Laboratório de Programação Paralela

Trabalho prático - apresentação e resultados

Arthur De Oliveira Paiva – (116031056)
Lucio Henrik Amorim Reis – (116031051)
Rafael Duarte Campbell – (117031036)

Sumário

1 Proposta e projeto

Nesta seção, será apresentado o problema e suas características, bem como um projeto de paralelização.

1.1 Problema escolhido

Para este trabalho prático, pretende-se implementar – de forma linear e paralela – uma aplicação para **resolução de sistemas de equações lineares por eliminação gaussiana com pivoteamento parcial**. Além da grande aplicabilidade, a resolução de sistemas lineares é uma tarefa bastante paralelizável, ainda mais pelo método de eliminação gaussiana.

1.2 Sistemas de equações lineares

Equações lineares são, sobretudo, equações polinomiais que podem ser escritas na forma $a_1X_1 + a_2X_2 + ... + a_nX_n = b$, onde $a_1,\ a_2,\ ...,\ a_n$ e b são constantes e $X_1,\ X_2,\ ...,\ X_n$ são incógnitas. Um *sistema de equações lineares* é, essencialmente, um conjunto de equações lineares que diz respeito ao mesmo conjunto de incógnitas, sendo normalmente escrito como:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ a_{31}x_1 + a_{32}x_2 + ... + a_{3n}x_n = b_3 \end{cases} \tag{1}$$

É comum, ainda, que o sistema seja expresso na forma matricial $AX = B$, onde:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \tag{2}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{3}$$

$$B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{4}$$

A representação matricial é ainda mais valiosa para o método de eliminação gaussiana, dado que as operações são aplicadas exclusivamente à matriz A. Isto posto, usaremos a representação matricial tanto para exemplificar o método quanto para implementar o método.

1.3 Resolução do sistema

O método de *eliminação gaussiana* tem três etapas, sendo elas:

1. Obter uma matriz aumentada $[A|b]$, onde o sistema de equação seja $AX = b$.

2. Obter uma matriz equivalente $[A'|b']$, onde A' seja triangular superior.

3. Resolver o sistema de equações $A'X = b'$ por substituição regressiva.

A etapa mais expressiva – e paralelizável – do processo é a segunda, que consiste em aplicar sucessivas operações elementares até que se obtenha uma matriz triangular superior equivalente. Assumindo uma constante não-nula a e um sistema de equações lineares $\{L_1,\ L_2,\ \cdots,\ L_m\}$, as premissas elementares que norteiam o processo são:

- $L'_1 \equiv L_1 \times a$
- $L'_1 \equiv L_1 - L_2$

Como forma de zerar as constantes abaixo da diagonal principal, executa-se:

```
Para cada coluna i, faça:
    Para cada linha L_k, se k > i, faça:
        L'_k ← L_k − (L_i × (a_ki/a_ii))
```

A série de operações que se faz para cada coluna i tem como objetivo zerar todas as constante abaixo de a_{ii}, chamado **pivô**. Entretanto, este método depende que a_{ii} seja não-nulo – e expressivo – para todas as k linhas da iteração, como forma de evitar falhas no cálculo. Uma forma de otimizar o processo – e contornar as falhas citadas – é escolher o melhor pivô possível, ou seja, escolher o valor mais expressivo da coluna i; a esse processo, damos o nome de *pivoteamento parcial*.

A escolha do pivô ocorre a cada *i-ésima* execução , onde deve-se executar:

```
p = i //define-se pivô como i
Para cada linha L_j, se j > i, faça:
    Se a_ji > a_pi, faça:
        p = j //define-se um novo pivô
Se p ≠ i, faça:
    L_i ↔ L_p //permutar linhas
```

Caso um sistema consistente e uma matriz densa, o algoritmo apresentado obterá uma matriz triangular superior A' equivalente. Em seguida, deve-se executar o procedimento de substituição regressiva para obter os valores das incógnitas.

1.4 Paralelização

Das três etapas da resolução do sistema, apenas a segunda é paralelizável. Isto decorre do fato de que a primeira é inerente à representação do dado – neste caso, representação por uma estrutura matricial – e a terceira, de natureza sequencial, tem substituições que dependem da operação anterior. Dito isso, o trabalho de paralelização foca na segunda etapa, que é também a mais longa e expressiva do método.A segunda etapa, ainda sim, tem duas subetapas de natureza distinta: busca pelo pivô e o cálculo da linha L'_k.

1.4.1 Busca pelo pivô

Neste caso, basta agrupar todos os valores de uma coluna i e particioná-los entre os processos. Cada processo realizará uma comparação sequencial, buscando o maior valor; em seguida, todos os maiores valores são reagrupados e busca-se, entre estes, o maior – e não nulo.

1.4.2 Cálculo da linha L'_k

Uma vez escolhido o pivô, cada processo receberá a linha pivô L_i e um conjunto de linhas, devendo aplicar a fórmula $L'_k \leftarrow L_k - (L_i \times (a_{ki}/a_{ii}))$ a cada uma. A forma como é feita a distribuição das linhas não é um problema central da paralelização, mas é passível de otimização. Neste projeto, foi escolhido o modelo *cyclic striped*, que visa distribuir as linhas de cima para baixo alternando entre os processos. [1]

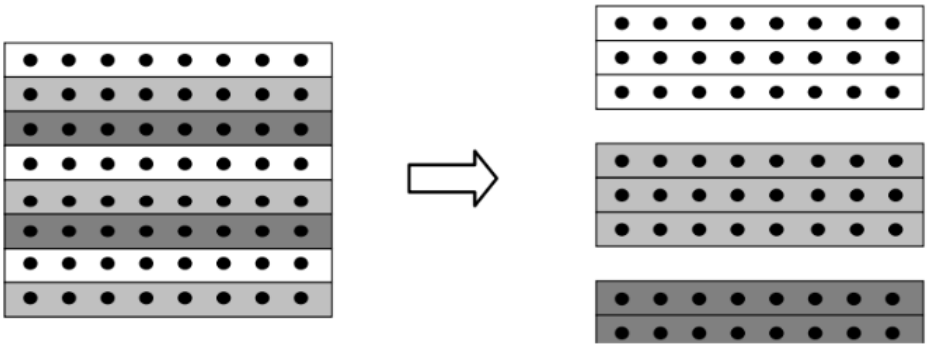

Figura 1: Representação da divisão entre 3 processos. [1]

1.5 Tecnologias e implementações

Para este projeto, serão implementadas quatro versões deste algoritmo. A primeira versão é sequencial e regular, será implementada em C e seguirá o algoritmo descrito na seção *Resolução do sistema*. As duas seguintes são paralelas e seguirão o modelo de paralelização descrito na seção anterior, devendo ser implementadas nos padrões MPI (*Message Passing Interface*) e OpenMP (*Open Multi-Processing*). A última versão também é paralela e seguirá as mesmas técnicas descritas, mas será implementada em C usando a biblioteca *pthread*. Ao fim, espera-se obter um comparativo de desempenho entre as quatro implementações, com testes para matrizes de diferentes tamanhos.

2 Resultados e conclusões

Nesta seção, serão apresentadas as implementações do projeto, códigos de apoio e um tutorial de execução dos programas. Ao fim, será realizado um comparativo entre as técnicas utilizadas, discutindo suas limitações.

2.1 Códigos de apoio

Para facilitar os testes, foram construídos alguns métodos de apoio para geração e leitura de sistemas de equações lineares. Para tornar todo o processo mais transparente, as matrizes geradas são salvas em arquivo e podem ser consultadas.

2.1.1 Geração de matrizes

Uma vez definida a matriz de variáveis (X), são gerados valores aleatórios dentro de um intervalo definido[1] para cada coluna da matriz A. Ao fim, somam-se os valores das colunas multiplicando-os pelas variáveis correspondentes e determina-se o valor para a matriz B.

O método pode receber diferentes variações de parâmetros. No primeiro caso, usa-se **"-n"** seguido de um valor inteiro (n) determinando o tamanho da matriz X. Os valores serão gerados sequencialmente, partindo de 1.

```
$./linearEquationSystemGenerator -n 3
```

Neste caso, os valores serão gerados sequencialmente, partindo de 1. Outra opção é passar os valores de X como parâmetros.

```
$./linearEquationSystemGenerator 2.2 4.1 5 2.2
```

Ao fim, imprime-se em tela as matrizes A e B geradas.

```
--- Matriz de coeficientes A ----
59.220000       5.110000        68.360000       89.050000
55.790000       11.560000       83.860000       22.280000
13.080000       67.990000       98.950000       89.580000
93.270000       44.600000       30.620000       42.130000

--- Matriz de resultado B ----
688.945000
638.450000
999.361000
633.840000
```

Ambas as matrizes serão armazenadas em um arquivo nomeado **randomMatrix.txt**. A primeira linha contem o tamanho da matriz, as seguintes definem os valores da matriz A. Ao fim, uma linha vazia e a linha com os valores de B.

[1] O intervalo varia entre 0 e VAL_RANGE, usando um fator $DECIMAL\_FACTOR$ que determina o número de casas decimais. Por padrão, usa-se 100 e 100 respectivamente, gerando valores entre $0,00$ e $100,00$.

```
4
59.22000 5.11000 68.36000 89.05000
55.79000 11.56000 83.86000 22.28000
13.08000 67.99000 98.95000 89.58000
93.27000 44.60000 30.62000 42.13000

688.94500 638.45000 999.36100 633.84000
```

2.1.2 Leitura das matrizes

Para facilitar a leitura dos arquivos, foi desenvolvido uma biblioteca que implementa o método **getMatrixAandB**.

```
int getMatrixAandB(int* n, double*** A, double** B)
```

Recebendo n – tamanho da matriz –, A – matriz de coeficientes – e B – matriz de resultados – como parâmetros, o método aloca espaço, lê o arquivo e atribui os valores lidos para cada um dos parâmetros recebidos.

Todas as implementações utilizam esta biblioteca para lerem os valores salvos no arquivo randomMatrix.txt, logo, basta utilizar o método gerador de matrizes para definir qual matriz será utilizada na eliminação gaussiana.

2.2 Implementações

Como proposto, foram feitas quatro implementações da eliminação gaussiana com pivoteamento parcial. Os métodos pode receber dois parâmetros (opcionais e inter-excludentes).

Normalmente, a execução imprimirá apenas o tempo de execução[2].

```
**** Cálculo realizado em 0.000489 segundos
```

Passando o parâmetro **"-r"**, a matriz X também será impressa ao fim da execução.

```
$./<implementacao> -r

*** Vetor X ***
X0 = 2.200000
X1 = 4.100000
X2 = 5.000000
X3 = 2.200000
**** Cálculo realizado em 0.000513 segundos
```

Com o parâmetro **"-p"**, imprime-se a matriz estendida $A|B$ lida, a matriz triangular estendida $A'|B'$ equivalente e o vetor X calculado.

```
$$./<implementacao> -p

*** Matriz A|B ****
59.220000 5.110000 68.360000 89.050000 688.945000
55.790000 11.560000 83.860000 22.280000 638.450000
13.080000 67.990000 98.950000 89.580000 999.361000
```

[2]Para estes exemplos, foi utilizada a implementação serial e a matriz gerada anteriormente

```
93.270000 44.600000 30.620000 42.130000 633.840000

*** Matriz Triangulada ****
93.270000 44.600000 30.620000 42.130000 633.840000
0.000000 61.735384 94.655912 83.671772 910.472534
0.000000 0.000000 88.723794 17.569225 482.271264
0.000000 0.000000 0.000000 77.021511 169.447324

*** Vetor X ***
X0 = 2.200000
X1 = 4.100000
X2 = 5.000000
X3 = 2.200000
**** Cálculo realizado em 0.001034 segundos
```

Todas as implementações têm as saídas padronizadas e podem receber os mesmos parâmetros.

2.2.1 Serial

A implementação serial seguiu o mesmo molde do algoritmo apresentado na proposta. Cabe aqui uma breve apresentação do algoritmo de *substituição regressiva* implementado.

```
void backSubstitution(int m,double **a,double *b,double* x)
{
    double sum;
    for(int i = m-1; i >= 0; i--){
        sum = 0;
        for(int j = i+1; j < m; j++){
            sum += a[i][j] * x[j];
        }
        x[i] = (b[i] - sum) / a[i][i];
    }
}
```

O primeiro passo é calcular a soma total de todas as variáveis já determinadas multiplicadas por seus coeficientes respectivos. Em seguida, subtrai-se essa soma do valor de B, restando apenas o valor equivalente à variável "pivô"da linha multiplicada por seu coeficiente. Desse modo, basta dividir o resultado obtido de B pelo coeficiente restante para obter a variável X que faltava.

Esse processo será executado para todas as linhas e, dado que a matriz equivalente obtida é triangular superior, a substituição deve ocorrer da última linha até a primeira – por este motivo, chama-se regressiva.

2.2.2 Pthreads

Conforme proposto, esta implementação paralela seguiu o molde da proposta, usando a modelo *cyclic striped* para divisão das partes.

Em suma, foi utilizada a função **pthread_create**, de acordo com número de threads, passando as informações necessárias para fazer as operações de pivoteamento – utilizando uma estrutura.

```
struct gauss_elimination_param{
    int col;
    int threadNumber;
    int matrixSize;
    double **matrix;
    double* arrayB;
};
```

A mesma estrutura é utilizada para chamar o método responsável pela eliminação gaussiana em si. Cada processo parte de um índice definido como **index = i+1+id**, onde *i* é o index do pivo atual e *id* é o número que indica a thread. O loop continua com o próximo index definido por **index = index + threads**, onde *threads* define o número de processos utilizadas.

Por fim, a função **pthread_join** é responsável por forçar o programa a esperar que todas as *threads* acabem seus processamentos antes de seguir com a execução ou finalização do programa.

2.2.3 MPI

Esta implementação utiliza a mesma biblioteca de leitura que as demais, salvo que apenas o processo mestre (número 0) contém as matrizes preenchidas. Para computar a matriz, foram utilizadas técnicas diferentes no pivoteamento parcial e na eliminação gaussiana.

Para pivoteamento, usa-se uma estrutura para armazenar o valor e o índex correspondente. O processo mestre primeiro percorre a coluna recuperando os valores e, por meio de uma chamada de **scatter**, esses valores são divididos entre os processos.

```
typedef struct pivotValueIndex{
    double val;
    int index;
} PIVOTVALUEINDEX;
```

Cada processo deve buscar dentro de sua parte o maior valor. Por fim, usa-se **reduce** com **MPI_MAXLOC** para encontrar o índice do valor mais significativo e chama-se **bcast** para compartilhá-lo com todos os processos.

Já para a Eliminação Gaussiana de fato, o processo mestre é responsável pelos envios, definindo um número de loops que seja múltiplo do número de processos e maior que a dimensão da matriz.

Para cada iteração, o processo mestre envia para outro processo – de modo cíclico – o índice da linha a ser enviada, o termo[3] já calculado, a linha que deve ser recalculada, a linha pivo, o valor de B da linha a ser alterada e o valor de B da linha pivô.

Cada processo, por sua vez, deve receber estes dados, recalcular cada coluna da linha de A recebida e, ainda, atualizar o valor de B recebido. Por fim, enviará de volta ao processo mestre – que receberá as mensagens de cada processo também de modo cíclico – o índice da linha calculada, a linha de A já atualizada e o valor de B atualizado. O processo mestre chama o método de *update* da matriz, passando as linhas recebidas.

Com a matriz já triangulada, chama-se o método de substituição regressiva assim como todas as demais implementações. Por ser um método pouco utilizado e

[3]Este é o coeficiente que multiplicará a linha pivô antes de subtraí-la da linha a ser alterada. Este termo, como definido na proposta, é dado por (a_{ki}/a_{ii}), onde a_{ki} é o valor da coluna pivô da linha a ser alterada e a_{ki} é o pivo escolhido.

inerentemente linear, optou-se por não implementar nenhum modo de paralelismo neste processo.

2.2.4 OMP

Dada a natureza das implementações de OpenMP, utilizou-se a implementação serial como base para a construção desta versão. Para a realização do pivoteamento parcial, usou-se uma estrutura – com valor e índice – e uma diretiva de *reduce* customizada.

```
    typedef struct Compare {
        double val;
        int index;
    } compare;
```

```
#pragma omp declare reduction(maximum : compare :
    omp_out = omp_in.val > omp_out.val ? omp_in : omp_out)
```

Esta diretiva, que recebe o nome de **maximum**, recebe um objeto *omp_in* do tipo **compare** e, caso este valor seja de fato maior que o valor já armazenado (*omp_out*), será definido como novo maior valor.

Deste modo, o pivoteamento parcial fica definido como um loop – similar ao utilizado na versão serial –, mas chamando a diretiva apresentada sobre um objeto da estrutura **compare**. Ao fim, verifica-se o índice do pivo obtido e, sendo diferente do pivô atual, as linhas são trocadas.

```
    compare max;
    max.val = fabs(a[i][i]);
    max.index = i;
    #pragma omp parallel for reduction(maximum:max)
    for(int k = i+1; k < m; k++){
        if(max.val < fabs(a[k][i])){ //valor absoluto maior
            max.val = fabs(a[k][i]);
            max.index = k;
        }
    }
```

Para a eliminação gaussiana, usa-se o mesmo código da versão serial, adicionando a diretiva para paralelizar um loop. Atenção à configuração de *schedule* definida, que garante a implementação do modelo *cyclic striped*.

```
#pragma omp parallel for private(term,k,j) schedule(static,1)
```

Assim como nos demais, a substituição regressiva não foi paralelizada.

2.3 Limitações e dificuldades

No geral, as implementações não representaram grande dificuldade de implementação. Deve-se, entretanto, informar que algumas limitações associadas ao *hardware* foram determinantes para os testes.

Dado que a máquina onde os testes foram realizados tem apenas 4 núcleos, todos as implementações definiram a contagem de processos em 4 – apresentando o resultado mais otimizado –.

Quanto à geração das matrizes, as primeiras versões utilizavam alocação direta, o que limitava a matriz a um valor máximo de **n=1023** antes de gerar erro por *stack overflow*. Alterando para alocação dinâmica, não ocorreram mais erros nesse

sentido e a limitação passou a ser por tempo de execução – tornando a execução bastante complicada a partir de $\mathbf{n = 3000}$).

Outra limitação que apareceu foi quanto ao consumo de memória por outros processos concorrentes. A execução serial, por ocupar apenas um núcleo, não apresentou grande variação de desempenho; as demais implementações paralelas tiveram resultados muito diferentes a depender de ter ou não outros programas executando na máquina. Em algun casos, era necessário reiniciar o computador para que os tempos voltassem ao normal.

2.4 Comparativo dos resultados

Para comparar os resultados, foram feitas cinco execuções de cada implementação para os tamanhos 10, 100, 500, 1000, 1500, 2000, 2500 e 3000. Ao fim, calcula-se uma média truncada –removendo o maior valor. Os resultados estão expressos na figura 2.

Tamanho	Serial	MPI	Pthread	OMP
10	0.000005	0.000838	0.001445	0.001980
100	0.002155	0.044342	0.020939	0.006778
500	0.251302	0.568266	0.529727	0.130077
1000	1.983525	9.063898	3.682738	1.023678
1500	6.699230	23.693881	12.798478	3.459354
2000	16.310575	44.483196	30.115219	8.148721
2500	30.926184	76.815634	59.849367	15.947293
3000	53.377539	127.159432	104.564415	27.473075

Tabela 1: Média dos tempos obtidos em segundos

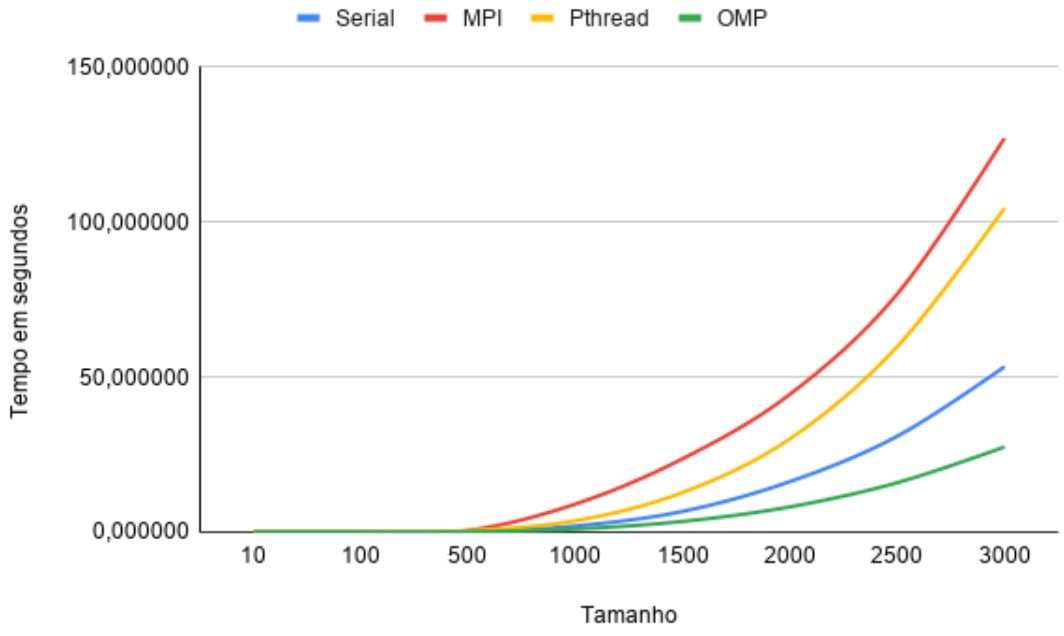

Figura 2: Comparativo de desempenho das implementações

Como podemos observar, os resultados são bastante próximos quando $\mathbf{n = 100}$ – avaliando em escala, as diferenças se mantém proporcionais –, mas divergem bastante a medida que a matriz cresce.

No geral, as aplicações com paralelismo têm um *overhead cost* associado a criação dos objetos e alocação de memória. É fácil concluir que, a medida que uma solução demanda mais alocação e mais comunicação – fora a computação em si –, o tempo tende a crescer.

A aplicação **MPI** inclui, além de muitos envios de mensagens entre processos, alocação e copia de vetores grandes durante toda a execução. Mesmo usando soluções que visassem diminuir a necessidade de cópias, é um problema inerente da memória distribuída. Neste caso, o ganho com paralelismo não consegue superar a perda com envio de mensagens.

Já para o caso **Pthread**, usa-se uma estrutura que deve ser alocada e preenchida em vários momentos durante a execução. Como não inclui tantos envios de mensagem e copias quanto a aplicação **MPI**, é natural que seu desempenho seja sensivelmente melhor. Ainda sim, inclui um custo grande com a criação das threads e alocação de espaço para estruturas.

O caso **OMP**, por sua vez, reduz ambos os principais problemas apresentados. Por se tratar de uma solução com memória compartilhada, seu custo não inclui envio de mensagens ou alocações desnecessárias, o que acaba deixando seu desempenho muito mais próximo do obtido na versão **serial** – especialmente se comparado com os demais –. Por outro lado, existe um ganho no uso de mais núcleos – por conta do paralelismo – que, associado ao baixo *overhead cost* de implementação, resulta em um desempenho melhor que a versão serial.

2.5 Considerações finais

Conforme discutimos, dado a característica do problema de estar diretamente ligado a um comportamento repetitivo de acesso a uma mesma memória, observamos que soluções que demandem cópias acabam resultando em tempos piores. Por outro lado, a solução de OpenMP, uma vez que não necessita de realocações das matrizes, acaba apresentando um ganho real no uso do paralelismo.

[1] *Parallel Methods for Solving Linear Equation Systems*. Disponível em https://drive.google.com/file/d/1hmitDI6WYPGkTsF3ZHpxTOhf5KElzBG/view. Acessado em 17/10/2020.