Universidade de Aveiro

Computação Paralela (40779)

Paralelização Em MPI De Métodos Iterativos Para Resolução De Equações Diferenciais A Duas Dimensões

Gonçalo Freitas
NºMEC: 98012

Mestrado Em Engenharia Computacional

Docente:
Rui Américo Ferreira da Costa

23 de junho de 2023

1. Introdução

A computação paralela é uma abordagem que visa otimizar a capacidade de processamento dos computadores, permitindo a execução simultânea de múltiplas tarefas ou partes de uma tarefa em diferentes processadores, núcleos ou unidades de processamento. Ao contrário da computação sequencial, que realiza as operações numa única linha de execução, a computação paralela divide a carga de trabalho em várias partes menores que podem ser executadas em paralelo (simultâneo), diminuindo assim o tempo de execução do código/programa.

O estudo realizado para este trabalho consiste na paralelização, em `MPI` (Message Passing Interface), [1], dos métodos iterativos de *Jacobi* e *Gauss-Seidel* para uma determinada equação de *Poisson* a duas dimensões.

Este projeto divide-se em 4 exercícios onde primeiramente, utilizando o método de *Jacobi* paralelizado, vamos analisar o problema com condições fronteiras definidas. Em seguida, para o mesmo método, vamos utilizar condições de fronteira periódicas, sendo que para estas vamos analisar tanto um estêncil de 5 pontos quanto um estêncil de 9 pontos. Por fim, vamos paralelizar o método de relaxação de *Gauss-Seidel* de forma a comparar os resultados obtidos com o método de *Jacobi*.

O código que implementa cada exercício foi devidamente anexado, divido em pastas por alínea, e possui um nome do tipo `98012_x`, onde `x` representa o número do exercício (a,b,c,d). Quanto as alterações realizadas ao código, em cada exercício, estas podem ser encontradas abaixo de um comentário que começa por `!!!` (`//!!!`).

1..1 Equações diferenciais 2D

A equação diferencial utilizada neste projeto é a equação de *Poisson* a duas dimensões:

$$\frac{\partial^2 V(x,y)}{\partial x^2} + \frac{\partial^2 V(x,y)}{\partial y^2} = f(x,y) \tag{1}$$

Onde, neste caso, a função $f(x,y)$ é dada por:

$$f(x,y) = 7sin(2\pi x)cos(3\pi x)sin(2\pi y)cos(3\pi y) \tag{2}$$

Serão desenvolvidos programas paralelizados em `MPI`, que utilizam os métodos iterativos de *Jacobi* e *Gauss-Seidel* para encontrar soluções numéricas desta equação no domínio $-1 \leq x \leq 1$ e $-1 \leq y \leq 1$. Estes métodos discretizam o domínio nas direções x e y, formando uma 'grelha' de pontos onde a função $V(x,y)$ é calculada de forma iterativa. Para tal, as segundas derivadas precisam ser aproximadas por diferenças finitas em cada ponto da grelha, utilizando os valores da função nos seus pontos vizinhos. [2]

$$\frac{\partial^2 V(x,y)}{\partial x^2} \approx \frac{V(x-h,y) - 2V(x,y) + V(x+h,y)}{h^2} \tag{3}$$

$$\frac{\partial^2 V(x,y)}{\partial y^2} \approx \frac{V(x,y-h) - 2V(x,y) + V(x,y+h)}{h^2} \tag{4}$$

Onde h é o espaçamento uniforme entre pontos vizinhos na grade, tendo-se considerado igual em ambas direções.

As coordenadas dos pontos da 'grelha' são dadas por $x = -1+jh$ e $y = 1-ih$, através dos índices $j = 0, 1, ..., nx-1$ e $i = 0, 1, ..., ny-1$, onde $nx = ny = 2/h + 1$, neste caso. A equação resultante da substituição das segundas derivadas pelas suas aproximações e das coordenadas x e y pelos índices j e i, respetivamente, é expressa na forma iterativa do método de *Jacobi* por:

$$V_{i,j}^{(k)} = \frac{1}{4}[V_{i-1,j}^{(k-1)} + V_{i+1,j}^{(k-1)} + V_{i,j-1}^{(k-1)} + V_{i,j+1}^{(k-1)} - h^2 f_{i,j}] \tag{5}$$

Onde k representa a iteração atual. A tolerância ϵ é definida previamente, e considera-se que o método convergiu quando a seguinte condição é satisfeita:

$$\frac{\sqrt{\sum_{i,j}[V_{i,j}^{(k)} - V_{i,j}^{(k-1)}]^2}}{\sum_{i,j} V_{i,j}^{(k)}} < \epsilon \tag{6}$$

Tendo-se considerado, ao longo de todo o projeto, $\epsilon = 10^{-6}$

1..2 Método de Jacobi

Um dos métodos usados neste trabalho para resolver a equação diferencial de *Poisson*, Equação 1, é o método de *Jacobi*. Este método é um algoritmo iterativo que resolve sistemas de equações lineares e traz vantagens quando o problema em questão possui grandes dimensões e é um sistema esparso. Normalmente, este tipo de sistemas, que possuem uma grande proporção de entradas com valor zero, surgem na análise de equações diferenciais parciais.

1..3 Método Gausss-Seidel

Já o método de relaxação de *Gauss-Seidel* é também um método iterativo, apresentando-se como uma alternativa ao método de *Jacobi* devido a sua mais rápida convergência para a solução. Este método é uma modificação do método de *Jacobi* que consiste na utilização dos valores mais recentes disponíveis em cada momento. Por outras palavras, para os vizinhos cujo $V^{(k)}$ ainda não foi calculado na iteração k, utiliza-se $V^{(k-1)}$, tal como no método de *Jacobi* mas, quando o $V^{(k)}$ do vizinho já foi calculado na iteração atual, utiliza-se o valor mais recente, sendo esta atualização mais 'local' dos valores o que ajuda a acelerar a convergência do método.

2. Exercícios

2..1 Exercício a)

Este exercício tem como objetivo adaptar o programa escrito nas aulas de forma a incorporar as seguintes condições de fronteira:

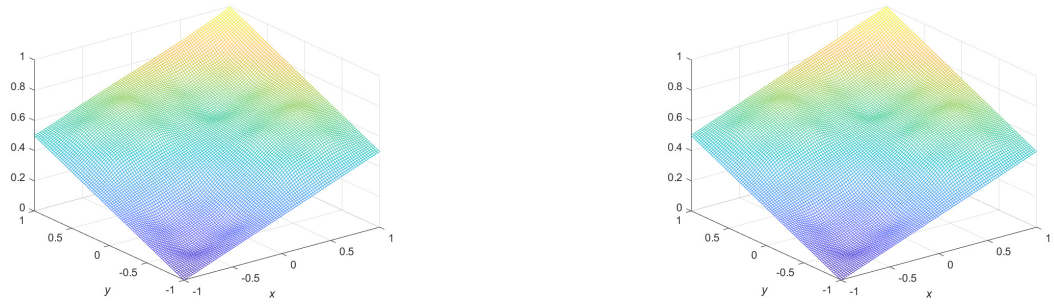
$$\begin{cases} V(-1, y) = (1 + y)/4 \\ V(x, -1) = (1 + x)/4 \\ V(x, 1) = (3 + x)/4 \\ V(1, y) = (3 + y)/4 \end{cases} \tag{7}$$

Para esta adaptação escreveu-se as linhas de códico abaixo, onde é possível observar que no primeiro ciclo *if* definiu-se a condição fronteira em $y = L$ e $y = -L$ e no segundo em $x = -L$ e $x = L$.

```
// Condicoes de fronteira em y=L e y=-L
if (newid == 0 || newid == 1){ // y = L, V(x,1)
    for (int j = 0; j < mycols+2; j++)
    {
        Vnew[0][j] = (1 + (-L + h*(firstcol+j-1)))/4;
        Vold[0][j] = Vnew[0][j];
    }
}
if (newid == nprocs - 2 || newid == nprocs - 1){ // y = -L, V(x,-1)
    for (int j = 0; j < mycols + 2; j++)
    {
        Vnew[myrows+1][j] = (3 + (-L + h*(firstcol+j-1)))/4;
        Vold[myrows+1][j] = Vnew[myrows+1][j];
    }
}

// Condicoes de fronteira em x=-L e x=-L
if (newid % 2 == 0)
{
    for (int i = 1; i < myrows + 1; i++) // x = -L, V(-1,y)
    {
        Vnew[i][0] = (1 + (-L + h*(firstrow+i-1)) )/4;
        Vold[i][0] = Vnew[i][0] ;
    }
}
else
{
  for (int i = 1; i < myrows + 1; i++) // x = L, V(1,y)
    {
        Vnew[i][mycols+1] = (3 + (-L + h*(firstrow+i-1)) )/4;
        Vold[i][mycols+1] = Vnew[i][mycols+1];
    }
}
```

Foi também realizada uma implementação do método de *Jacobi*, em `Matlab`, sem paralelização, seguindo as informações do enunciado deste projeto, de forma a poder ter um resultado teórico para comparar com os resultados obtidos através da implementação deste método paralelizado em `MPI`.

Assim, foi possível obter a Figura 1, onde podemos reparar que as formas das matrizes $V(x, y)$ são idênticas para ambas as implementações.



(a) (b)

Figura 1: Representação gráfica dos resultados obtidos no **Exercício a)**, em (a) `MPI` (experimentais) e (b) `Matlab` (teóricos), considerando $nx = ny = 100$

Procedeu-se também ao cálculo do erro quadrático médio (MSE), tendo-se obtido um valor de 6.06×10^{-10}. Visto este valor apresentar baixa magnitude, podemos concluir que o resultado obtido vai de acordo com o esperado. A equação utilizada para calcular este valor está apresentada na Equação 8.

$$MSE = \frac{1}{nx \times ny} \sum_{i=1}^{nx} \sum_{j=1}^{ny} (V(i,j)^{Matlab} - V(i,j)^{MPI})^2 \tag{8}$$

Relativamente ao número de iterações necessário para convergência e aos tempos de cálculo e escrita, em segundos, estes estão apresentados na Tabela 1, onde também referimos o número de processos considerado e os valores de nx e ny.

Implementação	Nº Iterações	Tempo de cálculo (s)	Tempo de escrita (s)	Nº processos	$nx = ny$
`MPI`	11804	0.546909	0.000796	4	100
`Matlab`	11759	7.630470	-	-	100

Tabela 1: Resultados obtidos no **Exercício a)**

Tal como representado na Tabela 1, o tempo de cálculo da implementação em `MPI` foi de 0.546909 segundos. Já a implementação em `Matlab` apresentou um tempo de cálculo de 7.630470 segundos, tendo-se assim obtido um *speedup* de 13.9520x. Desta forma, visto os resultados estarem de acordo com o esperado e o tempo de cálculo da implementação em `MPI` ter demonstrado um melhoramento, podemos concluir que esta implementação é uma mais valia e pode ser considerada em vez da implementação em `Matlab`. A partir desta tabela também é possível verificar que a implementação em `MPI` precisou de um maior número de iterações para convergir.

2..2 Exercício b)

Neste exercício o objetivo era adaptar o programa do exercício anterior para usar condições fronteira periódicas, em ambas as direções, x e y, esquecendo assim as condições fronteira

definidas no **Exercício a)** (eliminando as respetivas linhas no código).

Primeiramente, alterou-se a variável `periodic`, de forma a tornar a implementação das condições fronteira periódicas mais simplificada.

```
int periodic[2] = {1,1};
```

A definição do domínio de cálculo também foi alterada, relativamente ao que foi feito no **Exercício a)**. Visto estarmos a considerar condições fronteira periódicas é preciso incluir todos os pontos do domínio global, que não são pontos fantasmas, no domínio utilizado para efetuar os cálculos. O passo espacial h foi também alterado o passo : $h = 2.0 \times L/(nx)$.

```
//Linhas
for (int i = 0; i < nprocs_col; i++)
{
    listfirstrow[2*i] = i * (nrows);
    listmyrows[2*i] = nrows+1;
    listfirstrow[2*i+1] = i * (nrows);
    listmyrows[2*i+1] = nrows+1;
}

listfirstrow[nprocs-2] = 1 + (nprocs_col-1)*nrows;
listfirstrow[nprocs-1] = 1 + (nprocs_col-1)*nrows;
listmyrows[nprocs-2] = ny - (nprocs_col - 1) * nrows;
listmyrows[nprocs-1] = ny - (nprocs_col - 1) * nrows;

// Colunas
int ncols_temp = (int)((nx-2)/2);
for (int i = 0; i < nprocs_col; i++)
{
    listfirstcol[2*i] = 0;
    listmycols[2*i] = ncols_temp + 1;
    listfirstcol[2*i+1] = ncols_temp + 1;
    listmycols[2*i+1] = nx - 2 - ncols_temp;
}
```

Quanto à escrita do ficheiro, esta também sofreu alterações no código. Como agora estamos a considerar todos os pontos do domínio global (com exceção dos pontos fantasma), foi necessário criar um *datatype* que define a nova *view* para a escrita do arquivo. Desta forma, cada processo é responsável por escrever uma parte da matriz global no arquivo de saída.

```
int gsizes[2] = {ny, nx};
int lsizes[2] = {myrows, mycols};
int start_ind[2] = {firstrow, firstcol};
```

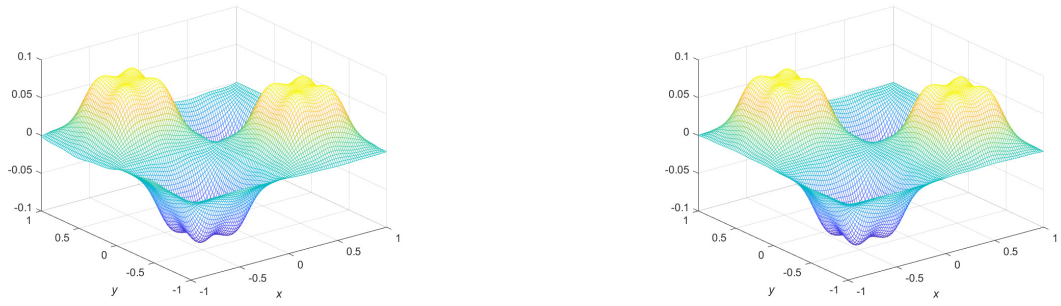
Por fim, quanto às comunicações entre os processos na direção 'vertical', a primeira linha do domínio é enviada para a última linha fantasma, e a última linha é enviada para

a primeira. O mesmo raciocínio se aplica na direção 'horizontal', ou seja, a primeira coluna do domínio (à esquerda) é enviada para a última coluna fantasma, e a última coluna (à direita) é enviada para a primeira.

```
// comunicacoes sentido descendente
MPI_Sendrecv(Vnew[1], mycols+2, MPI_DOUBLE, nbrbottom, 1, Vnew[myrows+1] ,
    mycols+2, MPI_DOUBLE, nbrtop, 1, comm2D, MPI_STATUS_IGNORE);

// comunicacoes sentido ascendente
MPI_Sendrecv(Vnew[myrows], mycols+2, MPI_DOUBLE, nbrtop, 0, Vnew[0] ,
    mycols+2, MPI_DOUBLE, nbrbottom, 0, comm2D, MPI_STATUS_IGNORE);

// comunicacoes sentido para esquerda
MPI_Sendrecv(&(Vnew[0][1]), 1, column, nbrleft, 3, &(Vnew[0][mycols+1]), 1,
    column, nbrright, 3, comm2D, MPI_STATUS_IGNORE);

// comunicacoes sentido para direita
MPI_Sendrecv(&(Vnew[0][mycols]), 1, column, nbrright, 2, &(Vnew[0][0]), 1,
    column, nbrleft, 2, comm2D, MPI_STATUS_IGNORE);
```

Comparando as Figuras 2a e 2b, que correspondem aos resultados obtidos através das implementações em `MPI` e `Matlab`, respetivamente, podemos concluir que a substituição das condições de fronteira explícitas por condições de fronteira periódicas alterou a morfologia do gráfico, e agora podemos observar uma certa periodicidade nos novos resultados mas que, mesmo assim, não existe uma diferença notória entre os resultados de ambas as implementações, o que é confirmado pelo baixo valor de MSE obtido, 2.26×10^{-5}.

Posto isto, podemos concluir que o resultado obtido com condições de fronteira periódicas se distância mais do resultado teórico em comparação com o caso em que as condições de fronteira não eram periódicas, **Exercício a)**. No entanto, como este valor continua relativamente baixo, ainda é possível considerar a implementação em `MPI` como uma boa aproximação do resultado teórico.



(a) (b)

Figura 2: Representação gráfica dos resultados obtidos no **Exercício b)**, em (a) `MPI` (experimentais) e (b) `Matlab` (teóricos), considerando $nx = ny = 100$

Implementação	Nº Iterações	Tempo de cálculo (s)	Tempo de escrita (s)	Nº processos	$nx = ny$
`MPI`	3845	0.185040	0.000824	4	100
`Matlab`	3840	3.692503	-	-	100

Tabela 2: Resultados obtidos no **Exercício b)**

Tal como no exercício anterior podemos comparar os tempos de cálculo, apresentados na Tabela 2, obtidos entre as implementações de forma a tirar conclusões. Os valores obtidos foram de 0.185040 (s), para a implementação em `MPI`, e 3.692503 (s), para a implementação em `Matlab`, tendo-se assim obtido um *speedup* de 19.9552x. Desta forma, visto os resultados estarem de acordo com o esperado e o tempo de cálculo da implementação em `MPI` ter sido inferior, podemos concluir que esta implementação se demonstra útil e eficaz, podendo ser considerada como alternativa à implementação em `Matlab`. Por fim, a partir desta tabela também é possível verificar que a implementação em `MPI` precisou de um maior número de iterações para convergir, tal como o que aconteceu no **Exercício a)**.

2..3 Exercício c)

Este exercício tem como objetivo alterar o código do exercício anterior, passando de um estêncil de 5 pontos para um estêncil de 9 pontos. Esta troca pode-se mostrar vantajosa pois, o uso de um estêncil de 9 pontos reduz a ordem do erro local de h^2 (5 pontos) para h^4.

Com a alteração para este novo estêncil temos 4 pontos adicionais a uma distância de $2h$. Quando aplicado ao método de Jacobi ponderado (aqui, a ponderação traz estabilidade ao método), este estêncil produz a seguinte equação iterativa:

$$\begin{aligned} V_{i,j}^{k} = \frac{w}{60}[&16V_{i-1,j}^{k-1} + 16V_{i+1,j}^{k-1} + 16V_{i,j-1}^{k-1} + V_{i,j+1}^{k-1} \\ &V_{i-2,j}^{k-1} - V_{i+2,j}^{k-1} - V_{i,j-2}^{k-1} - V_{i,j+2}^{k-1} - 12h^2 fi, j] + (1-w)V_{i,j}^{k-1} \end{aligned} \tag{9}$$

Onde $w = 15/16$.

Devido a existirem mais 4 pontos a serem utilizados para o cálculo, são agora necessários 4 pontos fantasmas em vez de 2, como nos exercícios anteriores. Desta forma, é necessário alocar espaço para 4 linhas fantasmas em vez de 2.

```
// Alocar memoria para as matrizes
double (*Vold)[mycols+4], (*Vnew)[mycols+4], (*myf)[mycols+4];
Vold = calloc(myrows + 4, sizeof(*Vold));
Vnew = calloc(myrows + 4, sizeof(*Vnew));
myf = calloc(myrows + 4, sizeof(*myf));
...
MPI_Type_vector(myrows + 4, 1, mycols + 4, MPI_DOUBLE, &column);
```

Ora, como as linhas fantasma não fazem parte do domínio utilizado para o cálculo, é necessário ajustar os índices no cálculo da função myf e na implementação do método de Jacobi (passando de $+1$ para $+2$). O código a seguir demonstra as alterações feitas:

```
// Calculo do myf
for (int j = 2; j < mycols + 2 ; j++)
{
    for (int i = 2; i < myrows + 2; i++)
    {
        myf[i][j] = f(-L + (firstcol + j - 2) * h, -L + (firstrow + i - 2) * h);
    }
}
...
double sums[2] = {0.0,0.0};
for (int j = 2; j < mycols + 2 ; j++)
{
    for (int i = 2; i < myrows + 2; i++)
    {
        Vnew[i][j] = (W/60)*(16*Vold[i-1][j] + 16*Vold[i+1][j] +
            16*Vold[i][j-1]+ 16*Vold[i][j+1]
             - Vold[i-2][j] - Vold[i+2][j] - Vold[i][j-2] - Vold[i][j+2]
                 -12*h*h*myf[i][j]) + (1-W)*Vold[i][j];
        sums[0] += (Vnew[i][j] - Vold[i][j]) * (Vnew[i][j] - Vold[i][j]);
        sums[1] += Vnew[i][j] * Vnew[i][j];
    }

}
```

Na escrita do ficheiro, também é necessário considerar os pontos que serão enviados e fazer os ajustes adequados.

```
// Alterado de +2 para +4
int memsizes[2] = {myrows+4, mycols+4};
start_ind[0] = 1;
start_ind[1] = newid % 2;
```

Neste caso é também necessário criar comunicações adicionais para realizar as transferências dos dados. Na direção "vertical", a informação da primeira linha será enviada para a penúltima linha fantasma, e a última linha do domínio de cálculo será enviada para a primeira linha fantasma. O mesmo raciocínio se aplica às colunas na direção "horizontal". Em seguida, o mesmo processo será repetido para os pontos restantes. Na direção "vertical", a informação da segunda linha será enviada para a última linha fantasma, e a penúltima linha do domínio de cálculo será enviada para a segunda linha fantasma, aplicando o mesmo raciocínio às colunas na direção "horizontal". Estas comunicações adicionais são necessárias para garantir a consistência dos dados e permitir os cálculos adequados nas iterações subsequentes.

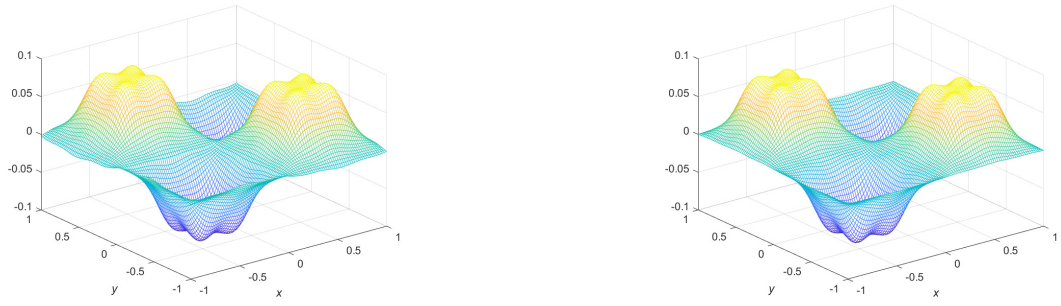
```
// Comunicacoes sentido ascendente
MPI_Sendrecv(Vnew[myrows], mycols+4, MPI_DOUBLE, nbrtop, 0, Vnew[0] ,
    mycols+4, MPI_DOUBLE, nbrbottom, 0, comm2D, MPI_STATUS_IGNORE);

// Comunicacoes sentido descendente
MPI_Sendrecv(Vnew[2], mycols+4, MPI_DOUBLE, nbrbottom, 1, Vnew[myrows+2] ,
    mycols+4, MPI_DOUBLE, nbrtop, 1, comm2D, MPI_STATUS_IGNORE);

// Comunicacoes sentido para direita
MPI_Sendrecv(&(Vnew[0][mycols]), 1, column, nbrright, 2, &(Vnew[0][0]), 1,
    column, nbrleft, 2, comm2D, MPI_STATUS_IGNORE);

// Comunicacoes sentido para esquerda
MPI_Sendrecv(&(Vnew[0][2]), 1, column, nbrleft, 3, &(Vnew[0][mycols+2]), 1,
    column, nbrright, 3, comm2D, MPI_STATUS_IGNORE);

// Outras comunicacoes
MPI_Sendrecv(Vnew[myrows+1], mycols+4, MPI_DOUBLE, nbrtop, 6, Vnew[1],
    mycols+4, MPI_DOUBLE, nbrbottom, 6, comm2D, MPI_STATUS_IGNORE);

MPI_Sendrecv(Vnew[3], mycols+4, MPI_DOUBLE, nbrbottom, 5, Vnew[myrows+3],
    mycols+4, MPI_DOUBLE, nbrtop, 5, comm2D, MPI_STATUS_IGNORE);

MPI_Sendrecv(&(Vnew[0][3]), 1, column, nbrleft, 7, &(Vnew[0][mycols+3]), 1,
    column, nbrright, 7, comm2D, MPI_STATUS_IGNORE);

MPI_Sendrecv(&(Vnew[0][mycols+1]), 1, column, nbrright, 8, &(Vnew[0][1]), 1,
    column, nbrleft, 8, comm2D, MPI_STATUS_IGNORE);
```

Por fim, é necessário fazer um ajuste nos índices utilizados para atualizar o valor de $Vold$, de modo a incluir todos os pontos do domínio global. Isso significa que, ao realizar a atualização de $Vold$ durante as iterações, os índices devem levar em consideração todos os pontos do domínio global, incluindo todas as linhas fantasmas.

```
// Alterado de +2 para +4
for (int i = 0; i < myrows + 4; i++)
{
```

```
    for (int j = 0; j < mycols + 4; j++)
    {
        Vold[i][j] = Vnew[i][j];
    }
}
```

Com estas alterações foi possível obter o gráfico representado na Figura 3a. Ao comparar estes resultados com os resultados teóricos, Figura 3b, é possível reparar que apesar da forma, em geral, ser semelhante, podemos observar, nos resultados experimentais, uma ondulação nas fronteiras, algo que não acontece nos resultados teóricos. Esta diferença é confirmada pelo valor do MSE obtido, 7.45×10^{-5}, que é um valor maior que aqueles obtidos anteriormente.



(a) (b)

Figura 3: Representação gráfica dos resultados obtidos no **Exercício c)**, em (a) `MPI` (experimentais) e (b) `Matlab` (teóricos), considerando $nx = ny = 100$

Apesar de o resultado obtido ainda poder ser considerado uma boa aproximação do valor teórico, não pode ser considerado um sucesso em relação à expectativa de um erro menor. Isto pois, tal como referido anteriormente, esperava-se que a nova implementação, com o estêncil de 9 pontos, reduzisse a ordem do erro local para h^4, o que seria menor do que a ordem do erro local da implementação anterior com o estêncil de 5 pontos (h^2). Isto não aconteceu, tal como se pode ver pelos valores para o erro relativo, $Err(\%)$, obtidos. Estes valores foram de 19.71%, para o **Exercício c)**, e 10.67%, para o **Exercício b)**, e foram obtidos a partir da seguinte equação:

$$Err(\%) = \frac{1}{nx \times ny} \sum_{i=1}^{nx} \sum_{j=1}^{ny} \frac{|V(i,j)^{Matlab} - V(i,j)^{MPI}|}{V(i,j)^{Matlab}} \times 100 \tag{10}$$

Os restantes resultados estão apresentados na Tabela 3, onde podemos ver que este método precisou de um maior número de iterações para convergir, relativamente ao método do exercício anterior. Quanto ao *speedup* obtido, este foi de 16.9281x, o que demonstra, novamente, a vantagem da computação paralela.

Implementação	Nº Iterações	Tempo de cálculo (s)	Tempo de escrita (s)	Nº processos	$nx = ny$
`MPI`	4966	0.337381	0.000933	4	100
`Matlab`	4925	5.711219	-	-	100

Tabela 3: Resultados obtidos no **Exercício c)**

2..4 Exercício d)

Por fim, neste exercício é pedido que se resolva o mesmo problema do **Exercício b)**, mas usando o método de *Gauss-Seidel* em vez do método de *Jacobi*. Este método, tal como referido na **Introdução**, utiliza os valores mais recentes disponíveis em cada momento, o que significa que, para os vizinhos cujo $V(k)$ ainda não tenha sido calculado na iteração k, utiliza-se $V(k-1)$, da mesma forma que no método de *Jacobi*. Porém, quando $V(k)$ do vizinho já foi calculado na iteração atual, utiliza-se esse valor mais recente. Com isto, é esperado que o método convirja mais rapidamente, isto é, com um menor número de iterações.

No entanto, num programa paralelizado, não seria possível usar a Equação 11 para todos os pontos, uma vez que, num programa deste tipo, não há garantia sobre a ordem em que os cálculos são realizados pelos diferentes processos. Isto significa que, num determinado ponto do programa, pode ocorrer que o valor $V^{k}_{i+1,j}$ já tenha sido atualizado num processo, enquanto noutro processo ainda é usado o valor antigo, $V^{k-1}_{i+1,j}$. Esta inconsistência na ordem dos cálculos pode levar a resultados incorretos ou inconsistentes ao usar a equação iterativa de *Gauss-Seidel* num programa paralelizado.

$$V^{(k)}_{i,j} = \frac{1}{4}[V^{(k)}_{i-1,j} + V^{(k)}_{i+1,j} + V^{(k)}_{i,j-1} + V^{(k)}_{i,j+1} - h^2 f_{i,j}] \tag{11}$$

Posto isto, a alternativa encontrada é a de utilizar um esquema de par/ímpar ou vermelho/preto, em que os pontos da mesma iteração são usados para calcular o ponto atual. Esta abordagem de par/ímpar é uma estratégia iterativa que permite que os pontos sejam atualizados de forma alternada, usando os valores mais recentes dos seus pontos vizinhos. [8]

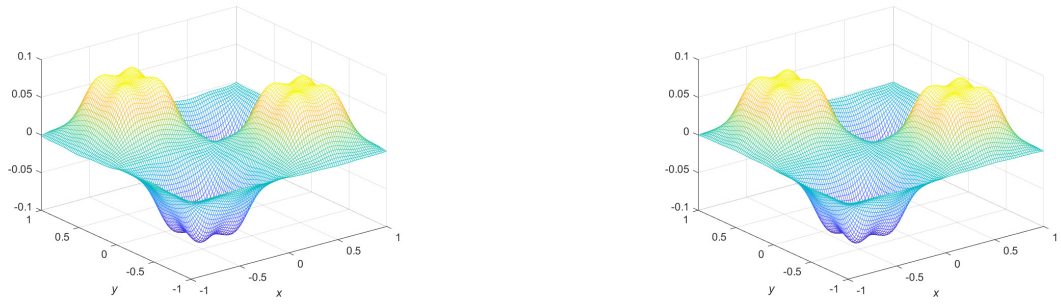
Neste caso, estamos novamente a considerar apenas mais duas linhas e colunas fantasmas, mantendo as condições de fronteira periódicas, exatamente como no **Exercício b)**. Dentro do domínio local os pontos pares são, em primeiro lugar, calculados e, em seguida, a informação é enviada para os vizinhos, sendo os pontos ímpares, posteriormente calculados. No final, a informação é novamente enviada para os vizinhos da mesma maneira que foi feita no **Exercício b)**.

```
// Calcular pares
for (int i = 1; i < myrows + 1; i++)
{
    for (int j = 1; j < mycols + 1 ; j++)
    {
        if (((firstcol + j - 1) + (firstrow + i - 1)) %2 == 0) // Verficacao
            que e par
```

```
        {
            Vnew[i][j] = (Vnew[i+1][j] + Vnew[i-1][j] + Vnew[i][j+1] +
                Vnew[i][j-1] - h * h * myf[i][j]) / 4.0;
            sums[0] += (Vnew[i][j] - Vold[i][j]) * (Vnew[i][j] - Vold[i][j]);
            sums[1] += Vnew[i][j] * Vnew[i][j];
        }
    }
}

// Comunica aos vizinhos
MPI_Sendrecv(&Vnew[1][1], mycols, MPI_DOUBLE, nbrbottom, 4,
    &Vnew[myrows+1][1], mycols, MPI_DOUBLE, nbrtop, 4, comm2D,
    MPI_STATUS_IGNORE);

MPI_Sendrecv(&Vnew[myrows][1], mycols, MPI_DOUBLE, nbrtop, 5, &Vnew[0][1],
    mycols, MPI_DOUBLE, nbrbottom, 5, comm2D, MPI_STATUS_IGNORE);

MPI_Sendrecv(&Vnew[1][1], 1, column, nbrleft, 6, &Vnew[1][mycols+1], 1,
    column, nbrright, 6, comm2D, MPI_STATUS_IGNORE);

MPI_Sendrecv(&Vnew[1][mycols], 1, column, nbrright, 7, &Vnew[1][0], 1, column,
    nbrleft, 7, comm2D, MPI_STATUS_IGNORE);

// Calcular impares
for (int i = 1; i < myrows + 1; i++)
{
    for (int j = 1; j < mycols + 1 ; j++)
    {
        if (((firstcol + j - 1) + (firstrow + i - 1)) %2 == 1) // verifca que e
            impar
        {
            Vnew[i][j] = (Vnew[i-1][j] + Vnew[i][j-1] + Vnew[i][j+1] +
                Vnew[i+1][j] - h * h * myf[i][j]) / 4.0 ;
            sums[0] += (Vnew[i][j]-Vold[i][j])*(Vnew[i][j]-Vold[i][j]);
            sums[1] += Vnew[i][j]*Vnew[i][j];
        }
    }
}

// Comunica aos vizinhos
MPI_Sendrecv(&Vnew[1][1], mycols, MPI_DOUBLE, nbrbottom, 8,
    &Vnew[myrows+1][1], mycols, MPI_DOUBLE, nbrtop, 8, comm2D,
    MPI_STATUS_IGNORE);

MPI_Sendrecv(&Vnew[myrows][1], mycols, MPI_DOUBLE, nbrtop, 9, &Vnew[0][1],
    mycols, MPI_DOUBLE, nbrbottom, 9, comm2D, MPI_STATUS_IGNORE);

MPI_Sendrecv(&Vnew[1][1], 1, column, nbrleft, 10, &Vnew[1][mycols+1], 1,
    column, nbrright, 10, comm2D, MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(&Vnew[1][mycols], 1, column, nbrright, 11, &Vnew[1][0], 1,
    column, nbrleft, 11, comm2D, MPI_STATUS_IGNORE);
```

Os resultados obtidos estão representados na Figura 4, onde podemos concluir que os resultados da implementação em `MPI`, Figura 4a, vão de acordo com os resultados teóricos, Figura 4b. Isto é confirmado pelo valor do MSE obtido, 2.11×10^{-5}, sendo, inclusivamente, mais baixo que aquele obtido no **Exercício b)**.



(a) (b)

Figura 4: Representação gráfica dos resultados obtidos no **Exercício d)**, em (a) `MPI` (experimentais) e (b) `Matlab` (teóricos), considerando $nx = ny = 100$

Finalmente, ao analisar as iterações necessárias para a convergência, Tabela 4, também é possível confirmar que o método foi aplicado com sucesso, visto terem sido necessárias muitas menos iterações para alcançar o resultado final, em comparação com o **Exercício b)**, tal como o esperado.

Implementação	Nº Iterações	Tempo de cálculo (s)	Tempo de escrita (s)	Nº processos	$nx = ny$
`MPI`	2113	0.144237	0.000853	4	100
`Matlab`	2101	3.830999	-	-	100

Tabela 4: Resultados obtidos no **Exercício d)**

Por fim, com a implementação paralelizada foi possível obter um *speedup* de 26.5604x, evidenciando-se assim, novamente, a mais valia que a computação paralela pode trazer.

3. Influência do número de processos

Foi também realizado um estudo da influência do número de processos no tempo de cálculo, para cada problema, onde se variou o número de processos como 2, 4, 8 e 16, estando os resultados obtidos apresentados na Tabela 5.

Com base nesta tabela vemos que, em geral, para todos os exercícios, à medida que o número de processos aumenta, o tempo de execução tende a diminuir. No entanto, nem

Nº Processos / Exercício	2	4	8	16	nx=ny
Exercício a)	0.672839	0.546909	0.387554	1.222988	100
Exercício b)	0.218008	0.185040	0.137032	0.344717	100
Exercício c)	0.404828	0.337381	0.256297	0.470550	100
Exercício d)	0.182757	0.144237	0.107627	0.295454	100

Tabela 5: Tempos de execução para cada exercício, variando o número de processos considerados

sempre acontece melhorias, como se pode ver quando se aumenta o número de processos de 8 para 16, o que sugere que existe um 'ponto de equilíbrio' em que a introdução de mais processos não traz benefícios adicionais.

Assim, podemos concluir que, aumentar o número de processos nem sempre traz vantagens significativas, e é importante considerar as características específicas do problema como, por exemplo, a complexidade computacional, a escalabilidade do algoritmo e os recursos disponíveis, antes de decidir sobre o número de processos a serem utilizados.

4. Conclusão

As soluções desenvolvidas para os problemas propostos neste projeto, quando analisadas individualmente, demonstraram-se eficazes, tendo todas elas se aproximado dos valores teóricos, como evidenciado pelos valores do erro quadrático médio (MSE) obtidos para cada problema.

No entanto, os resultados obtidos para o **Exercício c)** não atenderam às expectativas visto que se esperava obter um erro menor, relativamente aquele que foi obtido no **Exercício b)**.

Já o **Exercício d)**, permitiu concluir que o método de *Gauss-Seidel* converge mais rapidamente do que o método de *Jacobi*, o que vai de acordo com o esperado.

Com o estudo da influência do número de processos no tempo do cálculo do problema, foi possível concluir que a computação paralela pode trazer vantagens, pois pode permitir obter tempos de execução mais baixos mas que é, no entanto, preciso ter em conta as características específicas do problema, como, por exemplo, a complexidade computacional e a escalabilidade do algoritmo antes de decidir o grau de paralelização (número de threads/processos).

Posto isto, com este projeto, foi possível concluir que a paralelização, quando bem implementada, é algo vantajoso no desenvolvimento de código/programas pois permite diminuir os tempos de execução destes, fazendo uso dos recursos disponíveis que, por muitas vezes, não são utilizados.

Referências

[1] MPI: A Message-Passing Interface Standard. Version 4.0. Message Passing Interface Forum. June 9, 2021

[2] Americo, Rui. Enunciado Projeto 2: Paralelização em MPI de métodos iterativos para resolução de equações diferenciais a duas dimensões. Computação Paralela 2022/23.

[3] Americo, Rui. "MPI - Aula 1". Computação Paralela 2022/23.

[4] Americo, Rui. "MPI - Aula 2". Computação Paralela 2022/23.

[5] Americo, Rui. "MPI - Aula 3". Computação Paralela 2022/23.

[6] Americo, Rui. "MPI - Aula 4". Computação Paralela 2022/23.

[7] Americo, Rui. "MPI - Aula 5". Computação Paralela 2022/23.

[8] Jianping Zhu, Solving Partial Differential Equations on Parallel Computers, World Scientific, 1994.