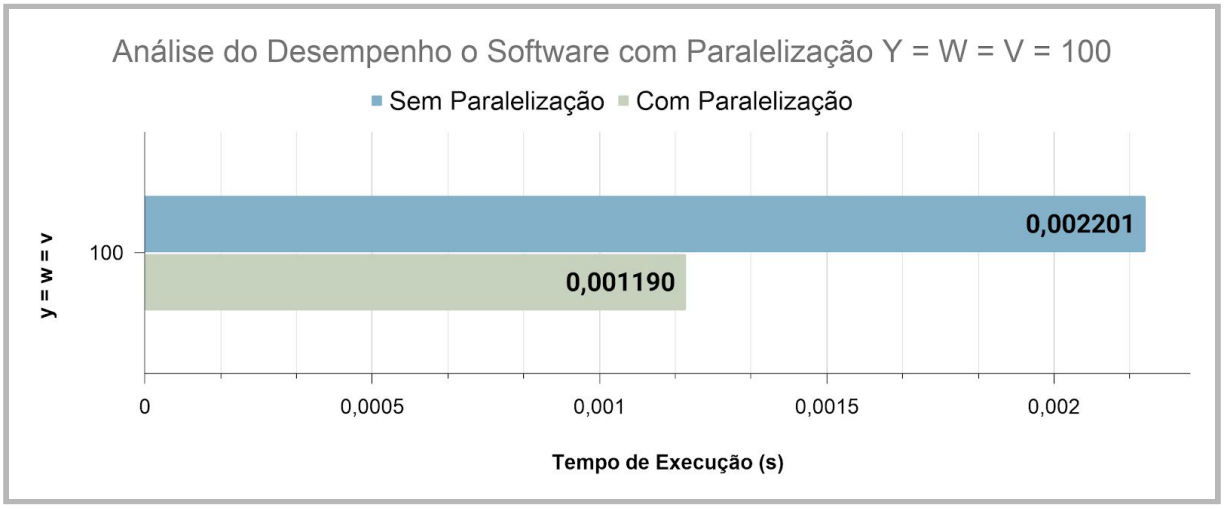


*Figura 3 - Gráfico de desempenho para y = w = v =100*

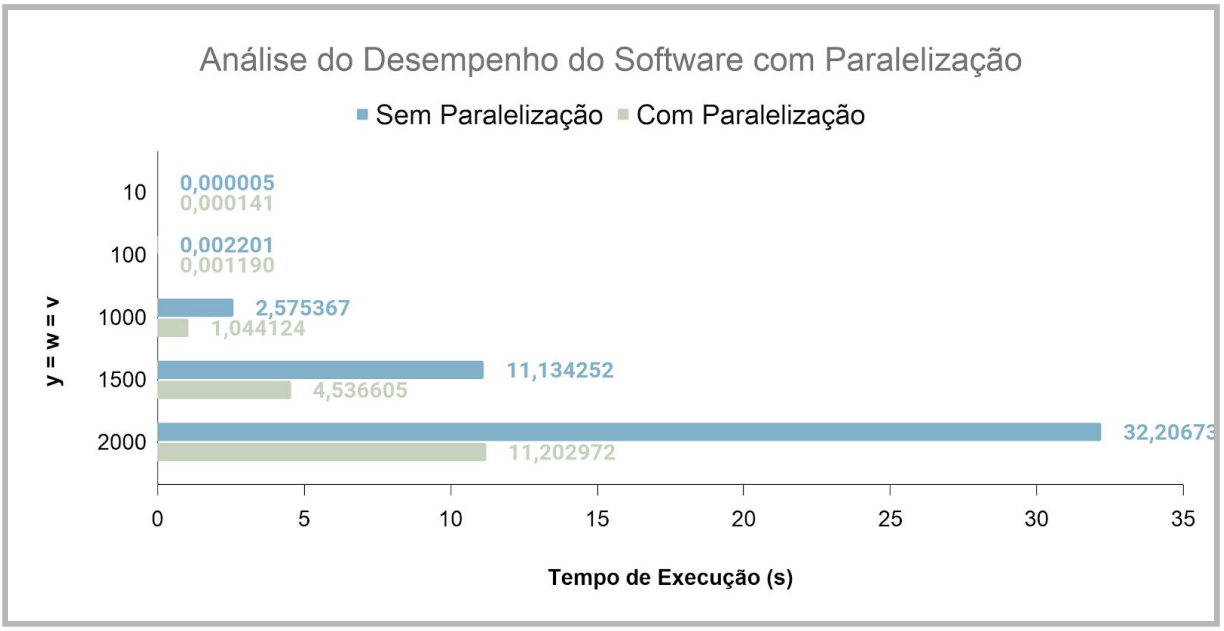


*Figura 4 - Gráfico de desempenho para todos os valores de y, w e v*

Assim, baseado nos tempos de execução obtidos, nos dados apresentados na tabela e na análise dos gráficos, pode-se concluir que a paralelização do programa diminuiu o seu tempo de execução, o que o torna mais rápido e mais eficiente.

# 5. TUTORIAIS

## 5.1. COMPILANDO

Há várias formas de compilar um programa escrito em linguagem C, abaixo encontram-se algumas destas formas.

### 5.1.1 COMPILANDO VIA TERMINAL DO LINUX

Para compilar o programa via terminal do Linux deve-se seguir os seguintes passos:

Digite o comando abaixo no terminal do Linux para gerar o arquivo executável:

```
gcc -o programa -fopenmp programa.c
```

Onde:

- `gcc` → executa o compilador.
- `-o programa` → define que o programa executável se chamará programa.

- `-fopenmp` → garante que a biblioteca OpenMP será incluída no programa durante a compilação.
- `programa.c` → nome do arquivo fonte escrito em linguagem C que será utilizado para gerar o arquivo executável.

#### 5.1.2. COMPILANDO VIA SCRIPT

Para automatizar o processo de compilação foi criado um script do tipo SH. O arquivo de script recebeu o nome de `script.sh` e, antes de executá-lo, é preciso **garantir que ele esteja na mesma pasta que o programa escrito em linguagem C**. Após isso, é necessário alterar a permissão de administrador ao script para garantir que não haja nenhuma falha durante a fase a compilação. Para conferir as devidas permissões ao script basta digitar e executar o código abaixo no terminal do Linux:

```
chmod +x  script.sh
```

Por fim, para executar o programa basta digitar e executar o seguinte código:

```
./script.sh
```

#### 5.1.2. COMPILANDO VIA MAKEFILE

Para automatizar o processo de compilação foi criado um arquivo makefile. O arquivo de makefile recebeu o nome padrão `makefile` e, antes de executá-lo, é preciso **garantir que ele esteja na mesma pasta que o programa escrito em linguagem C**. Após isso, é necessário digitar e executar o seguinte comando no terminal do Linux para compilar o programa:

```
make
```

## 5.1. EXECUTANDO O PROGRAMA

Para executar o programa, digite o comando abaixo no terminal:

```
./programa y w v arqA.dat arqB.dat arqC.dat arqD.dat
```

Onde:

- `./programa` → o programa gerado anteriormente.
- `y` → número de linhas da matriz A.
- `w` → número de colunas da matriz A e número de linhas da matriz B.
- `v` → número de colunas da matriz B e número de linhas da matriz C.
- `arqA.dat` → nome do arquivo que contém os valores da matriz $A_{yxw}$.

- `arqB.dat` → nome do arquivo que contém os valores da matriz $B_{wxv}$.
- `arqC.dat` → nome do arquivo que contém os valores da matriz $C_{vx1}$.
- `arqD.dat` → nome do arquivo que contém os valores da matriz resultante $D_{10x1}$.

### 5.3. GITHUB

O código fonte escrito em Linguagem C assim como o arquivo makefile e o script SH encontram-se no seguinte repositório no GitHub na pasta(diretório) OpenMP:

https://github.com/matheusaleso/PAD

# 6. REFERÊNCIAS

GRADVOHL, André Leon Sampaio. Teste de alocação de matrizes. GitHub. Disponível em: <https://gradvohl.github.io/alocaMatrizes/>. Acesso em: 22 de out. de 2020