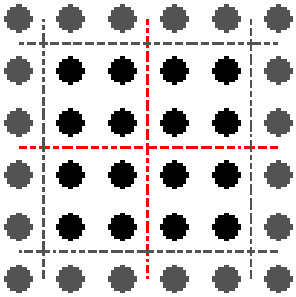

Paralelização MPI de métodos iterativos de resolução da equação de Poisson

Computação Paralela

Francisco Resende 84767

Departamento de Física
Universidade de Aveiro
03/07/2020

Conteúdo

1 Introdução **1**

2 Divisão do domínio global em subdomínios **2**

3 Criação de códigos de paralelização **2**
3.1 Escrita de resultados num ficheiro binário (alínea a) 2
3.2 Alteração das condições fronteira (alínea b) 4
3.3 Método de Jacobi com discretização com um estêncil de 9 pontos (alínea c) 5
3.4 Método de Gauss-Seidel paralelizado (alínea d) 7

4 Conclusão **9**

Lista de Figuras

1 Decomposição do domínio 2
2 Representação gráfica com as novas condições fronteira para 4 processos e um domínio global de 100x100. 4
3 Representação gráfica com as condições fronteira antigas para 4 processos e um domínio global de 100x100. 4
4 Representação das comunicações necessárias para o algorítmo. 5
5 Gauss-Seidel num programa não paralelizado. 7
6 Gauss-Seidel num programa paralelizado. 8

1 Introdução

De forma a tentar perceber o porquê de existir a necessidade de aplicar paralelização dos métodos iterativos que resolvem a equação de Poisson, é necessário entender o que é a equação de Poisson e como funcionam os métodos iterativos que a resolvem. A equação de Poisson é uma equação diferencial que é comummente utilizada em áreas como a eletrostática, física teórica e engenharia mecânica, a equação de Poisson usada para este trabalho pode ser representada pela equação 1.

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

onde

$$f(x,y) = 2 - x^2 - 10y + 50xy \tag{2}$$

Este tipo de equações de Poisson caracteriza-se, em 2D, como a soma das segundas derivadas parciais espaciais, sendo ela igual a uma função que depende das variáveis espaciais como se pode ver pela equação 1 e pela equação 2.

Dentre os variados métodos iterativos usados para resolução de sistemas lineares, pode-se destacar os métodos de Jacobi e Gauss-Siedel. O método de Jacobi apesar de apresentar uma convergência relativamente lenta, tem um bom funcionamento para sistemas esparsos. No entanto, este método não se comporta bem para todos os casos, para obter um melhor desempenho este método requer que os elementos da diagonal principal sejam não nulos. Mas para que este método seja aplicado à equação diferencial é preciso realizar a sua discretização, para isso foi aplicada uma aproximação por diferenças finitas com um estêncil de 5 pontos, como podemos ver na equação 3 e com um estêncil de 9 pontos como demonstrado na equação 4. Sendo esta aproximação um método de discretização, significa que transforma uma função contínua numa representação discreta, ou seja uma representação ponto a ponto.

$$V^{(k+1)}(i,j) = \frac{1}{4}[V^{(k)}(i-1,j) + V^{(k)}(i,j-1) + V^{(k)}(i,j+1) + V^{(k)}(i+1,j) - h^2 f(i,j)] \tag{3}$$

$$\begin{aligned} V^{(k+1)}(i,j) = \frac{1}{20}[&V^{(k)}(i-1,j-) + 4V^{(k)}(i-1,j) + V^{(k)}(i-1,j+1) + 4V^{(k)}(i,j-1) \\ &+4V^{(k)}(i,j+1) + V^{(k)}(i+1,j-1) + 4V^{(k)}(i+1,j) + V^{(k)}(i+1,j+1)] \\ &-\frac{h^2}{40}[f(i-1,j) + f(i,j-1) + 8f(i,j) + f(i,j+1) + f(i+1,j)] \end{aligned} \tag{4}$$

O método de Gauss-Seidel, o segundo método usado neste trabalho, consiste num melhoramento do método de Jacobi que para calcular cada $V^{(k+1)}(i,1)$ usa a solução atual dos quatro vizinhos caso ela já tenha sido calculada, ou seja usa $V^{(k+1)}$ se j<i e $V^{(k+1)}$ se j>i. A expressão para este método pode ser representada pela equação 5

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

Apesar do método usar valores calculados na iteração atual e outros da iteração anterior, os primeiros valores calculados de cada iteração irão usar valores da iteração passada. Se este último método não for associado a um outro método de auxilio, não irá funcionar quando num programa paralelizado.

Como é possível ver pelas equações 3, 4 e 5 ambos os métodos percorrem todos os pontos do sistema o que faz com que o programa tenha uma complexidade de $n_x * n_y$, onde n_x e n_y são as dimensões do sistema. Como este tipo complexidade computacional aparece a necessidade de dividir o domínio global do sistema em subdomínios e para isso realizar a paralelização do programa. Neste trabalho a paralelização foi totalmente realizada em MPI.

2 Divisão do domínio global em subdomínios

Para que os métodos possam ser implementados de forma paralela, o domínio global tem de ser dividido em subdomínios locais. Estes subdomínios locais estarão distribuídos por cada processo de forma a agilizar os cálculos dos métodos, como pode ser visto na figura 1.

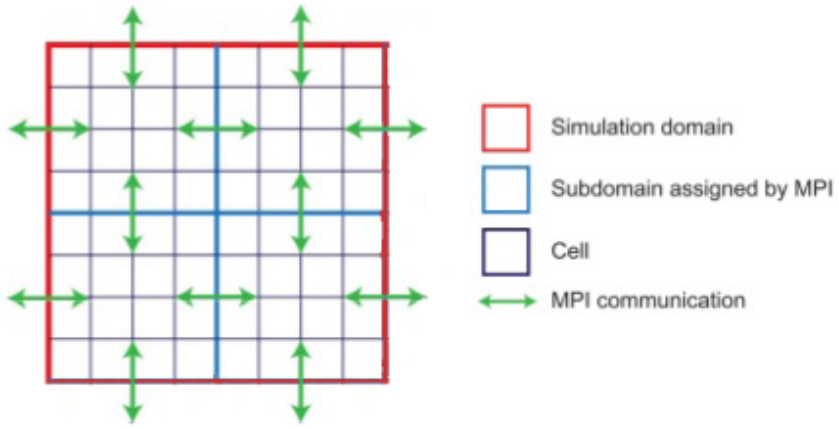

Figura 1: Decomposição do domínio

Como demonstra a figura 1 o domínio é dividido pelos vários processos, mas mantendo comunicação com os diversos processos pois são necessários valores dos processos vizinhos de cada processo para que os valores das fronteira entre processos sejam calculados.

3 Criação de códigos de paralelização

Tendo em conta o que foi falado na secção 2 e o guião do trabalho, foram implementados códigos para escrita de resultados num ficheiro binário, um código que comportava condições fronteira diferentes, método de Jacobi que usava discretização com um estêncil de 9 pontos e um algoritmo que possibilitasse a implementação paralelizada do método de Gauss-Seidel.

3.1 Escrita de resultados num ficheiro binário (alínea a)

A escrita de resultados tem de ser realizada assim que o método convergir, mas nem todos os valores dos subdomínios podem ser escritos no ficheiro, visto que cada subdomínio comporta mais 2 linhas e 2 colunas. Estas linhas e colunas a mais servem para armazenar os valores das condições fronteira e ainda valores dos processos vizinhos que serão usados para cálculos. Tal como todos os valores não podem ser todos escritos, cada processo tem de

saber em que parte do ficheiro irá escrever. De forma a resolver este último problema foi implementado um subarray que indicaria a cada processo onde escrever no ficheiro. Tendo em conta o constructor de um subarray, o array *gsizes* foi inicializado com o tamanho do domínio global ($gsizes[0] = n_x$ e $gsizes[1] = n_y$), o array *lsizes* foi inicializado com $lsizes[0] = myrows$ e $lsizes[1] = mycols + 1$ (*myrows* e *mycols* contêm o número de linhas e colunas, respetivamente, em que o processo irá trabalhar), e o array $start\_indices$ com $start\_indices[0] = mytoprow + 1$ e $start\_indices[1] = 0$. Mas estes valores iniciais não satisfazem todos os processos, por isso foram implementadas condições *if* de forma a acertar alguns valores como pode ser visto nos pseudocódigos 1, 2 e 3.

Algorithm 1: Correção no $lsizes[0]$

Result: $lsizes[0] = myrows + 1$
if *newid==0 or newid==1 or newid==numprocs-2 or newid==numprocs-1* **then**
  $lsizes[0] = myrows + 1$;

Este algorítmo 1 trata dos processos que se encontram nos 4 cantos do domínio global, pois estes têm de escrever a linha que corresponde às condições de fronteira de cima e de baixo.

Algorithm 2: Correção no $start\_indices[0]$

Result: $start\_indices[0] = mytoprow$
if *newid==0 or newid==1* **then**
  $start\_indices[0] = mytoprow$;

Este algorítmo 2 trata dos processos de *newid* 0 e 1, pois estes têm de começar a escrever a partir da linha que corresponde às condições de fronteira de cima, linha índice 0.

Algorithm 3: Correção no $start\_indices[1]$

Result: $start\_indices[1]$*=0*
if *newid*$\%2! = 0$ **then**
  $start\_indices[1] = myleftcol + 1$;

Este algorítmo 3 trata dos processos de *newid* ímpar, pois estes têm de começar a escrever a partir do índice myleftcol+1 devido ao índice myleft col corresponder ao processo da esquerda no caso destes processos.

Com esta preparação de arrays foi realizada a criação do subarray, aberto/criado o ficheiro binário e utilizado este subarray como *file view* para cada processo. Após esta criação foi realizada a criação do subarray que irá especificar que dados dos subdomínios será escrito no ficheiro, inicializando *memsizes* com o tamanho do do array local de cada processo (*memsizes[0]=myrows+2* e *memsizes[1]=mycols+2*), o array *lsizes* é o mesmo do subarray anterior, e o array $start\_indices$ com $start\_indices[0] = 1$ e $start\_indices[1] = newid\%2$, pois só os processos com *newid* par precisam de escrever a primeira coluna que corresponde à condição de fronteira da esquerda. Mas como os valores iniciais de $start\_indices$ não satisfazem todos os processos, foi implementada uma condição *if* de forma a acertar estes valores como pode ser visto no pseudocódigo 4.

Algorithm 4: Correção no $start\_indices[0]$

Result: $start\_indices[0] = 0$
if *newid==0 or newid==1* **then**
  $start\_indices[0] = 0$;

Este algorítmo 4 trata dos processos de *newid* 0 e 1, pois estes têm de escrever a partir da linha 0 devido a essa linha conter os valores da condição fronteira de cima.

Após esta correção de valores o subarray foi criado e usado para realizar a escrita segundo o file view criado a partir do primeiro subarray.

3.2 Alteração das condições fronteira (alínea b)

Única e exclusivamente para esta alínea foi nos pedido para que usássemos as condições fronteira, $V(x = -1, y) = 1y, V(x = +1, y) = \frac{5}{2} + \frac{y}{2}, V(x, y = -1) = \frac{1}{2} + \frac{3x}{2}$ e $V(x, y = +1) = 2 + x$. Esta alteração resumiu-se a trocar o código das condições fronteira desejadas.

Com estas condições fronteira foi obtida a representação gráfica que pode ser vista na figura 3 e comparado com a representação gráfica com as condições iniciais originas da figura XXX.

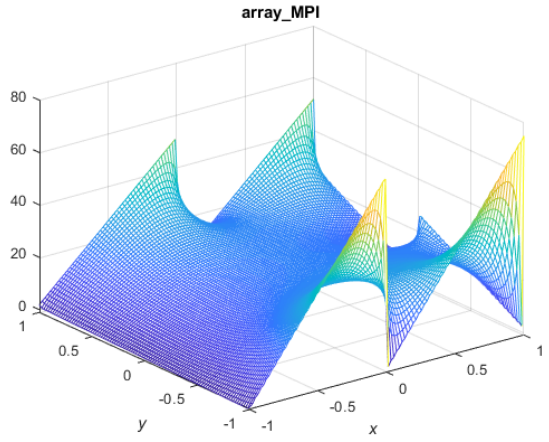

Figura 2: Representação gráfica com as novas condições fronteira para 4 processos e um domínio global de 100x100.

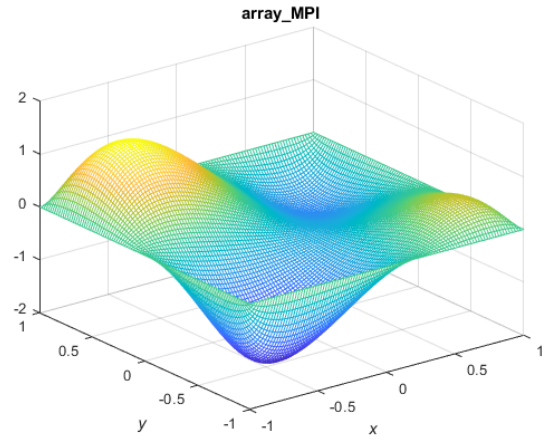

Figura 3: Representação gráfica com as condições fronteira antigas para 4 processos e um domínio global de 100x100.

3.3 Método de Jacobi com discretização com um estêncil de 9 pontos (alínea c)

Com objetivo de diminuir os erros da discretização foi pedido para que utilizássemos um discretização com um estêncil de 9 pontos como o da equação 4 da secção 1. Para que este novo algorítmo funcionasse para todos os pontos dos subdomínios, cada processo tem de conhecer não só os processos vizinhos de cima, esquerda, baixo e direita, mas também os vizinhos das diagonais. Esta necessidade pode ser vista na figura 4, onde a bola verde é o ponto que está a ser calculado, as bolas vermelhas os pontos comuns aos estêncil de 5 pontos e para quais as comunicações já estavam deitas, as bolas azuis os pontos novos que entram no estêncil de 9 pontos e dentro destas, a bola azul clara que faz parte de um processo do qual ainda não existia comunicação.

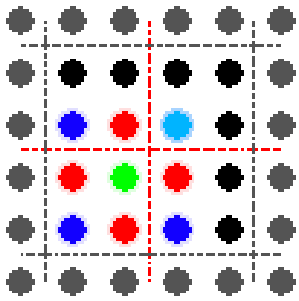

Figura 4: Representação das comunicações necessárias para o algorítmo.

A abordagem usada para encontrar os novos 4 vizinhos foi usado o comando MPI_Cart_coords que baseado nos *newids* do *comm2d* devolve as coordenadas como se os processos estivessem distribuídos numa grelha, estas coordenadas são armazenadas na variável *mycoords*. A partir destas coordenadas são calculadas as coordenadas dos 4 novos vizinhos (*topleftcoords*, *toprightcoords*, *bottomrightcoords* e *bottomleftcoords*), estes cálculos são realizados como os pseudocódigos 5, 6, 7 e 8. Estes vizinhos são considerados por *default* como MPI_PROC_NULL.

Algorithm 5: Coordenadas do processo vizinho de cima da esquerda

```
Result: topleftcoords
if newid%2! = 0 and newid!=1 then
    topleftcoords[0]=mycoordscoords[0]-1;
    topleftcoords[1]=mycoordscoords[1]-1;
```

O pseudocódigo 5 calcula as coordenadas do vizinho de cima da esquerda para todos os processos excepto para os de *newid* par e o processo 1, pois serão os únicos que não terão este vizinho.

Algorithm 6: Coordenadas do processo vizinho de cima da direita

```
Result: toprightcoords
if newid%2 == 0 and newid!=0 then
    toprightcoords[0]=mycoordscoords[0]-1;
    toprightcoords[1]=mycoordscoords[1]+1;
```

O pseudocódigo 6 calcula as coordenadas do vizinho de cima da direita para todos os processos excepto para os de *newid* ímpar e o processo 0, pois serão os únicos que não terão este vizinho.

Algorithm 7: Coordenadas do processo vizinho de baixo da esquerda

```
Result: bottomleftcoords
if newid%2! = 0 and newid!=numprocs-1 then
    bottomleftcoords[0]=mycoordscoords[0]+1;
    bottomleftcoords[1]=mycoordscoords[1]-1;
```

O pseudocódigo 7 calcula as coordenadas do vizinho de baixo da esquerda para todos os processos excepto para os de *newid* par e o processo numprocs-1, pois serão os únicos que não terão este vizinho.

Algorithm 8: Coordenadas do processo vizinho de baixo da direita

```
Result: bottomrightcoords
if newid%2 == 0 and newid!=numprocs-2 then
    bottomrightcoords[0]=mycoordscoords[0]+1;
    bottomrightcoords[1]=mycoordscoords[1]+1;
```

O pseudocódigo 8 calcula as coordenadas do vizinho de baixo da direita para todos os processos excepto para os de *newid* ímpar e o processo numprocs-2, pois serão os únicos que não terão este vizinho.

Após a obtenção dos 4 novos vizinhos foi realizada a comunicação entre os mesmos e armazenado o valor do vizinho no canto do array local referente ao vizinho, como por exemplo *mynew[myrows+1][mycols+1]* para o vizinho de baixo da direita.

3.4 Método de Gauss-Seidel paralelizado (alínea d)

Como é referido no guião, caso o programa não fosse paralelizado bastaria mudar o código referente à equação 3 e trocar o código referente a V^k e passar para V^{k+1}. Esta mudança pode ser realizada, pois como se pode ver pela figura 5 a partir do cálculo do ponto verde o método de Gauss-Seidel estará a seguir a equação 5, a utilização unicamente de V^{k+1} deve-se a dois fatores, um deles que inicialmente ambas as matrizes serem iguais e que no fim de cada iteração os valores da matriz V^{k+1} são transferidos para V^k. Ao utilizarmos um programa paralelizado irá aparecer problemas nas fronteiras dos processos, como pode ser visto na figura 6.

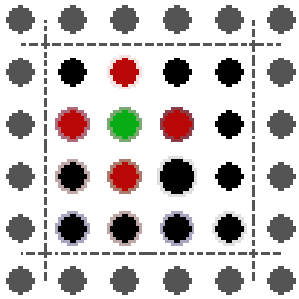

Figura 5: Gauss-Seidel num programa não paralelizado.

Na figura 6 está a ser considerado que ambos os 4 processos estão em determinada iteração a calcular o ponto que corresponde à bola verde, em vermelho estão assinalados os pontos que seriam considerados ”novos”e a azul os pontos considerados ”antigos”. Como é possível ver existem pontos que têm duas cores, o que faz com que a equação 5 não seja cumprida, ou seja não estaríamos perante o método de Gauss-Seidel.

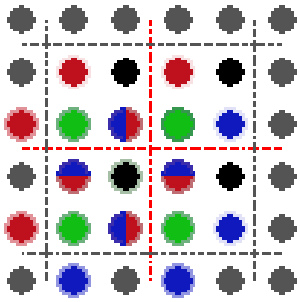

Figura 6: Gauss-Seidel num programa paralelizado.

De forma a contornar este problema, foi nos pedido para implementarmos o método de Gauss-Seidel com uma atualização do tipo vermelho-preto. Este tipo de atualização consiste em ”atribuir”a cor vermelho ou preto aos pontos de forma que os pontos vizinhos de um dado ponto tenham cor diferente desse mesmo ponto. Após esta ”atribuição”são calculados os valores de todos os pontos pretos, é feita a comunicação desses novos valores com os vizinhos, por seguinte é calculado os valores dos pontos vermelhos (este cálculo já usa valores ”novos”) e no fim é comunicado as mudanças dos pontos vermelhos.

Este tipo de atualização foi implementada como é mostrado nos pseudocódigos 9 e 10.

Algorithm 9: Cálculos dos valores dos pontos pretos.

Result: *mynew* dos pontos pretos
for *i=1;i<myrows+1;i++* **do**
  for $j = i\%2 == 0?2 : 1$*;j<mycols+1;j+=2* **do**
    mynew[i][j]=0.25*(*mynew[i-1][j]+mynew[i][j-1]+mynew[i][j+1]+mynew[i+1][j]-*h^2*myf[i][j]);
Comunicações;

O pseudocódigo 9 calcula os valores dos pontos pretos, irá percorrer todas as linhas e saltar as colunas com pontos vermelhos. Para que isto aconteça a primeira coluna será sempre a de índice 2 para linhas pares e a de índice 1 para linhas ímpares, o incremento realizado no índice das colunas é de 2. Após realizados os cálculos, são

efetuadas as comunicações.

Algorithm 10: Cálculos dos valores dos pontos vermelhos.

```
Result: mynew dos pontos pretos
for i=1;i<myrows+1;i++ do
    for j = i%2 == 0?1 : 2;j<mycols+1;j+=2 do
        mynew[i][j]=0.25*(mynew[i-1][j]+mynew[i][j-1]+mynew[i][j+1]+mynew[i+1][j]-h^2*myf[i][j]);
Comunicações;
```

O pseudocódigo 10 calcula os valores dos pontos vermelhos, irá percorrer todas as linhas e saltar as colunas com pontos pretos. Para que isto aconteça a primeira coluna será sempre a de índice 1 para linhas pares e a de índice 2 para linhas ímpares, o incremento realizado no índice das colunas é de 2. Após realizados os cálculos, são efetuadas as comunicações.

Após estas implementações o algorítmo de Gauss-Seidel pode ser usado em programas paralelizados.

4 Conclusão

Após as implementações e testes realizados, posso concluir que a maior diferença notada foi no número de iterações necessárias para que o método convirja. O algoritmo do esquema de atualização alternativo, vermelho-preto, poderia ser melhorado visto que as comunicações são de linhas e colunas completas, ou seja existem dados que serão repetidos e escritos no mesmo local. As maiores dificuldades enfrentadas neste trabalho residiram em tentar conjugar o MPI com os algorítmos que idealizava. Apesar das dificuldades enfrentadas penso que este trabalha contempla tópicos bastante interessantes e úteis às necessidades do quotidiano.