UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

Paralelização e Análise de Escalabilidade na Decomposição LU

Discentes: Flávio Henrique Lopes Barbosa**,**

Jaysa Keylla Siqueira Barbosa

José Augusto Agripino de Oliveira.

Introdução

A aplicação de Decomposição LU, mesmo sendo um algoritmo pequeno, em relação a sua lógica e linhas de código, requer um alto poder de processamento quando aplicada a matrizes com grandes ordens e quantidade de elementos. Esta prática tem como objetivo a aplicação da paralelização do código utilizado relativo ao respectivo algoritmo, bem como, através de recursos já trabalhados, como o API OpenMp e o compilador GCC, ter-se a capacidade de analisar a escalabilidade do programa, e por conseguinte seu desempenho

Metodologia

Para fazer a paralelização foi utilizada a ferramenta OpenMp, adicionando a biblioteca omp.h. Já para realizar a análise de escalabilidade, foi modificado o tamanho do problema que o algoritmo precisa resolver. Na nossa aplicação de decomposição LU, foi feita alterando o tamanho da matriz. Para análise, foram utilizadas as ordens 4000, 6000, 8000, 12000 e 16000 para a matriz a ser decomposta. Ademais, para cada ordem, foram utilizados quatro valores de threads diferentes: 1, 4, 8 e 16 threads, a fim de observar o desempenho obtido.

Utilizamos as seguintes linhas de comando para compilação, ativando os recursos avançados do compilador GCC e utilizando o API OpenMp

1. *$ gcc -pg -wall omp_decomposicaoLU.c -0 omp_decomposicaoLU.o -fopenmp*
2. *$ ./decomposicaoLU (parâmetro **nº Threads**)*
3. *$ gprof ./decomposicaoLU.out gmon.out > omp_ordem_n_T_x.txt*

Os parâmetros fornecidos foram o número de threads, por meio da linha de comando, e a ordem da matriz, sendo mudado diretamente no código em uma variável global. Esse segundo parâmetro está diretamente ligado à complexidade da aplicação, pois faz com que aumente o número de iterações nos laços **for.**

Os códigos implementados com OpenMP estão a seguir:

```
20  void preencheMatriz(float *A, int thread_count) {
21  #   pragma omp parallel num_threads(thread_count) default(none) \
22          shared(A)
23      for(int i=0; i<n; i++) {
24          for(int j=0; j<n; j++) {
25              A[(i*n)+j] = rand()%100000;
26          }
27      }
28  }
29
30  void calculaMatrizes(int i, int j, float *aux, float *M, int thread_count) {
31      M[(i*n)+j] = aux[(i*n)+j]/aux[(j*n)+i];
32
33      //multiplica a linha do multiplicar por -(multiplicador)
34      //Movimenta a coluna
35  #   pragma omp for
36      for(int c=j; c<n; c++){
37          aux[(i*n)+c] = aux[(i*n)+c]+aux[(j*n)+c]*(-1*(M[(i*n)+j]));
38      }
39  }
```

```
41  void gauss(float *A, float *M, float *aux, int thread_count){
42  #   pragma omp parallel num_threads(thread_count)
43      {
44      #   pragma omp for
45          for(int i=0; i<n; i++) {
46              for(int j=0; j<n; j++) {
47                  M[(i*n)+j] = 0;
48                  aux[(i*n)+j] = A[(i*n)+j];
49              }
50          }
51          //Calculando os multiplicadores da Gauss
52          //Movimenta a coluna
53          for(int j=0; j<(n-1); j++){
54          //Movimenta a linha
55              for(int i=j+1; i<n; i++){
56                  ( aux[(i*n)+j] != 0 ) ? calculaMatrizes(i, j, aux, M, thread_count) : NULL;
57          }//Todos os multiplicadores da coluna j estão definidos
58          }
59      #   pragma omp for
60          for(int i=0; i<n; i++) M[(i*n)+i]=1;
61      }
```

Foram utilizadas as diretivas **#pragma omp paralell**, modificadas de acordo com a análise a qual se queria direcionadas pelas cláusulas, nesse caso específico utilizamos **num_threads(treads_count)** para paralelizar explícitamente fornecendo o número de threads como parâmetro de entrada para execução. Essa entrada foi fornecida por meio da linha de comando na hora de executar. Foi explicitada também a matriz A como variável compartilhada através da cláusula **shared(A)**.

Além disso, foi utilizada a principal causa para que o OpenMP foi criado, que é a paralelização de laços de repetição, que, em alguns códigos, tornam-se os gargalos do algoritmo. Para isso, foi utilizada, também, a diretiva **#pragma omp for**.

Resultados

Análise de Escalabilidade

Entrada / Threads	1	4	8	16
4000	77.11s	75.79s	75.13s	119.18s
6000	278.57s	280.31s	274.93s	401.34s
8000	677.59s	576.62s	627.14s	1045.73s
12000	2061.47s	2093.02s	2144.85s	3093.00s
16000	4955.13s	4943.75s	4937.96s	5715.28s

Executando o código com as diferentes entradas e número de threads, esses foram os resultados obtidos em tempo de execução. Foram um pouco menores que o tempo de execução sequencial, mas deveriam ter sido bem mais baixos, uma vez que o trabalho está sendo dividido entre as várias threads.

Análise de Speed Up (Tempo Sequencial / Tempo Paralelo)

Entrada / Threads	1	4	8	16
4000	1	1.02	1.03	0.65
6000	1	0.99	1.01	0.70
8000	1	1.18	1.08	0.65
12000	1	0.98	0.96	0.67
16000	1	1.01	1.01	0.87

Os resultados obtidos não foram como esperados, uma vez que todos eles deveriam ter sido maiores que 1, para que representasse uma melhoria em relação ao tempo de execução sequencial, o que não ocorreu.

Análise de Eficiência (SpeedUp / nºde Threads)

Entrada / Threads	1	4	8	16
4000	1	0.26	0.13	0.04
6000	1	0.25	0.01	0.04
8000	1	0.26	0.14	0.04
12000	1	0.25	0.12	0.04
16000	1	0.25	0.13	0.05

A eficiência, como os outros resultados, não foi nada perto do esperado. Isso ocorre devido ao tempo de execução ter diminuído pouco conforme o número de threads era aumentado, gerando, desse modo, uma baixa eficiência.

Conclusão

Os resultados obtidos com a paralelização não foram satisfatórios, pois as variáveis de análise não demonstraram uma tendência de melhora na eficiência. Diante de tais dados, percebe-se que a aplicação da técnica de paralelização deve ser reconsiderada, em relação aos trechos onde foi aplicada e elementos, os quais se queria manipular para chegar ao objetivo proposto. Tendo que se fazer uma nova análise sobre as causas que levaram ao não ganho de desempenho, speedup e eficiência.