Universidade Federal Fluminense

INSTITUTO DE COMPUTAÇÃO

ALAN GOMES

MATHEUS PIMENTEL

THIAGO FREITAS

ALGORITMO DE STRASSEN

IMPLEMENTAÇÃO PARALELA E ANÁLISE DE DESEMPENHO

NITERÓI

2022

ALGORITMO DE STRASSEN

IMPLEMENTAÇÃO PARALELA E ANÁLISE DE DESEMPENHO

Curso: Ciência da Computação

Alunos: Alan Gomes

Matheus Pimentel

Thiago Freitas

Professor: Maria Cristina Silva Boeres

Disciplina: Laboratório de Programação Paralela

Sumário

1. Introdução

O algoritmo de *Strassen* é utilizado para realizar a multiplicação de matrizes. Dadas duas matrizes quadradas A e B de dimensão n x n, considerando a matriz C = A x B. Através do algoritmo padrão a complexidade será $O(n^3)$, o Algoritmo de *Strassen* é um método recursivo que realiza a divisão de uma matriz em 4 submatrizes de dimensão n/2 x n/2 a cada passo recursivo. A complexidade é reduzida para $O(n^{2,8})$ o que pode não parecer muito mas faz diferença significativa para entradas grandes.

Neste trabalho busca-se realizar uma implementação paralela do algoritmo de *Strassen* e posteriormente comparar o algoritmo serial ao algoritmo paralelo. A implementação paralela será realizada usando as bibliotecas MPI e *OpenMP* e os resultados serão apresentados.

2. O Algoritmo de Strassen

Sejam A e B matrizes quadradas de ordem 2^n x 2^n, e seja C o produto dessas matrizes, para calcular esse produto particiona-se A, B e C em quatro submatrizes de mesmo tamanho:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

Então

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\
\mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}
\end{aligned}$$

Com essa construção, o número de multiplicações não é reduzido, sendo necessárias 8 multiplicações para calcular as matrizes $C_{i,j}$ que é a mesma quantidade necessária para realizar a multiplicação de forma usual. Definem-se então as matrizes:

$$\begin{aligned}
\mathbf{P}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{P}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{P}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{P}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{P}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{P}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{P}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

Que serão usadas para expressar $C_{i,j}$ em termos dos P_k. Devido a definição das matrizes P, pode-se eliminar uma multiplicação de matrizes e reduzir para 7 a sua quantidade (Uma multiplicação para cada P_k), expressando assim os $C_{i,j}$ como:

$$\mathbf{C}_{1,1} = \mathbf{P}_1 + \mathbf{P}_4 - \mathbf{P}_5 + \mathbf{P}_7$$
$$\mathbf{C}_{1,2} = \mathbf{P}_3 + \mathbf{P}_5$$
$$\mathbf{C}_{2,1} = \mathbf{P}_2 + \mathbf{P}_4$$
$$\mathbf{C}_{2,2} = \mathbf{P}_1 - \mathbf{P}_2 + \mathbf{P}_3 + \mathbf{P}_6$$

Ao todo, o Algoritmo de *Strassen* realiza 7 operações de multiplicação e 18 operações de soma/subtração, essas menos custosas.

Em implementações práticas do método de *Strassen*, a multiplicação de submatrizes de tamanho suficientemente pequenas é feita pelo método usual, pois nesses casos, o método usual se mostra mais eficiente.

3. Método Sequencial

No método sequencial a função recebe como entrada o parâmetro n, que indica a dimensão das matrizes e os ponteiros para as matrizes a serem multiplicadas.

Inicialmente verifica-se o critério de parada da recursão, neste caso estabelecido em 64, assim, toda matriz de tamanho menor que n=64 será multiplicada pelo método direto de multiplicação, esse corte mais alto permite melhor gerenciamento de memória e para matrizes de tamanho pequeno não há ganho se realizado pelo algoritmo de *Strassen*.

Logo depois, define-se m = n/2 e definem-se por meio de uma função auxiliar obter_submatriz, as submatrizes a, b, c, d, e, f, g, h, que são os quadrantes das matrizes que serão multiplicadas (A11, A12, A21, A22, B11, B12, B21, B22):

```
//Função strassen Serial
//entradas: tamanho n da matriz, ponteiro para matriz 1 e ponteiro para matriz 2
//saida: ponteiro para matriz produto da multiplicação
int** strassen(int n, int** mat1, int** mat2)
{

    //Condição de parada de recursão
    if (n <= 64)
    {
        return multiplica_matrizes(n, mat1, mat2);
    }

    int m = n / 2;

    //Obtendo submatrizes quadrantes:
    int** a = obter_submatriz(n, mat1, 0, 0);
    int** b = obter_submatriz(n, mat1, 0, m);
    int** c = obter_submatriz(n, mat1, m, 0);
    int** d = obter_submatriz(n, mat1, m, m);
    int** e = obter_submatriz(n, mat2, 0, 0);
    int** f = obter_submatriz(n, mat2, 0, m);
    int** g = obter_submatriz(n, mat2, m, 0);
    int** h = obter_submatriz(n, mat2, m, m);
```

Logo após serão obtidas as matrizes P1 a P7, para isso vamos usar matrizes intermediárias, nas quais vamos calcular as somas/subtrações e usar a função soma_matrizes para tal, posteriormente chama-se recursivamente o *Strassen*:

```
  //Obtendo as submatrizes P1 a P7

  int** bds = soma_matrizes(m, b, d, false);
  int** gha = soma_matrizes(m, g, h, true);
  int** p1 = strassen(m, bds, gha);
  libera_matriz(m, bds);
  libera_matriz(m, gha);
```

```
int** ada = soma_matrizes(m, a, d, true);
int** eha = soma_matrizes(m, e, h, true);
int** p2 = strassen(m, ada, eha);
libera_matriz(m, ada);
libera_matriz(m, eha);

int** acs = soma_matrizes(m, a, c, false);
int** efa = soma_matrizes(m, e, f, true);
int** p3 = strassen(m, acs, efa);
libera_matriz(m, acs);
libera_matriz(m, efa);

int** aba = soma_matrizes(m, a, b, true);
int** p4 = strassen(m, aba, h);
libera_matriz(m, aba);
libera_matriz(m, b);

int** fhs = soma_matrizes(m, f, h, false);
int** p5 = strassen(m, a, fhs);
libera_matriz(m, fhs);
libera_matriz(m, a);
libera_matriz(m, f);
libera_matriz(m, h);
```

```
  int** ges = soma_matrizes(m, g, e, false);
  int** p6 = strassen(m, d, ges);
  libera_matriz(m, ges);
  libera_matriz(m, g);

  int** cda = soma_matrizes(m, c, d, true);
  int** p7 = strassen(m, cda, e);
  libera_matriz(m, cda);
  libera_matriz(m, c);
  libera_matriz(m, d);
  libera_matriz(m, e);
```

Após obtidas as submatrizes P1 a P7, calcula-se as submatrizes C11, C12, C21, C22 que irão ser reagrupadas na matriz produto, novamente fez-se uso de matrizes auxiliares intermediárias para organizar somas/subtrações:

```
//Matrizes P1 - P7 calculadas, agora vamos obter as quatro submatrizes C11, C12, C21, C22

int** s1s2a = soma_matrizes(m, p1, p2, true);
int** s6s4s = soma_matrizes(m, p6, p4, false);
int** c11 = soma_matrizes(m, s1s2a, s6s4s, true);
libera_matriz(m, s1s2a);
libera_matriz(m, s6s4s);
libera_matriz(m, p1);

int** c12 = soma_matrizes(m, p4, p5, true);
libera_matriz(m, p4);

int** c21 = soma_matrizes(m, p6, p7, true);
libera_matriz(m, p6);

int** s2s3s = soma_matrizes(m, p2, p3, false);
int** s5s7s = soma_matrizes(m, p5, p7, false);
int** c22 = soma_matrizes(m, s2s3s, s5s7s, true);
libera_matriz(m, s2s3s);
libera_matriz(m, s5s7s);
libera_matriz(m, p2);
libera_matriz(m, p3);
libera_matriz(m, p5);
libera_matriz(m, p7);
```

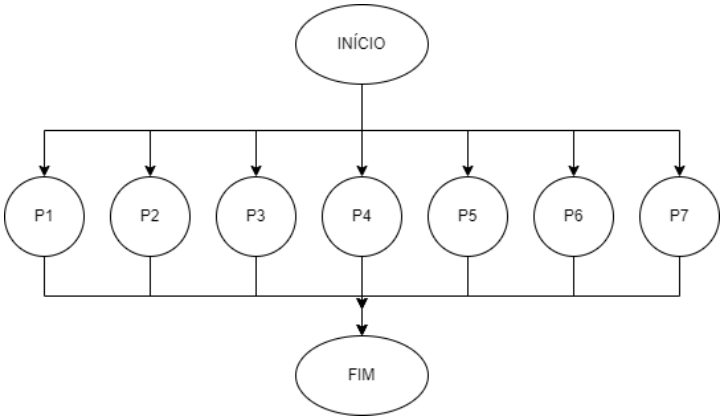
Finalizando esta etapa, a matriz C será gerada pelo agrupamento das submatrizes C11, C12, C21, C22:

```
 //Após obtidas as submatrizes serão combinadas para gerar a matriz produto
 int** prod = combina_matrizes(m, c11, c12, c21, c22);

 libera_matriz(m, c11);
 libera_matriz(m, c12);
 libera_matriz(m, c21);
 libera_matriz(m, c22);

 return prod;
```

4. Método Paralelo

A ideia central do algoritmo consiste em dividir esses cálculos entre os processos disponíveis para acelerar a obtenção do resultado. A implementação serial de *Strassen* também foi usada neste processo, em que temos uma função principal que vai se paralelizar e chamar recursivamente as funções seriais, assim sendo, cada um dos processos estará ocupado realizando as recursões para o cálculo final de uma das matrizes P.

Graficamente falando esta é a divisão proposta:



Por razões de quantidade de processadores disponíveis esta divisão seria mais eficiente com 8 cores, permitindo que cada processo assuma a operação de uma das matrizes.

O programa foi também preparado para trabalhar com menos processadores mas nestes casos 1 processo ficará encarregado de calcular mais de uma das matrizes P propostas, o que deve aumentar o tempo para obter o resultado.

5. Implementação Paralela (MPI)

A implementação em MPI foi realizada fazendo uso de funções auxiliares, dentre elas uma que aloca uma matriz na memória, uma que libera a matriz, função de soma, uma função direta de multiplicação (serial) e a função de *Strassen* sequencial.

Cada imagem será explicada a seguir:

```
void strassen(int n, int** mat1, int** mat2, int**& prod, int rank, int num_process)
{

    //Condição base de strassen
    if (n == 1)
    {
        prod = alocar_matriz(1);
        prod[0][0] = mat1[0][0] * mat2[0][0];
    }

    int m = n / 2;

    //Obtendo submatrizes quadrantes
    int** a = obter_submatriz(n, mat1, 0, 0);
    int** b = obter_submatriz(n, mat1, 0, m);
    int** c = obter_submatriz(n, mat1, m, 0);
    int** d = obter_submatriz(n, mat1, m, m);
    int** e = obter_submatriz(n, mat2, 0, 0);
    int** f = obter_submatriz(n, mat2, 0, m);
    int** g = obter_submatriz(n, mat2, m, 0);
    int** h = obter_submatriz(n, mat2, m, m);

    //Alocando P1 - P7
    int** p1 = alocar_matriz(m);
    int** p2 = alocar_matriz(m);
    int** p3 = alocar_matriz(m);
    int** p4 = alocar_matriz(m);
    int** p5 = alocar_matriz(m);
    int** p6 = alocar_matriz(m);
    int** p7 = alocar_matriz(m);
```

Primeiramente se realiza o teste se n é 1, caso seja basicamente multiplica-se os termos da matriz, essa também é uma condição base do *Strassen* para evitar a divisão das matrizes. Caso não seja, define-se m = n/2 e, com auxílio da função obter_submatriz, obtém-se as submatrizes dos quadrantes das duas matrizes a serem multiplicadas. A seguir a função aloca as submatrizes p1 a p7.

Logo após realiza-se um teste para verificar a quantidade de processadores/processos disponíveis, dependendo da quantidade, a distribuição da paralelização será definida:

Se existe apenas 1 processador, rank 0 vai calcular as matrizes p1 a p7:

```
if(num_process == 1){
    if (rank == 0){
        //Calculando P1
        int** bds = soma_matrizes(m, b, d, false);
        int** gha = soma_matrizes(m, g, h, true);
        p1 = strassen(m, bds, gha);
        //Calculando P2
        int** ada = soma_matrizes(m, a, d, true);
        int** eha = soma_matrizes(m, e, h, true);
        p2 = strassen(m, ada, eha);
        //Calculando P3
        int** acs = soma_matrizes(m, a, c, false);
        int** efa = soma_matrizes(m, e, f, true);
        p3 = strassen(m, acs, efa);
        //Calculando P4
        int** aba = soma_matrizes(m, a, b, true);
        p4 = strassen(m, aba, h);
        //Calculando P5
        int** fhs = soma_matrizes(m, f, h, false);
        p5 = strassen(m, a, fhs);
        //Calculando P6
        int** ges = soma_matrizes(m, g, e, false);
        p6 = strassen(m, d, ges);
        //Calculando P7
        int** cda = soma_matrizes(m, c, d, true);
        p7 = strassen(m, cda, e);
```

Se o número de processadores for 2, o processo 0 inicia tudo e aguarda que o processo 1 realize os cálculos das matrizes P, posteriormente o processo 0 finaliza:

```
else if(num_process == 2){
    //Processo 0 verifica e recebe resultados dos outros processos
    if (rank == 0)
    {
        cout << "Numero de processos: " << num_process << endl;
        MPI_Recv(&(p1[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p2[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p3[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p4[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p5[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p6[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p7[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    //processo 1 calcula P1 - P7
    if (rank == 1)
    {
        //Calculando P1
        int** bds = soma_matrizes(m, b, d, false);
        int** gha = soma_matrizes(m, g, h, true);
        p1 = strassen(m, bds, gha);
        //Calculando P2
        int** ada = soma_matrizes(m, a, d, true);
        int** eha = soma_matrizes(m, e, h, true);
        p2 = strassen(m, ada, eha);
        //Calculando P3
        int** acs = soma_matrizes(m, a, c, false);
        int** efa = soma_matrizes(m, e, f, true);
        p3 = strassen(m, acs, efa);
        //Calculando P4
        int** aba = soma_matrizes(m, a, b, true);
        p4 = strassen(m, aba, h);
        //Calculando P5
        int** fhs = soma_matrizes(m, f, h, false);
        p5 = strassen(m, a, fhs);
        //Calculando P6
        int** ges = soma_matrizes(m, g, e, false);
        p6 = strassen(m, d, ges);
        //Calculando P7
        int** cda = soma_matrizes(m, c, d, true);
        p7 = strassen(m, cda, e);
```

Quando há 4 processadores, distribui-se da seguinte forma:

O processo 0 prepara e aguarda os outros processos:

```
else if(num_process == 4){
    //Processo 0 verifica e recebe resultados dos outros processos
    if (rank == 0)
    {
        cout << "Numero de processos: " << num_process << endl;
        MPI_Recv(&(p1[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p2[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p3[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p4[0][0]), m * m, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p5[0][0]), m * m, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p6[0][0]), m * m, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p7[0][0]), m * m, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
```

O processo 1 calcula P1, P2 e P3:

```
    //processo 1 calcula P1, P2 e P3
    if (rank == 1)
    {
        //Calculando P1
        int** bds = soma_matrizes(m, b, d, false);
        int** gha = soma_matrizes(m, g, h, true);
        p1 = strassen(m, bds, gha);
        //Calculando P2
        int** ada = soma_matrizes(m, a, d, true);
        int** eha = soma_matrizes(m, e, h, true);
        p2 = strassen(m, ada, eha);
        //Calculando P3
        int** acs = soma_matrizes(m, a, c, false);
        int** efa = soma_matrizes(m, e, f, true);
        p3 = strassen(m, acs, efa);

        libera_matriz(m, bds);
        libera_matriz(m, gha);
        libera_matriz(m, ada);
        libera_matriz(m, eha);
        libera_matriz(m, acs);
        libera_matriz(m, efa);

        MPI_Send(&(p1[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&(p2[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&(p3[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
```

O processo 2 calcula P4 e P5:

```
    //processo 2 calcula P4 e P5
    if (rank == 2)
    {
        //Calculando P4
        int** aba = soma_matrizes(m, a, b, true);
        p4 = strassen(m, aba, h);
        //Calculando P5
        int** fhs = soma_matrizes(m, f, h, false);
        p5 = strassen(m, a, fhs);

        libera_matriz(m, aba);
        libera_matriz(m, fhs);

        MPI_Send(&(p4[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&(p5[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, b);
    libera_matriz(m, a);
    libera_matriz(m, f);
    libera_matriz(m, h);
```

E por fim, o processo 3 calcula P6 e P7:

```
    //Processo 3 calcula P6 e P7
    if (rank == 3)
    {
        //Calculando P6
        int** ges = soma_matrizes(m, g, e, false);
        p6 = strassen(m, d, ges);
        //Calculando P7
        int** cda = soma_matrizes(m, c, d, true);
        p7 = strassen(m, cda, e);

        libera_matriz(m, ges);
        libera_matriz(m, cda);

        MPI_Send(&(p6[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&(p7[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, g);
    libera_matriz(m, c);
    libera_matriz(m, d);
    libera_matriz(m, e);
```

Quando há disponibilidade de 6 processadores, distribui-se da seguinte forma:

O processo 0 prepara e aguarda os outros processos:

```
else if(num_process == 6){
    //Processo 0 verifica e recebe resultados dos outros processos
    if (rank == 0)
    {
        cout << "Numero de processos: " << num_process << endl;
        MPI_Recv(&(p1[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p2[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p3[0][0]), m * m, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p4[0][0]), m * m, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p5[0][0]), m * m, MPI_INT, 4, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p6[0][0]), m * m, MPI_INT, 5, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p7[0][0]), m * m, MPI_INT, 5, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
```

O processo 1 calcula P1 e P2:

```
    //processo 1 calcula P1 e P2
    if (rank == 1)
    {
        //Calculando P1
        int** bds = soma_matrizes(m, b, d, false);
        int** gha = soma_matrizes(m, g, h, true);
        p1 = strassen(m, bds, gha);
        //Calculando P2
        int** ada = soma_matrizes(m, a, d, true);
        int** eha = soma_matrizes(m, e, h, true);
        p2 = strassen(m, ada, eha);

        libera_matriz(m, bds);
        libera_matriz(m, gha);
        libera_matriz(m, ada);
        libera_matriz(m, eha);

        MPI_Send(&(p1[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&(p2[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
```

O processo 2 calcula P3, o processo 3 calcula P4 e o processo 4 calcula P5:

```
    //processo 2 calcula P3
    if (rank == 2)
    {
        int** acs = soma_matrizes(m, a, c, false);
        int** efa = soma_matrizes(m, e, f, true);
        p3 = strassen(m, acs, efa);
        libera_matriz(m, acs);
        libera_matriz(m, efa);
        MPI_Send(&(p3[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    //processo 3 calcula P4
    if (rank == 3)
    {
        int** aba = soma_matrizes(m, a, b, true);
        p4 = strassen(m, aba, h);
        libera_matriz(m, aba);
        MPI_Send(&(p4[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, b);

    //Processo 4 calcula P5
    if (rank == 4)
    {
        int** fhs = soma_matrizes(m, f, h, false);
        p5 = strassen(m, a, fhs);
        libera_matriz(m, fhs);
        MPI_Send(&(p5[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, a);
    libera_matriz(m, f);
    libera_matriz(m, h);
```

Então o processo 5 calcula P6 e P7:

```
    //Processo 5 calcula P6 e P7
    if (rank == 5)
    {
        //Calculando P6
        int** ges = soma_matrizes(m, g, e, false);
        p6 = strassen(m, d, ges);
        //Calculando P7
        int** cda = soma_matrizes(m, c, d, true);
        p7 = strassen(m, cda, e);

        libera_matriz(m, ges);
        libera_matriz(m, cda);

        MPI_Send(&(p6[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&(p7[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, g);
    libera_matriz(m, c);
    libera_matriz(m, d);
    libera_matriz(m, e);
```

Caso existam 8 ou mais processadores, a divisão otimizada como no desenho será realizada, neste caso o processo 0 aguarda enquanto cada processo calcula uma das matrizes P_k :

```
else if(num_process >=8){
    //Processo 0 verifica e recebe resultados dos outros processos
    if (rank == 0)
    {
        cout << "Numero de processos: " << num_process << endl;
        MPI_Recv(&(p1[0][0]), m * m, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p2[0][0]), m * m, MPI_INT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p3[0][0]), m * m, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p4[0][0]), m * m, MPI_INT, 4, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p5[0][0]), m * m, MPI_INT, 5, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p6[0][0]), m * m, MPI_INT, 6, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&(p7[0][0]), m * m, MPI_INT, 7, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    //processo 1 calcula P1
    if (rank == 1)
    {
        int** bds = soma_matrizes(m, b, d, false);
        int** gha = soma_matrizes(m, g, h, true);
        p1 = strassen(m, bds, gha);
        libera_matriz(m, bds);
        libera_matriz(m, gha);
        MPI_Send(&(p1[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    //processo 2 calcula P2
    if (rank == 2)
    {
        int** ada = soma_matrizes(m, a, d, true);
        int** eha = soma_matrizes(m, e, h, true);
        p2 = strassen(m, ada, eha);
        libera_matriz(m, ada);
        libera_matriz(m, eha);
        MPI_Send(&(p2[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
```

```
    //processo 3 calcula P3
    if (rank == 3)
    {
        int** acs = soma_matrizes(m, a, c, false);
        int** efa = soma_matrizes(m, e, f, true);
        p3 = strassen(m, acs, efa);
        libera_matriz(m, acs);
        libera_matriz(m, efa);
        MPI_Send(&(p3[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    //processo 4 calcula P4
    if (rank == 4)
    {
        int** aba = soma_matrizes(m, a, b, true);
        p4 = strassen(m, aba, h);
        libera_matriz(m, aba);
        MPI_Send(&(p4[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, b);

    //Processo 5 calcula P5
    if (rank == 5)
    {
        int** fhs = soma_matrizes(m, f, h, false);
        p5 = strassen(m, a, fhs);
        libera_matriz(m, fhs);
        MPI_Send(&(p5[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, a);
    libera_matriz(m, f);
    libera_matriz(m, h);
```

```
    //Processo 6 calcula P6
    if (rank == 6)
    {
        int** ges = soma_matrizes(m, g, e, false);
        p6 = strassen(m, d, ges);
        libera_matriz(m, ges);
        MPI_Send(&(p6[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, g);

    //Processo 7 calcula P7
    if (rank == 7)
    {
        int** cda = soma_matrizes(m, c, d, true);
        p7 = strassen(m, cda, e);
        libera_matriz(m, cda);
        MPI_Send(&(p7[0][0]), m * m, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    libera_matriz(m, c);
    libera_matriz(m, d);
    libera_matriz(m, e);
```

Se um número incompatível de processadores (3, 5, 7) for fornecido, o programa aborta e imprime a mensagem de erro:

```
else{
    cout << "Quantidade de processadores incompatível" << endl;
    abort;
}
```

Independentemente da quantidade de processadores, uma barreira vai garantir a sincronização dos dados, e após finalizados os cálculos das matrizes P, o processo 0 finaliza obtendo as submatrizes C11, C12, C21 e C22 e posteriormente combinando-as para obter a matriz C final:

```
MPI_Barrier(MPI_COMM_WORLD); //BARREIRA DE SINCRONIA

//Após obter resultados, P0 finaliza
if (rank == 0)
{
    //Criando as submatrizes C11, C12, C21, C22
    int** s1s2a = soma_matrizes(m, p1, p2, true);
    int** s6s4s = soma_matrizes(m, p6, p4, false);
    int** c11 = soma_matrizes(m, s1s2a, s6s4s, true);
    libera_matriz(m, s1s2a);
    libera_matriz(m, s6s4s);

    int** c12 = soma_matrizes(m, p4, p5, true);

    int** c21 = soma_matrizes(m, p6, p7, true);

    int** s2s3s = soma_matrizes(m, p2, p3, false);
    int** s5s7s = soma_matrizes(m, p5, p7, false);
    int** c22 = soma_matrizes(m, s2s3s, s5s7s, true);
    libera_matriz(m, s2s3s);
    libera_matriz(m, s5s7s);

    // Submatrizes C criadas agora serão combinadas para gerar a matriz produto resultante da multiplicação
    prod = combina_matrizes(m, c11, c12, c21, c22);

    libera_matriz(m, c11);
    libera_matriz(m, c12);
    libera_matriz(m, c21);
    libera_matriz(m, c22);
}

libera_matriz(m, p1);
libera_matriz(m, p2);
libera_matriz(m, p3);
libera_matriz(m, p4);
libera_matriz(m, p5);
libera_matriz(m, p6);
libera_matriz(m, p7);
```

6. Implementação Paralela (OpenMP)

A implementação em *OpenMP* faz uso das mesmas funções auxiliares usadas nas implementações anteriores, dentre elas uma que aloca uma matriz na memória, uma que libera a matriz, função de soma, uma função direta de multiplicação (implementada paralelamente). A vantagem do *OpenMP* é que a chamada recursiva pode ser executada paralelamente e tudo se mantém organizado já que no *OpenMP* temos blocos de paralelização.

A função começa com a condição de parada, caso a condição de parada seja satisfeita chama-se o método multiplica_matrizes e logo a seguir define-se m=n/2 e as submatrizes necessárias no processo:

```
int** strassen(int n, int** mat1, int** mat2)
{
    //Condição final de recursão
    if (n <= 64)
    {
        return multiplica_matrizes(n, mat1, mat2);
    }

    int m = n / 2;

    //Obtendo submatrizes quadrantes
    int** a = obter_submatriz(n, mat1, 0, 0);
    int** b = obter_submatriz(n, mat1, 0, m);
    int** c = obter_submatriz(n, mat1, m, 0);
    int** d = obter_submatriz(n, mat1, m, m);
    int** e = obter_submatriz(n, mat2, 0, 0);
    int** f = obter_submatriz(n, mat2, 0, m);
    int** g = obter_submatriz(n, mat2, m, 0);
    int** h = obter_submatriz(n, mat2, m, m);
```

Organizando em *tasks* (tarefas), temos a obtenção das matrizes P1 a P7 paralelamente:

```
//Obtendo P1
int** p1;
#pragma omp task shared(p1)
{
    int** bds = soma_matrizes(m, b, d, false);
    int** gha = soma_matrizes(m, g, h, true);
    p1 = strassen(m, bds, gha);
    libera_matriz(m, bds);
    libera_matriz(m, gha);
}

//Obtendo P2
int** p2;
#pragma omp task shared(p2)
{
    int** ada = soma_matrizes(m, a, d, true);
    int** eha = soma_matrizes(m, e, h, true);
    p2 = strassen(m, ada, eha);
    libera_matriz(m, ada);
    libera_matriz(m, eha);
}
```

```
//Obtendo P3
int** p3;
#pragma omp task shared(p3)
{
    int** acs = soma_matrizes(m, a, c, false);
    int** efa = soma_matrizes(m, e, f, true);
    p3 = strassen(m, acs, efa);
    libera_matriz(m, acs);
    libera_matriz(m, efa);
}

//Obtendo P4
int** p4;
#pragma omp task shared(p4)
{
    int** aba = soma_matrizes(m, a, b, true);
    p4 = strassen(m, aba, h);
    libera_matriz(m, aba);
}

//Obtendo P5
int** p5;
#pragma omp task shared(p5)
{
    int** fhs = soma_matrizes(m, f, h, false);
    p5 = strassen(m, a, fhs);
    libera_matriz(m, fhs);
}
```

```
//Obtendo P6
int** p6;
#pragma omp task shared(p6)
{
    int** ges = soma_matrizes(m, g, e, false);
    p6 = strassen(m, d, ges);
    libera_matriz(m, ges);
}

//Obtendo P7
int** p7;
#pragma omp task shared(p7)
{
    int** cda = soma_matrizes(m, c, d, true);
    p7 = strassen(m, cda, e);
    libera_matriz(m, cda);
}

//Aguardar threads processarem tarefas
#pragma omp taskwait

libera_matriz(m, a);
libera_matriz(m, b);
libera_matriz(m, c);
libera_matriz(m, d);
libera_matriz(m, e);
libera_matriz(m, f);
libera_matriz(m, g);
libera_matriz(m, h);
```

Ao final, foi usado o *taskwait* para garantir que todas as tarefas tenham sido executadas para continuação.

Após obtidas as matrizes P1 a P7, obtém-se as submatrizes C11, C12, C21, C22. Aqui faz-se uso do paralelismo mais uma vez dividindo em *tasks* e usando o *taskwait* para aguardar que cada thread execute a sua tarefa:

```
//Matrizes P1-P7 obtidas, agora vamos gerar as submatrizes C11, C12, C21, C22

//Gerando C11
int** c11;
#pragma omp task shared(c11)
{
    int** s1s2a = soma_matrizes(m, p1, p2, true);
    int** s6s4s = soma_matrizes(m, p6, p4, false);
    c11 = soma_matrizes(m, s1s2a, s6s4s, true);
    libera_matriz(m, s1s2a);
    libera_matriz(m, s6s4s);
}

//Gerando C12
int** c12;
#pragma omp task shared(c12)
{
    c12 = soma_matrizes(m, p4, p5, true);
}

//Gerando C21
int** c21;
#pragma omp task shared(c21)
{
    c21 = soma_matrizes(m, p6, p7, true);
}

//Gerando C22
int** c22;
#pragma omp task shared(c22)
{
    int** s2s3s = soma_matrizes(m, p2, p3, false);
    int** s5s7s = soma_matrizes(m, p5, p7, false);
    c22 = soma_matrizes(m, s2s3s, s5s7s, true);
    libera_matriz(m, s2s3s);
    libera_matriz(m, s5s7s);
}

//Aguardar threads processarem tarefas
#pragma omp taskwait
```

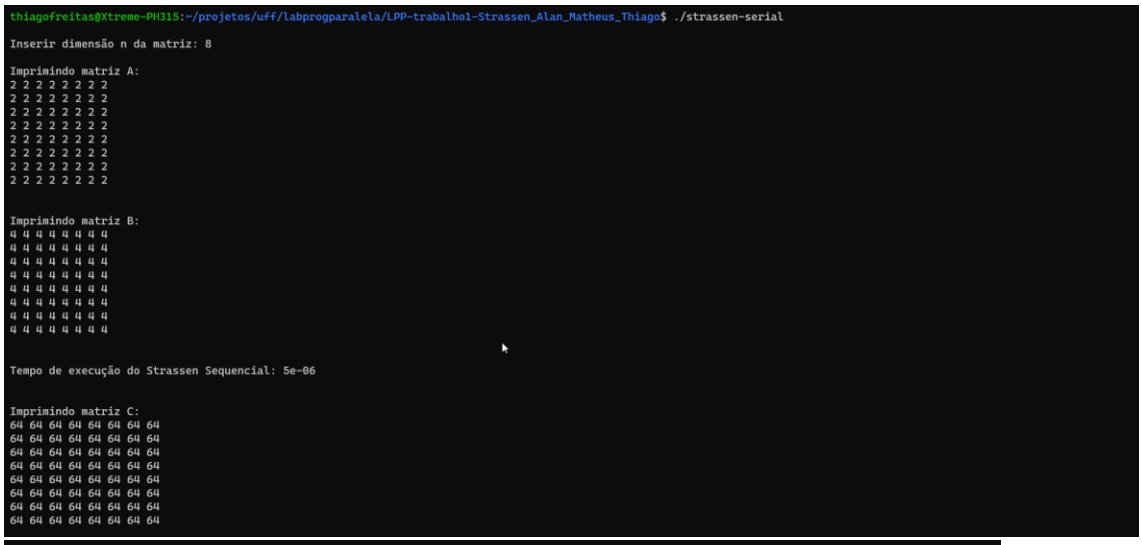
Finalmente define-se a matriz C pela combinação das submatrizes C11, C12, C21, C22 e retorna-se a matriz prod:

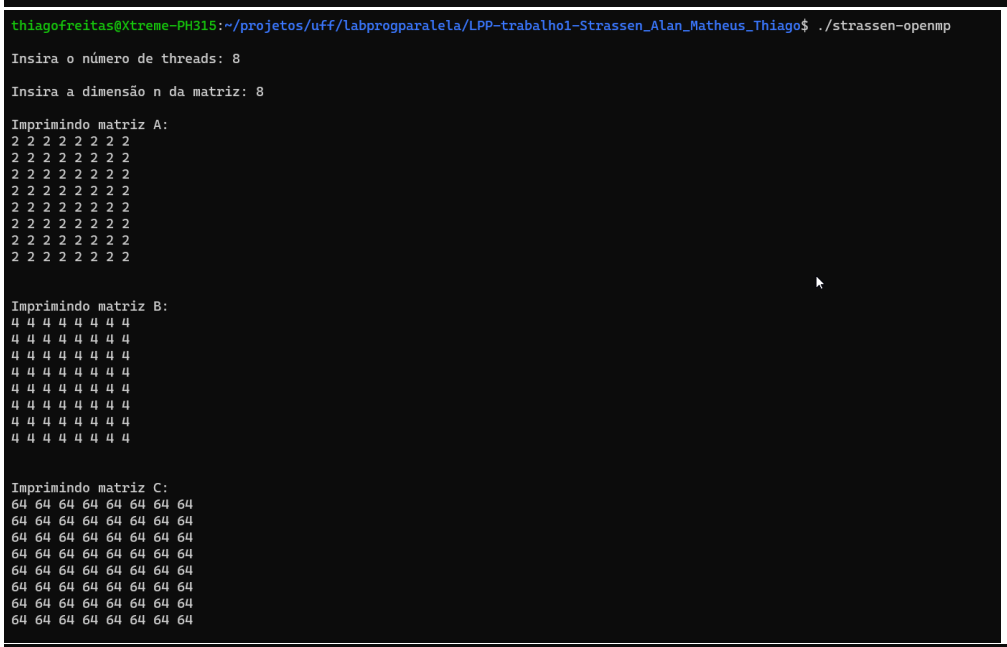
```
    //Aguardar threads processarem tarefas
    #pragma omp taskwait

    libera_matriz(m, p1);
    libera_matriz(m, p2);
    libera_matriz(m, p3);
    libera_matriz(m, p4);
    libera_matriz(m, p5);
    libera_matriz(m, p6);
    libera_matriz(m, p7);

    //Agora vamos obter a matriz produto pela combinação das submatrizes C
    int** prod = combina_matrizes(m, c11, c12, c21, c22);

    libera_matriz(m, c11);
    libera_matriz(m, c12);
    libera_matriz(m, c21);
    libera_matriz(m, c22);

    return prod;
}
```

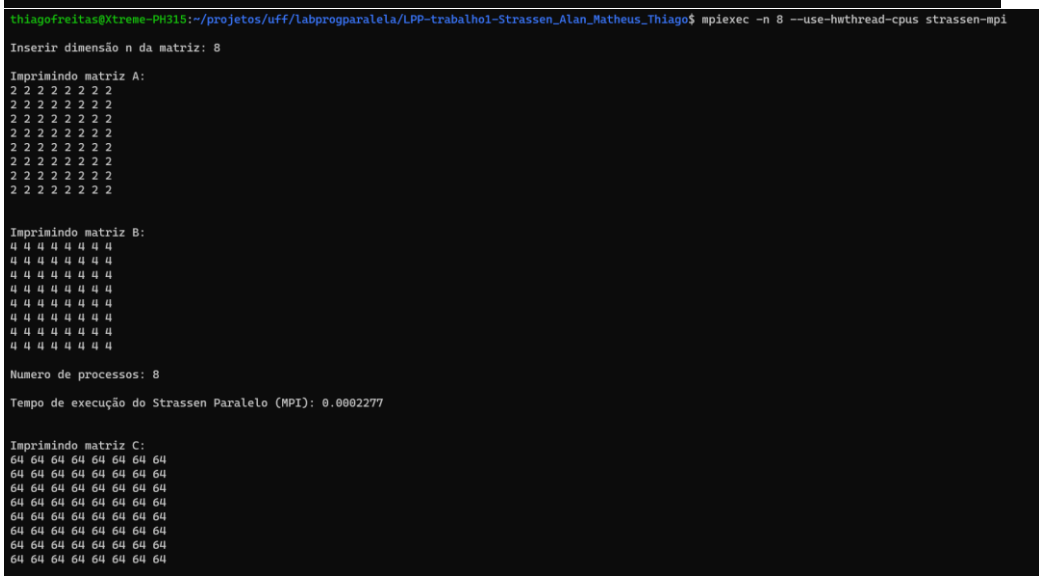
7. Experimentos Computacionais

Para o experimento foi utilizado uma máquina com a seguinte configuração: Processador Intel Core i7 9750h (6 núcleos/12 threads), 32GB de Ram DDR4, SSD Nvme 500GB, Sistema operacional Windows (porém os testes foram executados em ambiente Linux, por meio do Ubuntu instalado via WSL – Subsistema do Windows para Linux) É importante ressaltar que a máquina virtual do WSL dispõe de 50% da memória principal por limitação da tecnologia, logo, considera-se que os testes rodaram em 16GB de Ram.

Para testar se o código estava multiplicando corretamente as matrizes, foram realizados testes com matrizes menores, onde elas foram impressas e assim, foi possível comparar os resultados:







Os resultados obtidos foram validados de forma externa por meio de uma calculadora online, disponível no site: https://matrix.reshish.com/ptBr/multCalculation.php, onde foi possível obter o mesmo resultado dos testes, mostrando que a multiplicação funciona:

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
1	64	64	64	64	64	64	64	64
2	64	64	64	64	64	64	64	64
3	64	64	64	64	64	64	64	64
4	64	64	64	64	64	64	64	64
5	64	64	64	64	64	64	64	64
6	64	64	64	64	64	64	64	64
7	64	64	64	64	64	64	64	64
8	64	64	64	64	64	64	64	64

Outro importante fator considerado é o corte da recursão, uma vez que para matrizes suficientemente pequenas o processo de multiplicação direto é mais eficiente, após realizados os testes, em todos os casos foi considerado um corte para n=64. Ou seja, se a matriz fosse menor que 64x64 ou 2^6 x 2^6, a matriz seria passada para um método direto de multiplicação. Esse corte também ajuda a evitar estouro de memória, uma vez que a cada chamada recursiva novas submatrizes serão alocadas para o processo, visto que não há ganho de desempenho nos casos, o corte então foi definido para garantir melhor consumo de memória e velocidade.

Logo, para o experimento foram testadas a multiplicação de matrizes em tamanhos a partir de 128x128, em todos os casos a matriz A foi preenchida com 2 e a matriz B preenchida com 4. A matriz C foi calculada usando o algoritmo de *Strassen*.

Os testes foram executados considerando a implementação serial, as implementações paralelas foram testadas com 4,6 e 8 processos/threads.

Cada versão foi executada por múltiplas vezes seguidas e os dados foram anotados para obtenção de média e comparação do resultado.

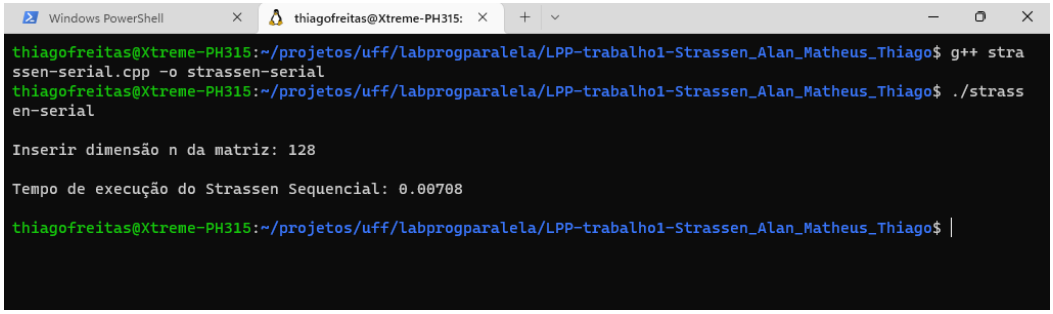
7.1. Compilando e Executando os Programas:

Para compilar o programa c++ da implementação sequêncial foi usado o comando:

g++ strassen-serial.cpp -o strassen-serial

Para executá-lo, apenas usamos ./strassen-serial

Abaixo o programa sendo compilado e executado para n=128:



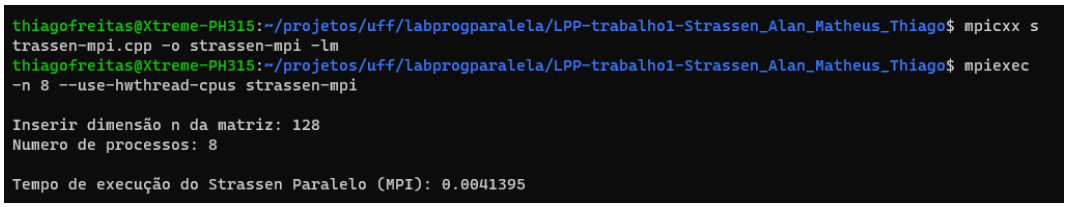
Para compilar o programa paralelo com mpi foi usado o comando:

mpicxx strassen-mpi.cpp -o strassen-mpi -lm

E para executá-lo:

mpiexec -n 8 --use-hwthread-cpus strassen-mpi

Onde n é o número de processadores e –use-hwthread-cpus foi usado para permitir a execução com 8 processadores usando o recurso de *hiperthreading* disponível nos processadores intel. A seguir o programa executando para uma matriz n=128:



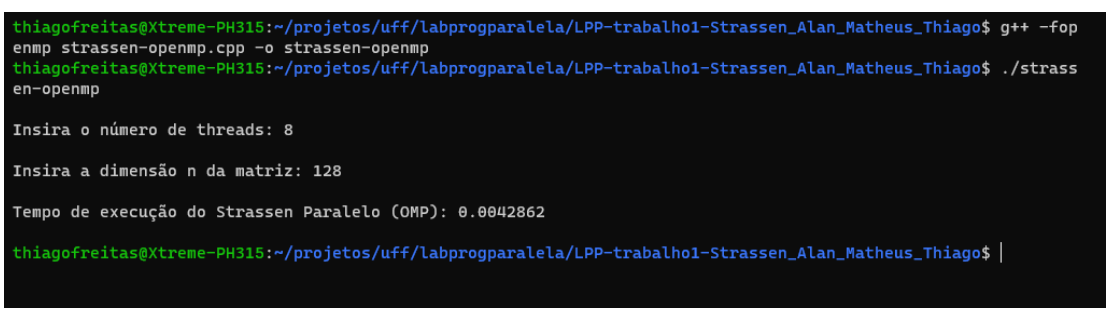
Para compilar o programa paralelo com *OpenMP* foi usado o comando:

g++ -fopenmp strassen-openmp.cpp -o strassen-openmp

E para executá-lo:

./strassen-openmp

No programa será solicitado não apenas o n mas a quantidade de processadores a ser usada, definindo por meio da função omp_set_num_threads() a quantidade de threads para execução. Abaixo o programa executando para uma matriz n=128 com 8 threads:



7.2 Resultados Obtidos

A seguir temos as tabelas e gráficos obtidos para as execuções, considerando:

N = 128, 256, 512, 1024, 2048 e 4096. Cada algoritmo foi executado por pelo menos 5x para aproximar os resultados e desconsiderar valores ocasionalmente discrepantes.

7.2.1. Resultados para n=128:

Neste caso foi possível observar que as versões paralelas, embora consigam executar em tempo menor, não representam uma redução considerável comparado a versão sequencial, reforça-se que em alguns casos, a execução com mais threads até empata ou se mostra ligeiramente inferior.

Considerou-se um empate técnico, embora seja possível observar uma redução se comparado ao sequencial.

N = 128							
execução	Sequencial	MPI(P=4)	MPI(P=6)	MPI(P = 8)	OMP(P=4)	OMP(P=6)	OMP(P=8)
1	0,0069	0,0063	0,0071	0,0064	0,0024	0,0035	0,0030
2	0,0078	0,0052	0,0065	0,0051	0,0034	0,0031	0,0029
3	0,0067	0,0047	0,0054	0,0055	0,0029	0,0031	0,0033
4	0,0085	0,0063	0,0053	0,0046	0,0026	0,0031	0,0030
5	0,0066	0,008	0,0058	0,0043	0,0028	0,0029	0,0034
MÉDIA	0,0073	0,0061	0,0060	0,0052	0,0028	0,0031	0,0031

Tabela 1 – Resultados para n=128

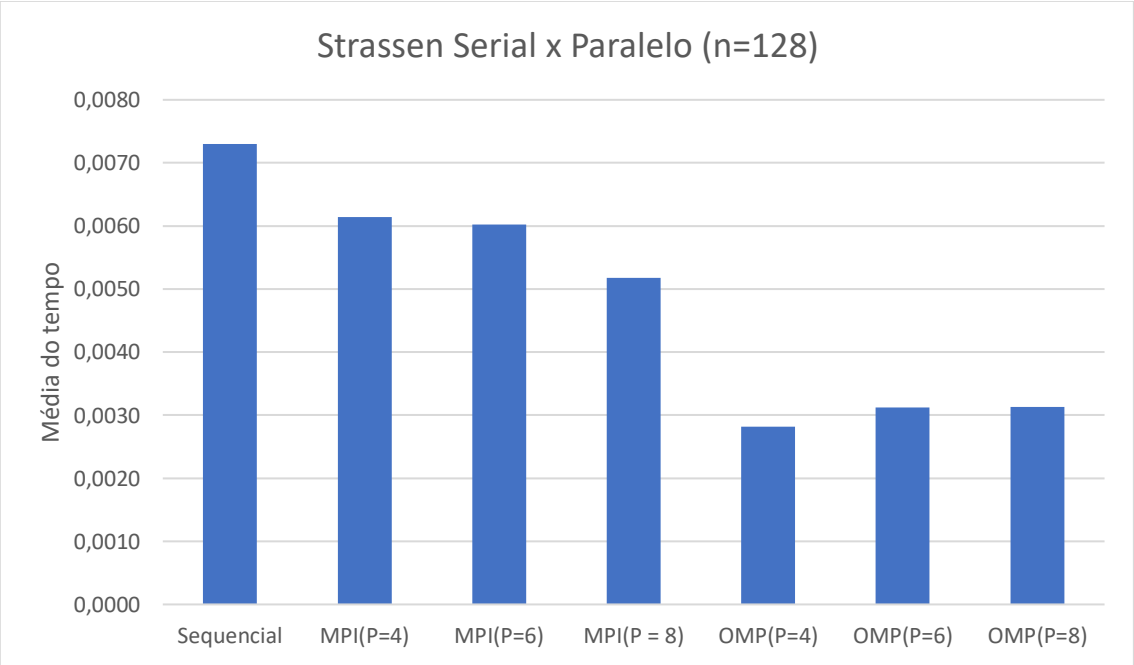


Gráfico 1 – Média de tempo obtido para multiplicação de matrizes com n=128

7.2.2. Resultados para n=256:

Ainda é observado um ganho de desempenho pequeno nos casos, mas neste caso já podemos observar que quanto mais threads disponíveis, menor o tempo de execução.

N = 256							
execução	Sequencial	MPI(P=4)	MPI(P=6)	MPI(P = 8)	OMP(P=4)	OMP(P=6)	OMP(P=8)
1	0,0543	0,0287	0,0328	0,0329	0,0178	0,0187	0,0195
2	0,0527	0,0393	0,0335	0,0277	0,0221	0,0197	0,0192
3	0,0609	0,0307	0,0343	0,0204	0,0169	0,0193	0,0153
4	0,0500	0,0290	0,0233	0,0212	0,0207	0,0158	0,0161
5	0,0551	0,0387	0,0245	0,0201	0,0224	0,0251	0,0178
MEDIA	0,0546	0,0333	0,0297	0,0245	0,0200	0,0197	0,0176

Tabela 2 – Resultados para n=256

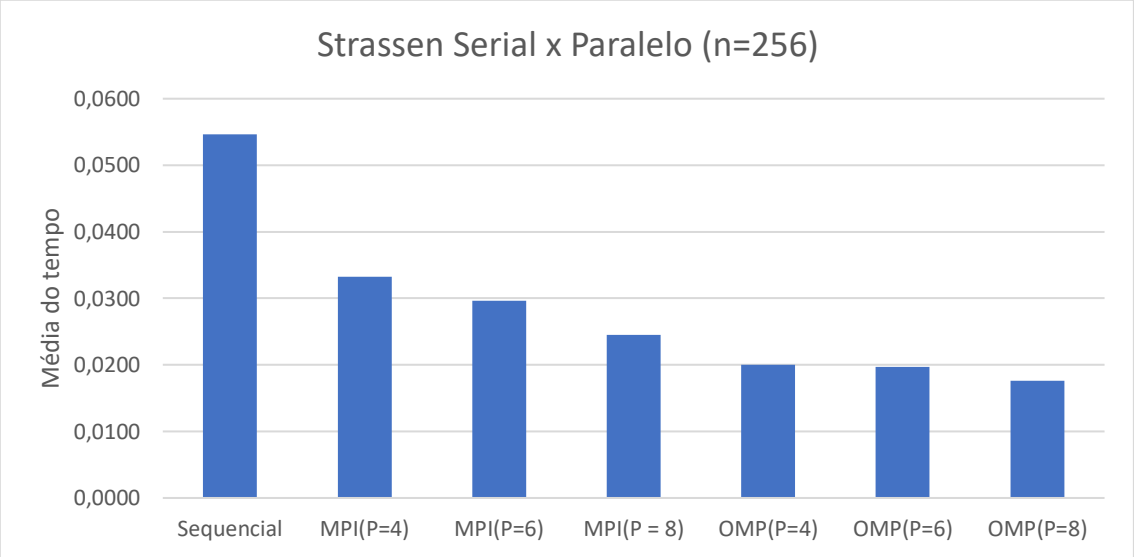


Gráfico 2 - Média de tempo obtido para multiplicação de matrizes com n=256

7.2.3. Resultados para n=512:

A partir deste ponto é possível começar a observar a diferença de tempo entre a execução serial e as execuções paralelas. A versão MPI já apresenta uma boa redução quando comparado a sequencial, e o mais rápido foi a execução *openMP* com 8 threads:

N = 512							
execução	Sequencial	MPI(P=4)	MPI(P=6)	MPI(P = 8)	OMP(P=4)	OMP(P=6)	OMP(P=8)
1	0,3732	0,2222	0,2254	0,1413	0,1364	0,1275	0,1051
2	0,3641	0,2277	0,1888	0,1430	0,1552	0,1114	0,1106
3	0,3604	0,2506	0,1735	0,1243	0,1309	0,1165	0,1071
4	0,3807	0,2651	0,1723	0,1227	0,1447	0,1153	0,1127
5	0,3864	0,2323	0,1971	0,1211	0,1410	0,1232	0,1038
MÉDIA	0,3730	0,2396	0,1914	0,1305	0,1416	0,1188	0,1079

Tabela 3 – Resultados para n=512

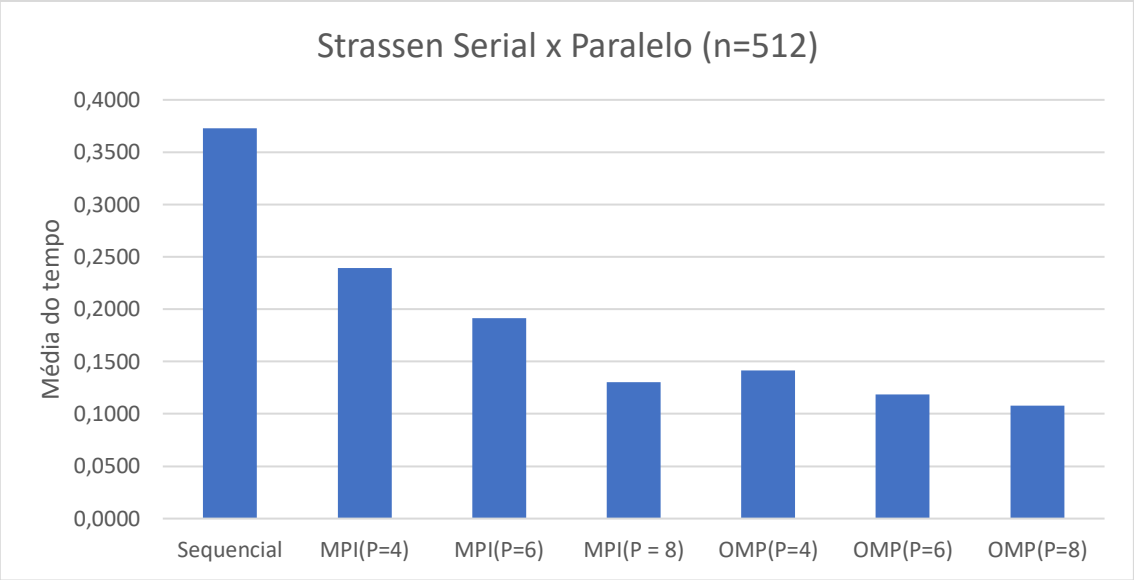


Gráfico 3 - Média de tempo obtido para multiplicação de matrizes com n=512

7.2.4. Resultados para n=1024:

Novamente vemos a execução MPI com ganho considerável de desempenho e a execução *OpenMP* obtendo os menores tempos, principalmente nos casos com 4 e 6 threads:

N = 1024							
execução	Sequencial	MPI(P=4)	MPI(P=6)	MPI(P = 8)	OMP(P=4)	OMP(P=6)	OMP(P=8)
1	2,6397	1,7836	1,3161	0,8118	1,0482	0,8310	0,7469
2	2,8666	1,5053	1,2755	0,8081	1,0018	0,8003	0,7441
3	2,5929	1,6139	1,2539	0,8555	1,0758	0,8624	0,7760
4	2,8459	1,6398	1,3547	0,8269	1,0515	0,7920	0,7554
5	2,7669	1,5994	1,2610	0,9007	1,0743	0,8130	0,7321
MÉDIA	2,7424	1,6284	1,2922	0,8406	1,0503	0,8198	0,7509

Tabela 4 – Resultados para n=1024

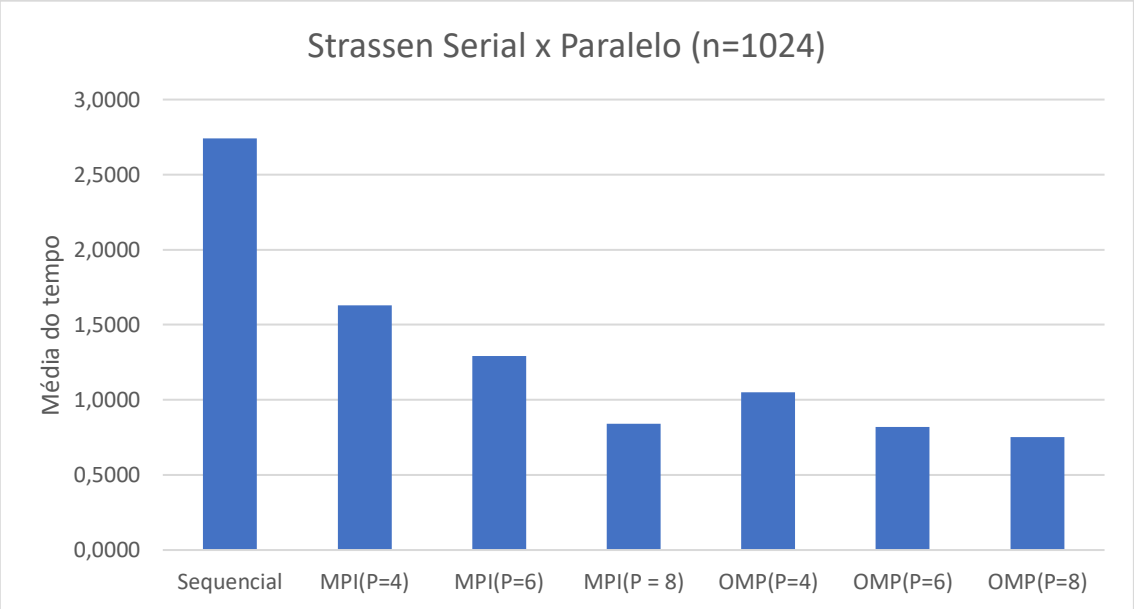


Gráfico 4 - Média de tempo obtido para multiplicação de matrizes com n=1024

7.2.5. Resultados para n=2048:

Mais uma vez é possível ver a execução MPI com ganho considerável de desempenho e a execução *OpenMP* obtendo os menores tempos, principalmente nos casos com 4 e 6 threads:

N = 2048							
execução	Sequencial	MPI(P=4)	MPI(P=6)	MPI(P = 8)	OMP(P=4)	OMP(P=6)	OMP(P=8)
1	18,5100	11,1500	9,3013	6,0516	7,4392	5,7490	5,5318
2	19,0800	11,2470	9,1649	5,6135	7,8574	6,0082	5,2286
3	18,7770	11,6460	9,0820	5,7353	7,3557	5,9319	5,0991
4	18,8190	11,5010	8,9399	5,5879	7,3621	5,7073	5,1015
5	18,7740	11,3230	8,8398	5,5365	7,2726	5,7094	4,9395
MÉDIA	18,7920	11,3734	9,0656	5,7050	7,4574	5,8212	5,1801

Tabela 5 – Resultados para n=2048

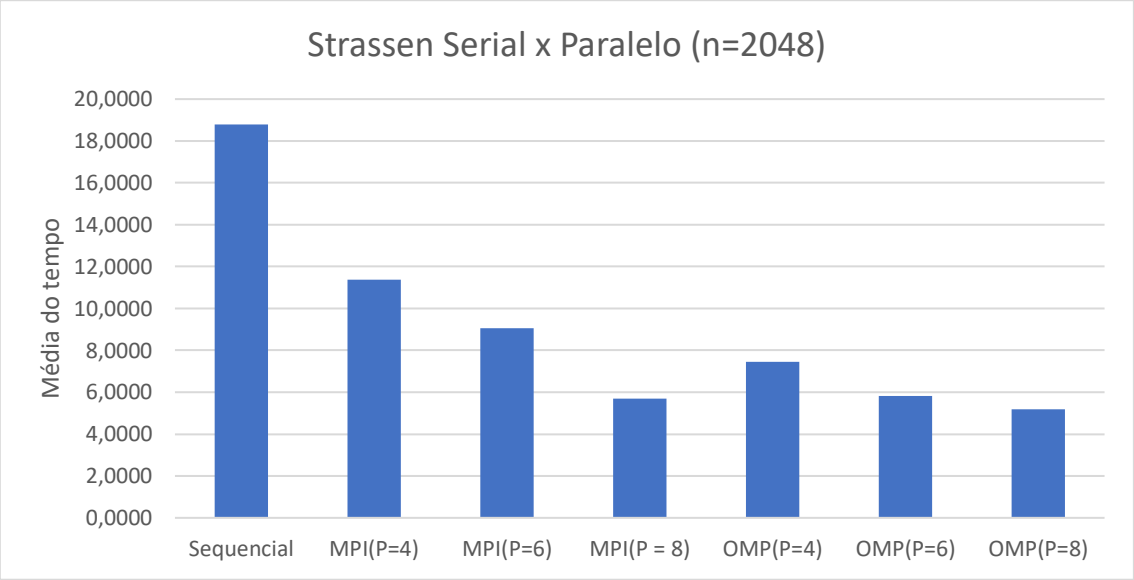


Gráfico 5 - Média de tempo obtido para multiplicação de matrizes com n=2048

7.2.6. Resultados para n=4096:

Neste teste novamente vemos a implementação *OpenMp* mais rápida, principalmente quando temos 4 ou 6 processos/threads. Observa-se no entanto que o tempo quando com 8 processadores ficou bem similar ao obtido pelo MPI. De qualquer forma, o ganho de tempo quando comparado ao sequencial é considerável em todos os casos quando paralelizado:

N = 4096							
execução	Sequencial	MPI(P=4)	MPI(P=6)	MPI(P = 8)	OMP(P=4)	OMP(P=6)	OMP(P=8)
1	129,9800	79,2250	61,4720	38,4610	55,3200	42,7120	37,618
2	131,8300	80,9650	62,3120	38,9830	53,4170	41,8140	38,3400
3	131,5700	90,6070	63,2230	38,9100	50,7650	40,7090	37,3650
4	132,2700	79,9580	61,2810	38,9390	55,9900	41,9340	36,5800
5	131,5900	78,9390	61,0420	39,0180	51,0510	40,5540	36,4410
MEDIA	131,4480	81,9388	61,8660	38,8622	53,3086	41,5446	37,2688

Tabela 6 – Resultados para n=4096

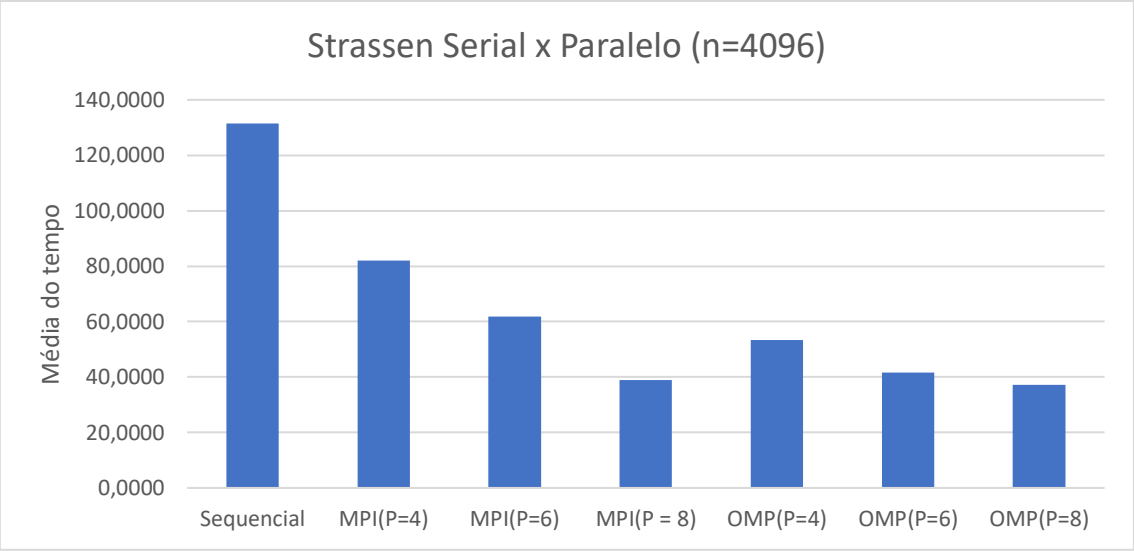


Gráfico 6 - Média de tempo obtido para multiplicação de matrizes com n=4096

7.2.7. Comparando Desempenho:

Para finalizar, foi realizada a comparação do desempenho das médias, a fim de se estabelecer o ganho de desempenho à medida que o valor de n cresce:

N	Sequencial	MPI(P=4)	MPI(P=6)	MPI(P = 8)	OMP(P=4)	OMP(P=6)	OMP(P=8)
N=128	0,007	0,006	0,006	0,005	0,003	0,003	0,003
N=256	0,055	0,033	0,030	0,024	0,020	0,020	0,018
N=512	0,373	0,240	0,191	0,130	0,142	0,119	0,108
N=1024	2,742	1,628	1,292	0,841	1,050	0,820	0,751
N=2048	18,792	11,373	9,066	5,705	7,457	5,821	5,180
N=4096	131,448	81,939	61,866	38,862	53,309	41,545	37,269

Tabela 7 – Comparativo das médias obtidas

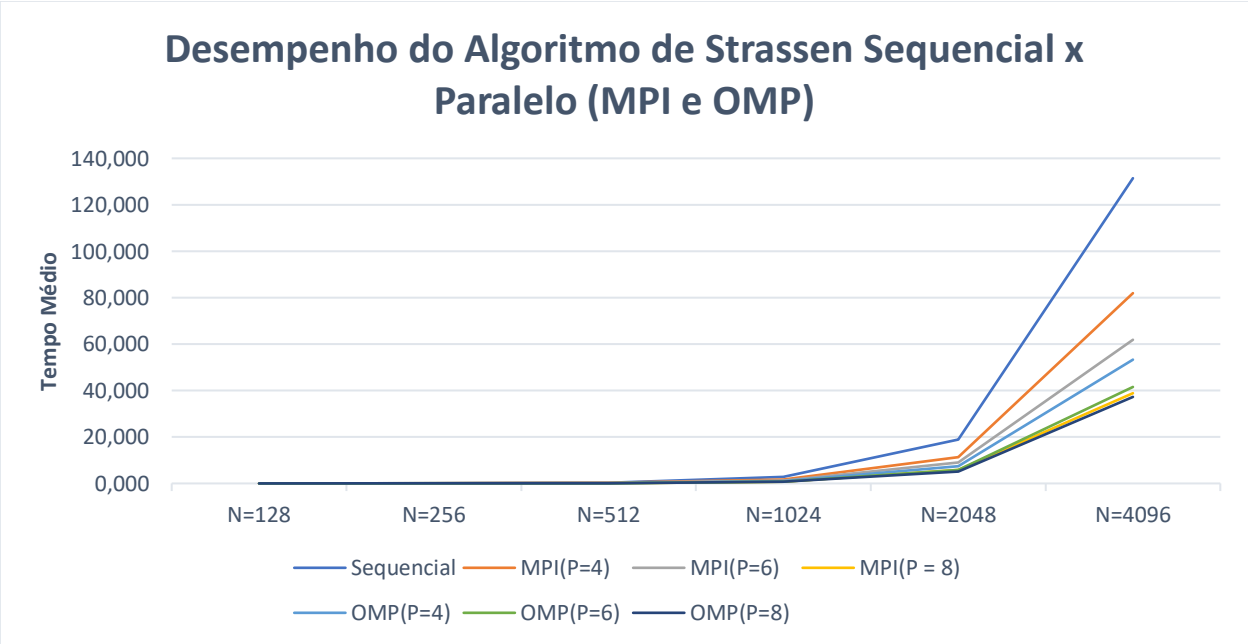


Gráfico 7 – Comparativo do desempenho de Strassen Sequêncial vs Paralelo a medida que o valor de N cresce.

É possível observar claramente que a partir de n=512 a diferença começa a ser perceptível quanto ao desempenho, e essa diferença cresce à medida que o n também cresce, no caso de n=4096 nota-se que os algoritmos paralelos executados com 8 processadores/threads chegam a apresentar ganho de 70% quando comparados ao algoritmo sequencial.

8. Conclusão

O Algoritmo de *Strassen* com certeza se mostra bem eficiente quando o tamanho da matriz é suficientemente grande. O algoritmo sequencial já mostra ganhos de desempenho quando comparado à solução direta, mas podemos ver que com o uso do paralelismo essa vantagem fica mais evidente.

Para matrizes de tamanho 2048 e 4096 nota-se uma melhora de desempenho que chega na casa dos 70% quando executado com 8 processadores.

Embora um grande problema do *Strassen* seja o alto consumo de memória (uma vez que a cada chamada recursiva novas alocações de matrizes são realizadas), as implementações apresentadas neste trabalho foram capazes de melhorar este consumo. Isso permitiu a execução de testes com matrizes maiores (como n=4096). Claro que em função das alocações/liberações no processo há um aumento no tempo.

De qualquer forma, as implementações apresentadas mostram eficiência e consumo razoável de memória bem como tempos excelentes (ainda que um pouco maiores quando comparado a outras implementações que chegam a gastar mais que o triplo de memória), desta forma, considera-se que as soluções apresentadas foram as com melhor equilíbrio entre o consumo e o tempo de execução.

Ideias futuras para este algoritmo incluem: A possibilidade de execução por meio de clusters/rede, possibilitando maior quantidade de nós/processos disponíveis; A possibilidade de implementação considerando paralelização por meio de GPU.

9. Referências

Matrix computations (em inglês) - Golub, Gene Howard; Van Loan, Charles F. (1996). 3 ed. [S.l.]: JHU Press. pags. 31 - 33.

Divisão e Conquista – Prof. Maria Inés Castiñeira - Disponível em: https://slideplayer.com.br/slide/385710/ - Acesso em Junho de 2022

EXPERIMENTS WITH STRASSEN'S ALGORITHM: FROM SEQUENTIAL TO PARALLEL - Fengguang Song, Jack Dongarra, Shirley Moore - Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.6510&rep=rep1&type=pdf – Acesso em Junho/2022

Matrix Multiplication Using Strassen's Algorithm With MPI – Mazarakis Periklis, Papadopoulous Aristeidis, Tsapekos Theodoros. Disponível em: https://github.com/aristosp/StrassenMPI_Project/blob/main/Report_english.pdf Acesso em junho/2022