Implementação da Decomposição LU Paralelo Usando Openmp

Gissely de Souza - João Henrique Martins

28 de novembro de 2019

1 Decomposiçãao LU

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}$$

Figure 1: Matriz A = L * U

1.1 Eliminação de Gauss, matriz U

A eliminação gaussiana é um método para resolver sistemas lineares. Este método consiste em manipular o sistema através de determinadas operações elementares, transformando a matriz estendida do sistema em uma matriz triangular superior.

- 1.multiplicação de um linha por uma constante não nula;
- 2.substituição de uma linha por ela mesma somada a um múltiplo de outra linha;
- 3.permutação de duas linhas;

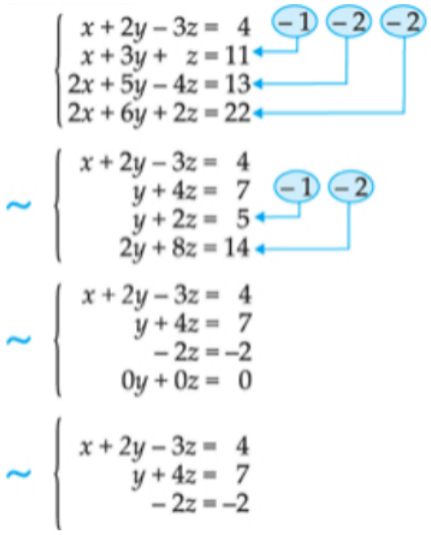

Figure 2: Etapas da Eliminação de Gauss

1.2 Matriz L

Os multiplicadores utilizados para zerar as linhas abaixo da diagonal principal no método da eliminaçõo de Gauss são organizados numa matriz L triangular inferior com diagonal unitária, ou seja:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ m_{21} & 1 & 0 & \dots & 0 \\ m_{31} & m_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \dots & 1 \end{bmatrix}$$

Figure 3: Matriz L

1.3 Aplicações

Matriz inversa. As matrizes L e U podem ser usadas para calcular a matriz inversa. Algumas implementaçõeses que invertem matrizes usam este método. No caso em que Ax=LUx=b x e b são tratados como vetores. Ao trocar o vetor x pela matriz X e o vetor b pela matriz B passamos a ter AX=LUX=B. Agora, supondo que B seja a matriz identidade, teremos entãoo que X será a inversa de A. Determinante de uma matriz. As matrizes L e U podem ser usadas para calcular o determinante da matriz A muito eficientemente porque

det(A)=det(L)det(U) e os determinantes de matrizes triangulares sãoo simplesmente o produto dos elementos de suas diagonais.

2 Complexidade e Paralelização

2.1 Sequencial

No processo de eliminaçãoao de gauss (matriz U) teremos (n*n+3n)*n, n+n flops. Ou seja complexidade de n, mais n para a matriz L:

2.2 Paralela

No processo de eliminação de gauss (matriz U) teremos (n/p)*ns. Ou seja complexidade de n/p, mais n/p para a matriz U.

Para a parelelização da eliminação gaussiana, obtenção da matriz U, foi particionado entre as threads o cálculo de obtenção das novas linhas, como para se obter o novo valor das linhas é necessário se ter feito o cálculo sobre as linhas anteriores a obtenção a determinação do pivô é feita sequêncialmente, após obter-se o pivô cada thread cálcula o valor de n/nthreads colunas. Na obtenção da Matriz L a paralelização acaba sendo maior por não existir dependência de dados, a matriz é dividida entre as threads e cada uma a preenche com 0's, 1's ou com os coeficientes de multiplicação da matriz U, o preenchimento da parte inferior da matriz e feito separadamente em um for com o offset(f) e a variavel das colunas (i) da matriz sendo variaves privadas.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char*argv[])
{
        int n = atoi(argv[1]); //tamanho da matriz
        int t = atoi(argv[2]); //numero de threads
        int aux=(n/2)*n;
        double U[n][n], C[aux];
        int i, j, k, cont =0;
        int l=0;
        double c,  soma=0.0;

        omp_set_num_threads(t);

        for(i=0; i<n;i++) //criando a matriz
        {
                for(j=0;j<n;j++)
                {
                        U[i][j]=rand()%100;
                }
        }
        printf("\nMatriz A\n");
        for(i=0;i<n;i++)
```

```
{
        for(j=0;j<n;j++)
        {
                printf("%f␣␣", U[i][j]);
        }
        printf("\n");
}
double start, end;
start = omp_get_wtime();
for(j=0; j<n; j++)
{
        for(i=0; i<n; i++)
        {
                if(i>j)
                {
                        c=-(U[i][j]/U[j][j]);
                        #pragma omp parallel
                        {
                                #pragma omp for
                                for(k=0; k<n; k++)
                                {
                                        U[i][k]=c*U[j][k]+U[i][k];
                                }
                        }
                        C[l]=-c;
                        l++;
                }
        }
}

printf("␣\nMatriz␣U\n␣");
for(i=0;i<n;i++)
{
        for(j=0;j<n;j++)
        {
                printf("%0.1f␣␣", U[i][j]);
        }
        printf("\n␣");
}
printf("␣\n␣Vetor␣C\n␣");
for(j=0;j<aux;j++)
{
        printf("%0.1f␣␣", C[j]);
}
#pragma omp parallel for
        for(i=0; i<=n;i++)//matriz L
        {
                for(j=0;j<n;j++)
                {
                        if(i==j)
                        {
                                U[i][j]=1.0;
                        }
                        else
                        {
                                if(j>i)
                                        U[i][j]=0.0;
```

```
                        }
                    }
                }
        k=aux/t;
        int f;
        #pragma omp parallel private(f, j)//valores de f e j privados
        {
                int id=omp_get_thread_num();
                f=id*k; //off set do vetor de pivos
                #pragma omp for
                for(i=1;i<n;i++)
                {
                        for(j=0;j<n;j++)
                        {
                                if(i>j)
                                {
                                        U[i][j]=C[f];
                                        f++;
                                }
                        }
                }
        }
        end = omp_get_wtime();
        double tempo = end-start;
        printf("\n Matriz L \n ");
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%0.1f  ", U[i][j]);
                }
                printf("\n ");
        }
        printf("\ntempo de execu  o: %f\n", tempo);
}
```

2.3 Resultados

Os testes foram efetuados de forma que: com três diferentes entradas (200, 400, 800), correlacionando com (1, 2, 4) processadores ou threas. Utilizou-se também de 8 threads, mas na máquina em teste, os valores não se considerou real, e sim virtual pela multithreading. O valor real dos núcleos é 4. Ao executar o código por várias vezes, obtemos uma média de valores que consta nas seguintes tabelas abaixo:

Número de Entrada	1 Thread	2 Thread	4 Thread
200	0.06	0.06	0.06
400	0.34	0.33	0.31
800	1.78	1.59	1.46

Table 1: Tempos relacionando número de entradas vs número de Threads

Pela tabela e pelo gráfico dos tempos, verificamos que o tempo de execução teve uma decaída conforme foi aumentando as entradas junto com o aumento dos processadores.

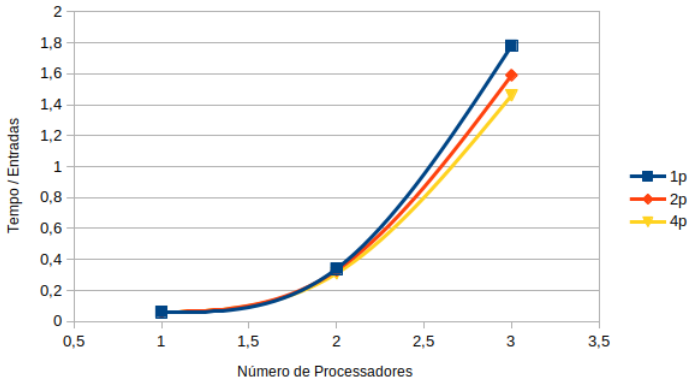

Figure 4: Relacição das entradas e processadores com o tenpo

Número de Entrada	1 Thread	2 Thread	4 Thread
200	1	1	1
400	1	1,03	1,09
800	1	1,11	1,22

Table 2: Valores dos Speedup relacionando número de entradas vs número de Threads

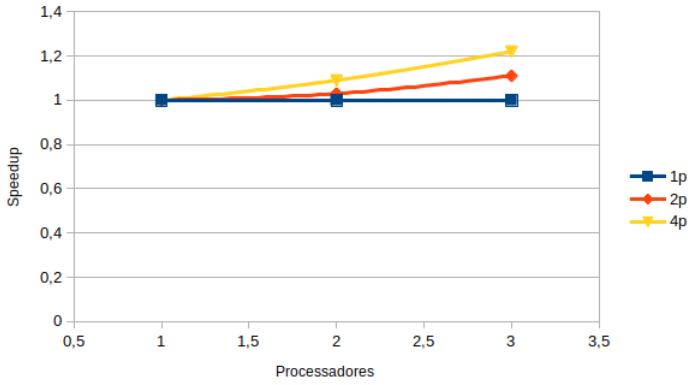

Figure 5: Speedup relacionando tempo sequencial vs tempo paralelo

Com os valores dos Speedups e seu gráfico, observamos que o Speedup escalou a medida que foi aumentado o número de processadores ou threads.

3 Conclusão

O programa OpenMP parece ser otimizado mesmo para grandes matrizes, demostrou bom desempenho escalando ate 4 processadores, melhor que o programa sequencial. Isto se deve, pelo motivo que os índices das colunas podem ser resolvidos independentemente. Então, quando usamos a diretiva pragma para o looping de coluna, para que cada execução da coluna seja feita em paralelo. Podemos notar no gráfigo um bom Speedup com 4 processadores.