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:

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

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

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

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.

Neste trabalho, as matrizes M (M1 a M7) serão nomeadas como P (P1 a P7).

3. Método Sequencial

No método sequencial a função recebe as matrizes A e B e o tamanho n (2^k) da matriz, inicialmente cria uma matriz C resposta, há uma condição de parada da recursão, caso o tamanho não seja menor que a condição de parada, realiza o procedimento de fragmentação das matrizes A e B, gerando as matrizes A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}.

Após este processo chama recursivamente a função *Strassen* para calcular as matrizes P1 a P7 por meio das operações de soma/multiplicações descritas no tópico anterior. Após o cálculo das matrizes, monta as matrizes $C_{i,j}$ seguindo o modelo informado no tópico 2, unindo-as no final do processo. Quando o tamanho da matriz é pequeno o suficiente (de acordo com a condição de parada) o algoritmo chama o processo direto de multiplicação.

A seguir a implementação usada nos testes deste trabalho:

```
//Função que implementa a solução divisão e conquista do algoritmo de Strassen
//Entradas: tamanho da matriz nxn, ponteiro para a matriz A, ponteiro para a matriz B
//Saida: Matriz C, resultado da operação A x B
int** multiplica_strassen_rec(int n, int **A, int** B){

    //Cria Matriz Resposta C e a popula com zeros
    int **C = cria_matriz(n, 0);

    //Se n for maior que 64, condição de parada não satisfeita.. divide a matriz
    if(n>64) {
        int ** a11 = dividir_matriz(n, A, 0, 0);
        int ** a12 = dividir_matriz(n, A, 0, (n/2));
        int ** a21 = dividir_matriz(n, A, (n/2), 0);
        int ** a22 = dividir_matriz(n, A, (n/2), (n/2));
        int ** b11 = dividir_matriz(n, B, 0, 0);
        int ** b12 = dividir_matriz(n, B, 0, n/2);
        int ** b21 = dividir_matriz(n, B, n/2, 0);
        int ** b22 = dividir_matriz(n, B, n/2, n/2);

        //Chamada Recursiva para dividir e Conquistar
        int** P1= multiplica_strassen_rec(n/2, soma_matriz(n/2, a11, a22),soma_matriz(n/2, b11, b22));
        int** P2= multiplica_strassen_rec(n/2, soma_matriz(n/2, a21, a22), b11);
        int** P3= multiplica_strassen_rec(n/2, a11,subtrai_matriz(n/2, b12, b22));
        int** P4= multiplica_strassen_rec(n/2, a22,subtrai_matriz(n/2, b21, b11));
        int** P5= multiplica_strassen_rec(n/2, soma_matriz(n/2, a11, a12),b22);
        int** P6= multiplica_strassen_rec(n/2,subtrai_matriz(n/2, a21, a11),soma_matriz(n/2, b11, b12));
        int** P7= multiplica_strassen_rec(n/2, subtrai_matriz(n/2, a12, a22), soma_matriz(n/2, b21, b22));

        //Operações para cálculo das matrizes Cij
        int** c11 = soma_matriz(n/2, subtrai_matriz(n/2, soma_matriz(n/2, P1, P4), P5), P7);
        int** c12 = soma_matriz(n/2, P3,P5);
        int** c21 = soma_matriz(n/2, P2,P4);
        int** c22 = soma_matriz(n/2, subtrai_matriz(n/2, soma_matriz(n/2, P1, P3), P2), P6);
```

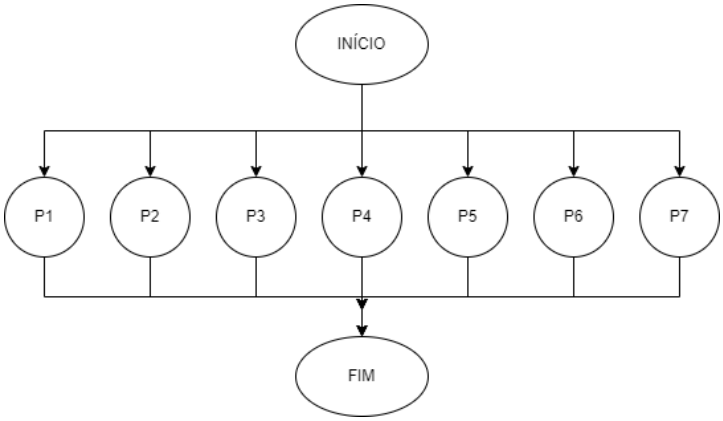
```
        //Compor (juntar) as Matrizes
        compor_matriz(n/2, c11, C, 0, 0);
        compor_matriz(n/2, c12, C,0,n/2);
        compor_matriz(n/2, c21, C, n/2, 0);
        compor_matriz(n/2, c22, C, n/2, n/2);

        //Liberar memória
        a11 = libera_matriz(a11, n/2);
        a12 = libera_matriz(a12, n/2);
        a21 = libera_matriz(a21, n/2);
        a22 = libera_matriz(a22, n/2);
        b11 = libera_matriz(b11, n/2);
        b12 = libera_matriz(b12, n/2);
        b21 = libera_matriz(b21, n/2);
        b22 = libera_matriz(b22, n/2);
        P1 = libera_matriz(P1, n/2);
        P2 = libera_matriz(P2, n/2);
        P3 = libera_matriz(P3, n/2);
        P4 = libera_matriz(P4, n/2);
        P5 = libera_matriz(P5, n/2);
        P6 = libera_matriz(P6, n/2);
        P7 = libera_matriz(P7, n/2);
        c11 = libera_matriz(c11, n/2);
        c12 = libera_matriz(c12, n/2);
        c21 = libera_matriz(c21, n/2);
        c22 = libera_matriz(c22, n/2);
    }
    else {
        //Condição para fim da recursão, chamada ao algoritmo direto/usual.
        multiplica_matrizes(n, A, B);
    }
    return C;
}
```

4. Método Paralelo

Para facilitar no cálculo da matriz pelo método paralelo, fez-se uso de matrizes auxiliares S1 a S10, usadas nos processos de soma/subtração para simplificar a chamada dos processos de multiplicação das matrizes P1 a P7.

A ideia central do algoritmo consiste em dividir esses cálculos entre os processos disponíveis para acelerar a obtenção do resultado. Uma 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.

5. Implementação Paralela

Segue a implementação dividida em trechos com a respectiva descrição:

A função cria ou define as variáveis necessárias ao processo e aloca as matrizes na memória:

```
//Função Principal que implementa o algoritmo de Strassen Paralelo
//Entradas: tamanho da matriz, valor para preencher matriz A, valor para preencher matriz B, argumentos do main (argc argv)
//Saida: executa o processo e retorna 0
int Strassen_mpi(int tam, int val_a, int val_b, int argc, char *argv[]) {

    //Definição de variáveis da interação
    int n, va, vb;
    n = tam;
    va= val_a;
    vb= val_b;
    int new_n = n/2;
    int rank,comm_size;
    int i,j;
    srand(0);

    //Inicializando as Matrizes A e B
    int **A = cria_matriz(n, va);
    int **B = cria_matriz(n, vb);

    //Alocando Memória para matrizes do processo
    int **P1 = mem_alloc(new_n);
    int **P2 = mem_alloc(new_n);
    int **P3 = mem_alloc(new_n);
    int **P4 = mem_alloc(new_n);
    int **P5 = mem_alloc(new_n);
    int **P6 = mem_alloc(new_n);
    int **P7 = mem_alloc(new_n);
    int **C11 = mem_alloc(new_n);
    int **C12 = mem_alloc(new_n);
    int **C21 = mem_alloc(new_n);
    int **C22 = mem_alloc(new_n);
    int **C_parallel = mem_alloc(n);
    float parallel_start;
```

A função inicia o MPI e obtém as variáveis para a continuação. Se rank=0 inicia as matrizes enviando-as para todos os nós disponíveis, após isso aloca as submatrizes A e B para continuidade do processo, neste momento também é obtido o tempo de início do procedimento:

```
    //MPI
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    //printf("inicializou MPI\n");
    //printf("comm_size: %d\n", comm_size);

    //Iniciando Matrizes
    if (rank == 0){

        parallel_start = MPI_Wtime();
        for (i=1;i<comm_size;i++){
            // Enviando as matrizes A,B para todos os nós
            MPI_Send(&(A[0][0]),(n*n),MPI_INT,i,1,MPI_COMM_WORLD);
            MPI_Send(&(B[0][0]),(n*n),MPI_INT,i,2,MPI_COMM_WORLD);
        }

        //Criar submatrizes para processos
        int **A11 = mem_alloc(new_n);
        int **A12 = mem_alloc(new_n);
        int **A21 = mem_alloc(new_n);
        int **A22 = mem_alloc(new_n);
        int **B11 = mem_alloc(new_n);
        int **B12 = mem_alloc(new_n);
        int **B21 = mem_alloc(new_n);
        int **B22 = mem_alloc(new_n);
```

```
//Preenchendo submatrizes
for (i = 0; i < new_n; i++) {
    for(j = 0; j < new_n; j++){
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][j + new_n];
        A21[i][j] = A[i + new_n][j];
        A22[i][j] = A[i + new_n][j + new_n];
        B11[i][j] = B[i][j];
        B12[i][j] = B[i][j + new_n];
        B21[i][j] = B[i + new_n][j];
        B22[i][j] = B[i + new_n][j + new_n];
    }
}

A = libera_matriz(A, n);
B = libera_matriz(B, n);
```

Por meio de um loop a função vai preencher as submatrizes usando os elementos das matrizes A e B, para evitar um estouro de memória, uma vez que não serão mais utilizadas, as matrizes A e B são liberadas assim que as submatrizes são definidas.

Considerando a possibilidade de quantidades diversas de processadores disponíveis, o código distribui de acordo a quantidade de matrizes P a ser calculadas, se temos 2 processadores disponíveis, o processo 0 calcula P1 – P4 e o processo 1 calcula P5 a P7. De igual modo se existem 4 processadores disponíveis o processo 0 calcula P1 e P2, o processo 1 calcula P3 e P4, o processo 2 calcula P5 e P6 e o processo 3 calcula P7. Se há mais de 4 processadores, cada processo calcula uma matriz P. O código a seguir mostra todo este processo:

```
// Dependendo do número de processos, calcular adequadamente
// Si, Pi
if (comm_size == 2){
    int **S1 = mem_alloc(new_n);
    int **S2 = mem_alloc(new_n);
    int **S3 = mem_alloc(new_n);
    int **S4 = mem_alloc(new_n);
    for (i=0;i<new_n;i++){
        for(j=0;j<new_n;j++){
            S1[i][j] = B12[i][j] - B22[i][j];
            S2[i][j] = A11[i][j] + A12[i][j];
            S3[i][j] = A21[i][j] + A22[i][j];
            S4[i][j] = B21[i][j] - B11[i][j];
        }
    }
    P1 = Strassen(A11,S1,new_n);
    P2 = Strassen(S2,B22,new_n);
    P3 = Strassen(S3,B11,new_n);
    P4 = Strassen(A22,S4,new_n);

    //Liberando memoria
    A11 = libera_matriz(A11, new_n);
    A12 = libera_matriz(A12, new_n);
    A21 = libera_matriz(A21, new_n);
    A22 = libera_matriz(A22, new_n);
    B11 = libera_matriz(B11, new_n);
    B12 = libera_matriz(B12, new_n);
    B21 = libera_matriz(B21, new_n);
    B22 = libera_matriz(B22, new_n);
    S1 = libera_matriz(S1, new_n);
    S2 = libera_matriz(S2, new_n);
    S3 = libera_matriz(S3, new_n);
    S4 = libera_matriz(S4, new_n);
}
```

```
    else if (comm_size == 4){
        int **S1 = mem_alloc(new_n);
        int **S2 = mem_alloc(new_n);
        for (i=0;i<new_n;i++){
            for(j=0;j<new_n;j++){
                S1[i][j] = B12[i][j] - B22[i][j];
                S2[i][j] = A11[i][j] + A12[i][j];
            }
        }
    P1 = Strassen(A11,S1,new_n);
    P2 = Strassen(S2,B22,new_n);
    //Liberando memoria
    A11 = libera_matriz(A11, new_n);
    A12 = libera_matriz(A12, new_n);
    A21 = libera_matriz(A21, new_n);
    A22 = libera_matriz(A22, new_n);
    B11 = libera_matriz(B11, new_n);
    B12 = libera_matriz(B12, new_n);
    B21 = libera_matriz(B21, new_n);
    B22 = libera_matriz(B22, new_n);
    S1 = libera_matriz(S1, new_n);
    S2 = libera_matriz(S2, new_n);
    }
```

```
    else if (comm_size > 4){
        int **S1 = mem_alloc(new_n);
        free(A12);free(A21);free(A22);
        free(B11);free(B21);
        for (i=0;i<new_n;i++){
            for(j=0;j<new_n;j++){
                S1[i][j] = B12[i][j] - B22[i][j];
            }
        }
        P1 = Strassen(A11,S1,new_n);

        //Liberando memoria
        A11 = libera_matriz(A11, new_n);
        B12 = libera_matriz(B12, new_n);
        B22 = libera_matriz(B22, new_n);
        S1 = libera_matriz(S1, new_n);
    }
}
```

Na sequência cada processo aloca localmente a memória e recebe do processo principal as matrizes, realiza também a divisão das matrizes A e B em submatrizes e executa sua tarefa:

```
    // Alocar memoria para A,B para todos os processos
    int **local_A = mem_alloc(n);
    int **local_B = mem_alloc(n);
    MPI_Recv(&(local_A[0][0]),(n*n),MPI_INT,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Recv(&(local_B[0][0]),(n*n),MPI_INT,0,2,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

    // Criar cópias locais das submatrizes de A e B
    int **local_A11 = mem_alloc(new_n);
    int **local_A12 = mem_alloc(new_n);
    int **local_A21 = mem_alloc(new_n);
    int **local_A22 = mem_alloc(new_n);
    int **local_B11 = mem_alloc(new_n);
    int **local_B12 = mem_alloc(new_n);
    int **local_B21 = mem_alloc(new_n);
    int **local_B22 = mem_alloc(new_n);
    for (i = 0; i < new_n; i++) {
        for(j = 0; j<new_n; j++){
            local_A11[i][j] = local_A[i][j];
            local_A12[i][j] = local_A[i][j + new_n];
            local_A21[i][j] = local_A[i + new_n][j];
            local_A22[i][j] = local_A[i + new_n][j + new_n];
            local_B11[i][j] = local_B[i][j];
            local_B12[i][j] = local_B[i][j + new_n];
            local_B21[i][j] = local_B[i + new_n][j];
            local_B22[i][j] = local_B[i + new_n][j + new_n];
        }
    }

    local_A = libera_matriz(local_A, n);
    local_B = libera_matriz(local_B, n);
```

Caso haja 2 processos, o processo a seguir calcula P5 a P7:

```
// Continuando... de acordo com o número de processos, calcular adequadamente
if (comm_size == 2){
    int **local_S5 = soma_matriz(local_A11,local_A22,new_n);
    int **local_S6 = soma_matriz(local_B11,local_B22,new_n);
    int **local_S7 = subtrai_matriz(local_B21,local_B22,new_n);
    int **local_S8 = soma_matriz(local_B21,local_B22,new_n);
    int **local_S9 = subtrai_matriz(local_A11,local_A21,new_n);
    int **local_S10 = soma_matriz(local_B11,local_B12,new_n);

    //Liberar memoria
    local_A11 = libera_matriz(local_A11, new_n);local_A12 = libera_matriz(local_A12, new_n);
    local_A21 = libera_matriz(local_A21, new_n);local_A22 = libera_matriz(local_A22, new_n);
    local_B11 = libera_matriz(local_B11, new_n);local_B12 = libera_matriz(local_B12, new_n);
    local_B21 = libera_matriz(local_B21, new_n);local_B22 = libera_matriz(local_B22, new_n);

    int **local_P5 = Strassen(local_S5,local_S6,new_n);
    int **local_P6 = Strassen(local_S7,local_S8,new_n);
    int **local_P7 = Strassen(local_S9,local_S10,new_n);

    // Mandar os coeficientes Pi calculados ao processo principal
    MPI_Send(&(local_P5[0][0]),(new_n*new_n),MPI_INT,0,5,MPI_COMM_WORLD);
    MPI_Send(&(local_P6[0][0]),(new_n*new_n),MPI_INT,0,6,MPI_COMM_WORLD);
    MPI_Send(&(local_P7[0][0]),(new_n*new_n),MPI_INT,0,7,MPI_COMM_WORLD);

    //Liberar memoria
    local_S5 = libera_matriz(local_S5, new_n);local_S6 = libera_matriz(local_S6, new_n);
    local_S7 = libera_matriz(local_S7, new_n);local_S8 = libera_matriz(local_S8, new_n);
    local_P5 = libera_matriz(local_P5, new_n);local_P6 = libera_matriz(local_P6, new_n);
    local_P7 = libera_matriz(local_P7, new_n);
}
```

Se existem 4 processos, o rank 1 calcula P3 e P4:

```
else if (comm_size == 4){
    if (rank == 1){

        int **local_S3 = soma_matriz(local_A21,local_A22,new_n);
        int **local_S4 = subtrai_matriz(local_B21,local_B11,new_n);
        int **local_P3 = Strassen(local_S3,local_B11,new_n);
        int **local_P4 = Strassen(local_A22,local_S4,new_n);

        //Liberar Memoria
        local_A11 = libera_matriz(local_A11, new_n); local_A12 = libera_matriz(local_A12, new_n);
        local_A21 = libera_matriz(local_A21, new_n); local_A22 = libera_matriz(local_A22, new_n);
        local_B11 = libera_matriz(local_B11, new_n); local_B12 = libera_matriz(local_B12, new_n);
        local_B21 = libera_matriz(local_B21, new_n); local_B22 = libera_matriz(local_B22, new_n);
        local_S3 = libera_matriz(local_S3, new_n); local_S4 = libera_matriz(local_S4, new_n);

        // Mandar os coeficientes Pi calculados ao processo principal
        MPI_Send(&(local_P3[0][0]),(new_n*new_n),MPI_INT,0,3,MPI_COMM_WORLD);
        MPI_Send(&(local_P4[0][0]),(new_n*new_n),MPI_INT,0,4,MPI_COMM_WORLD);

        //Liberar Memoria
        local_P3 = libera_matriz(local_P3, new_n);
        local_P4 = libera_matriz(local_P4, new_n);

    }
```

O rank 2 calcula P5 e P6:

```
    else if(rank == 2){

        int **local_S5 = soma_matriz(local_A11,local_A22,new_n);
        int **local_S6 = soma_matriz(local_B11,local_B22,new_n);
        int **local_S7 = subtrai_matriz(local_A12,local_A22,new_n);
        int **local_S8 = soma_matriz(local_B21,local_B22,new_n);

        //Liberar Memoria
        local_A11 = libera_matriz(local_A11, new_n);local_A12 = libera_matriz(local_A12, new_n);
        local_A21 = libera_matriz(local_A21, new_n);local_A22 = libera_matriz(local_A22, new_n);
        local_B11 = libera_matriz(local_B11, new_n);local_B12 = libera_matriz(local_B12, new_n);
        local_B21 = libera_matriz(local_B21, new_n);local_B22 = libera_matriz(local_B22, new_n);

        int **local_P5 = Strassen(local_S5,local_S6,new_n);
        int **local_P6 = Strassen(local_S7,local_S8,new_n);

        //Liberar Memoria
        local_S5 = libera_matriz(local_S5, new_n);local_S6 = libera_matriz(local_S6, new_n);
        local_S7 = libera_matriz(local_S7, new_n);local_S8 = libera_matriz(local_S8, new_n);

        // Mandar os coeficientes Pi calculados ao processo principal
        MPI_Send(&(local_P5[0][0]),(new_n*new_n),MPI_INT,0,5,MPI_COMM_WORLD);
        MPI_Send(&(local_P6[0][0]),(new_n*new_n),MPI_INT,0,6,MPI_COMM_WORLD);

        //Liberar Memoria
        local_P5 = libera_matriz(local_P5, new_n);
        local_P6 = libera_matriz(local_P6, new_n);
    }
```

O rank 3 calcula P7:

```
        else if(rank == 3){

            int **local_S9 = subtrai_matriz(local_A11,local_A21,new_n);
            int **local_S10 = soma_matriz(local_B11,local_B12,new_n);
            int **local_P7 = Strassen(local_S9,local_S10,new_n);

            // Mandar os coeficientes Pi calculados ao processo principal
            MPI_Send(&(local_P7[0][0]),(new_n*new_n),MPI_INT,0,7,MPI_COMM_WORLD);

            //Liberar Memoria
            local_S9 = libera_matriz(local_S9, new_n);
            local_S10 = libera_matriz(local_S10, new_n);
            local_P7 = libera_matriz(local_P7, new_n);
        }
    }
```

Caso haja mais de 4 processos, cada processo calcula 1 matriz P:

```
    else if (comm_size > 4){
        // Cada processo calcula um Pi e envia ao processo principal
        if (rank == 1){
                int **local_S2 = soma_matriz(local_A11,local_A12,new_n);
                int **local_P2 = Strassen(local_S2,local_B22,new_n);
                MPI_Send(&(local_P2[0][0]),(new_n*new_n),MPI_INT,0,2,MPI_COMM_WORLD);
        }
        else if (rank == 2){
                int **local_S3 = soma_matriz(local_A21,local_A22,new_n);
                int **local_P3 = Strassen(local_S3,local_B11,new_n);
                MPI_Send(&(local_P3[0][0]),(new_n*new_n),MPI_INT,0,3,MPI_COMM_WORLD);
        }
        else if (rank == 3){
                int **local_S4 = subtrai_matriz(local_B21,local_B11,new_n);
                int **local_P4 = Strassen(local_A22,local_S4,new_n);
                MPI_Send(&(local_P4[0][0]),(new_n*new_n),MPI_INT,0,4,MPI_COMM_WORLD);
        }
        else if(rank == 4){
                int **local_S5 = soma_matriz(local_A11,local_A22,new_n);
                int **local_S6 = soma_matriz(local_B11,local_B22,new_n);
                int **local_P5 = Strassen(local_S5,local_S6,new_n);
                MPI_Send(&(local_P5[0][0]),(new_n*new_n),MPI_INT,0,5,MPI_COMM_WORLD);
        }
        else if(rank == 5){
                int **local_S7 = subtrai_matriz(local_A12,local_A22,new_n);
                int **local_S8 = soma_matriz(local_B21,local_B22,new_n);
                int **local_P6 = Strassen(local_S7,local_S8,new_n);
                MPI_Send(&(local_P6[0][0]),(new_n*new_n),MPI_INT,0,6,MPI_COMM_WORLD);
        }
        else if(rank == 6){
                int **local_S9 = subtrai_matriz(local_A11,local_A21,new_n);
                int **local_S10 = soma_matriz(local_B11,local_B12,new_n);
                int **local_P7 = Strassen(local_S9,local_S10,new_n);
                MPI_Send(&(local_P7[0][0]),(new_n*new_n),MPI_INT,0,7,MPI_COMM_WORLD);
        }
    }
}
```

Após o cálculo das matrizes pelo algoritmo de *Strassen* o processo principal recupera os resultados dos processos paralelos:

```
if (rank == 0){
    // Dependendo do número de processos, receber adequadamente os coeficientes Pi
    if (comm_size == 2){
        MPI_Recv(&(P5[0][0]),(new_n*new_n),MPI_INT,1,5,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P6[0][0]),(new_n*new_n),MPI_INT,1,6,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P7[0][0]),(new_n*new_n),MPI_INT,1,7,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    else if (comm_size == 4){
        MPI_Recv(&(P3[0][0]),(new_n*new_n),MPI_INT,1,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P4[0][0]),(new_n*new_n),MPI_INT,1,4,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P5[0][0]),(new_n*new_n),MPI_INT,2,5,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P6[0][0]),(new_n*new_n),MPI_INT,2,6,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P7[0][0]),(new_n*new_n),MPI_INT,3,7,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    else if (comm_size > 4){
        MPI_Recv(&(P2[0][0]),(new_n*new_n),MPI_INT,1,2,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P3[0][0]),(new_n*new_n),MPI_INT,2,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P4[0][0]),(new_n*new_n),MPI_INT,3,4,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P5[0][0]),(new_n*new_n),MPI_INT,4,5,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P6[0][0]),(new_n*new_n),MPI_INT,5,6,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&(P7[0][0]),(new_n*new_n),MPI_INT,6,7,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
```

```
            // Calcular as submatrizes C(i,j)
            for (i = 0; i<new_n; i++){
                for (j = 0;j<new_n; j++){
                C11[i][j] = P5[i][j] + P4[i][j] - P2[i][j] + P6[i][j];
                C12[i][j] = P1[i][j] + P2[i][j];
                C21[i][j] = P3[i][j] + P4[i][j];
                C22[i][j] = P5[i][j] + P1[i][j] - P3[i][j] - P7[i][j];
                }
            }
            // Criar uma matriz C, formata pelas submatrizes C(i,j)
            for (i=0;i<new_n;i++){
                for(j=0;j<new_n;j++){
                    C_parallel[i][j] = C11[i][j];
                    C_parallel[i][j+new_n] = C12[i][j];
                    C_parallel[new_n+i][j] = C21[i][j];
                    C_parallel[new_n+i][new_n+j] = C22[i][j];
                }
            }
            float parallel_end = MPI_Wtime();
            printf("Com %d core(s), O tempo para execução paralela foi %f \n",comm_size,parallel_end - parallel_start);
        }
        MPI_Finalize();
}
```

As submatrizes C são geradas e depois unidas para a matriz C final, resultado do processo. O algoritmo então pega o tempo final da execução e imprime na tela a quantidade de cores e o tempo calculado.

Durante todo o tempo no código pode ser observado o cuidado de liberar a memória para todas as matrizes que não eram mais necessárias, isto permitiu que matrizes maiores fossem calculadas pelo procedimento sem estouro de memória.

6. Experimentos Computacionais

Para o experimento foi utilizado uma máquina com a seguinte configuração: Processador Intel Core i7 9750h (6 nucleos/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 virtualizada dispõe de 50% da memória principal por limitação da tecnologia, logo, vamos considerar que os testes rodaram em 16GB de Ram.

Outro importante fator considerado é o corte da recursão, uma vez que para matrizes suficientemente pequenas o processo de multiplicação direto é mais eficiente, 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 a partir do momento em que os tempos começam a diferir entre os processos.

Logo, para o experimento foram testadas a multiplicação de matrizes em tamanho 2048x2048 e 4096x4096, 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*.

Cada versão foi executada 10x seguidas e os dados foram anotados para obtenção de média e comparação do resultado.

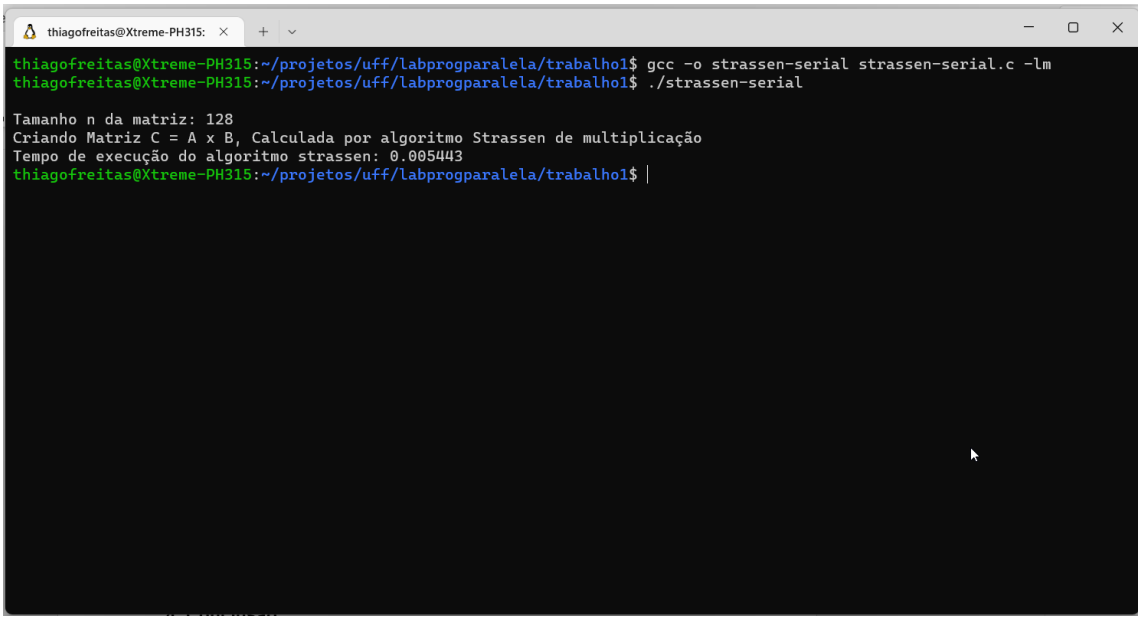
6.1. Compilando e Executando os Programas:

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

gcc -o strassen-serial strassen-serial.c -lm

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

Abaixo o programa sendo compilado e executado para k=7, n=128:



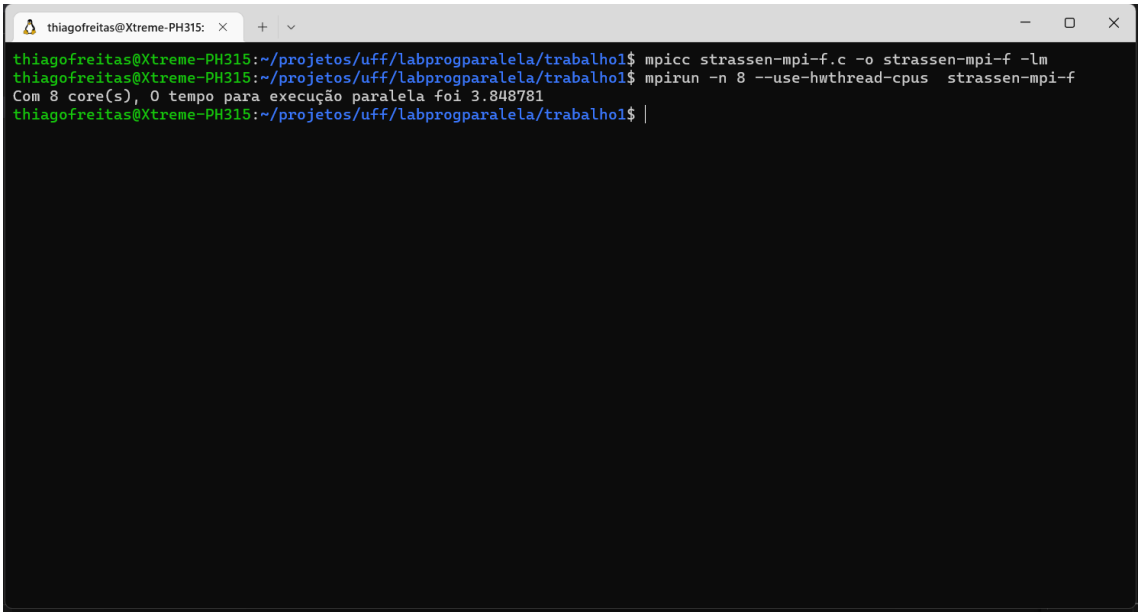
Para compilar o programa paralelo com mpi foi usado o comando:

mpicc strassen-mpi.c -o strassen-mpi-f -lm

E para executá-lo:

mpirun -n 8 --use-hwthread-cpus strassen-mpi-f

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.



6.2 Resultados Obtidos

Primeiro, o algoritmo foi comparado a partir de 512 sendo executado no modo sequencial e paralelo com 8 processadores, o gráfico a seguir mostra o resultado obtido:

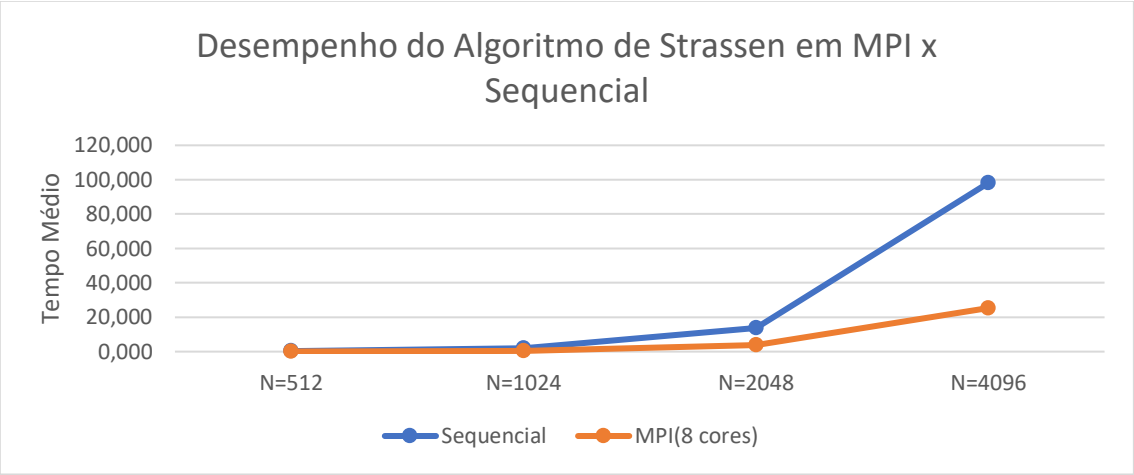


Gráfico 1 – Média de tempo gasto pelo algoritmo sequencial comparado ao Paralelo

Como podemos perceber, a diferença entre os algoritmos começa a se destacar a partir de N=2048, por isso os próximos testes focaram em matrizes grandes o suficiente para notar a diferença de desempenho.

Para uma matriz grande tamanho 2048x2048 ($n = 2^{11}$), executando 10x o teste considerando o algoritmo sequencial, a execução paralela com 2, 4 e 8 processadores, foram obtidos os seguintes resultados:

N = 2048				
execução	Sequencial	MPI(P=2)	MPI(P=4)	MPI(P = 8)
1	14,526	8,197	5,326	3,705
2	14,143	8,051	5,206	3,768
3	13,847	8,207	5,285	3,726
4	13,422	8,278	5,298	3,665
5	13,438	8,685	5,124	3,771
6	13,457	8,468	5,227	3,719
7	13,309	8,137	5,424	3,752
8	13,596	8,242	5,284	3,673
9	13,478	8,312	5,230	3,791
10	14,531	8,424	5,314	3,775
MEDIA	13,775	8,300	5,272	3,734

Tabela 1 – Resultados obtidos para matrizes de tamanho n=2048

Para uma matriz grande tamanho 4096x4096 ($n = 2^{12}$), executando 10x o teste considerando o algoritmo sequencial, a execução paralela com 2, 4 e 8 processadores, foram obtidos os seguintes resultados:

N = 4096				
execução	Sequencial	MPI(P=2)	MPI(P=4)	MPI(P = 8)
1	100,602	59,704	37,458	25,047
2	96,951	59,604	36,786	25,065
3	101,123	58,953	38,415	25,339
4	98,368	59,135	37,780	25,478
5	97,410	57,880	36,349	25,700
6	96,485	59,084	37,622	25,566
7	96,546	60,266	38,337	24,949
8	96,965	60,299	38,137	25,419
9	97,701	58,833	37,156	25,765
10	99,108	58,686	37,367	25,151
MEDIA	98,126	59,244	37,541	25,348

Tabela 2 -Resultados obtidos para matrizes de tamanho n=4096

Os dados obtidos mostram uma considerável melhora do Algoritmo paralelo frente a solução Sequencial, principalmente quando executado em maior quantidade de processadores.

Para uma melhor observação das médias obtidas, a seguir o gráfico de desempenho para cada caso citado:

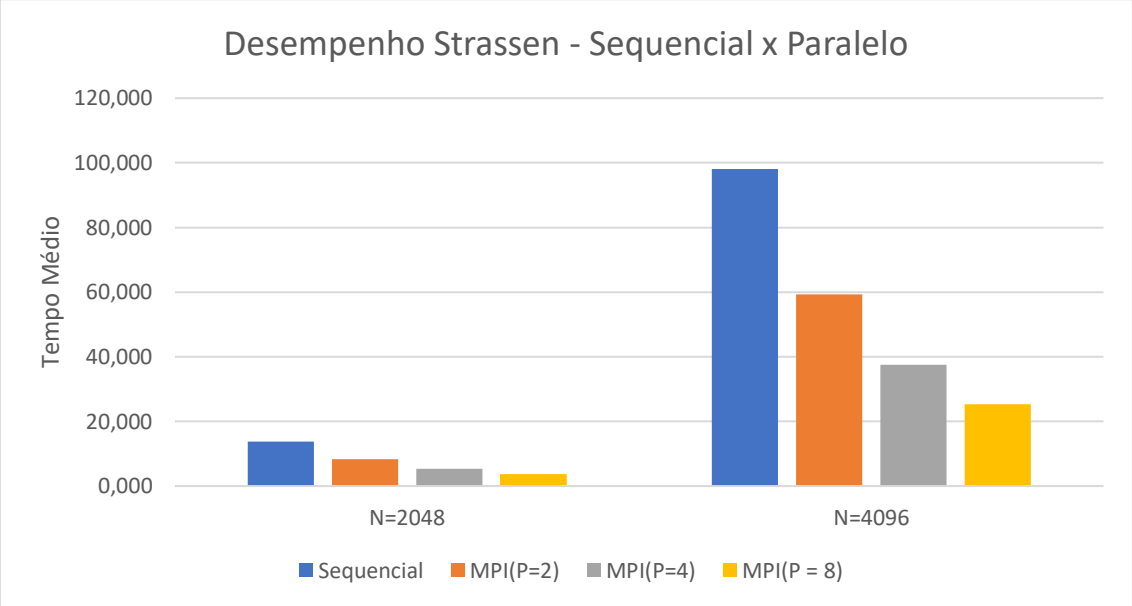


Gráfico 2 – Média de tempo gasto pelo algoritmo para multiplicação de matrizes

7. 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 melhor a de desempenho que chega a aproximadamente 70% quando executado com 8 processadores. O grande problema do *Strassen* é o alto consumo de memória (uma vez que a cada chamada

recursiva novas alocações de matrizes são realizadas), deste modo não foi possível até o momento realizar testes para matrizes maiores que 4096.

Ideias futuras para este algoritmo incluem: a verificação das alocações/liberações de memória a fim de verificar espaço para otimizar o processo; A possibilidade de execução por meio de clusters/rede, possibilitando maior quantidade de nós/processos disponíveis; A implementação em *OpenMP* e a possibilidade de implementação hibrida.

8. 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