Trabalho Prático OpenMP

Heloisa Benedet Mendes
Bacharelado em Ciência da Computação
Universidade Federal do Paraná – UFPR
Curitiba, Brasil
heloisamendes@ufpr.br

Conteúdo

I. Introdução

O *Problema da Maior Subsequência Comum* (MSC) ou o *Longest Common Subsequence* (LCS) *Problem* é um problema clássico em algoritmos de computadores. Uma subsequência é uma série de símbolos contidos em uma sequência original, preservando a ordem, mas não necessariamente a contiguidade. Dada uma sequência de símbolos $S =< s_0, s_1, ..., s_n >$, uma subsequência S' de S é formada exclusivamente por elementos de S, mantendo sua ordem relativa.

Determinar a maior subsequência comum entre duas sequências é um desafio computacional relevante, com aplicações em áreas como bioinformática, comparação de arquivos e análise de textos.

Este trabalho tem como objetivo paralelizar o algoritmo de cálculo da LCS, originalmente implementado de forma sequencial. O relatório descreve o funcionamento das versões sequencial e paralela, além de apresentar os experimentos realizados e os resultados obtidos.

II. Programa Principal

O programa recebe dois arquivos de entrada, A e B, cada um contendo uma sequência de caracteres. O tamanho da sequência do arquivo A é denotado por N_A, e no arquivo B, por M_B. A partir dessas sequências, é criada uma matriz de dimensões $M_B + 1 \times N_A + 1$, representada abaixo:

$$AB = \begin{bmatrix} \times & 0 & a_0 & a_1 & ... & a_n \\ 0 & 0 & 0 & 0 & ... & 0 \\ b_0 & 0 & 0 & 0 & ... & 0 \\ b_1 & 0 & 0 & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ b_m & 0 & 0 & 0 & ... & 0 \end{bmatrix} \tag{1}$$

Essa matriz será preenchida com as pontuações que indicam o comprimento da maior subsequência comum até cada posição (i, j). O valor de cada célula é calculado da seguinte maneira:

$$p[i][j] = \begin{cases} 0, \ se \ i = 0 \ ou \ j = 0 \\ p[i-1][j-1], \ se \ i, j > 0 \ e \ a_j = b_i \\ max(p[i-1][j], p[i][j-1]), \ caso \ contrario \end{cases} \tag{2}$$

Nas subseções a seguir, será discutido como cada abordagem (sequencial e paralela) realiza esse cálculo.

A. Abordagem Sequencial

O programa sequencial de cálculo da LCS segue o pseudocódigo apresentado na Figura 1.

A execução percorre a matriz AB linha por linha, onde b_i representa as linhas e a_j as colunas.

Se b_i for igual a a_j, o valor da célula atual é definido como o valor da célula diagonal superior esquerda somado a 1. Caso contrário, o valor é o máximo entre os valores das células à esquerda e acima da posição atual, assim como descrito na Equação (2).

```
Para i = 1 até M_B + 1
    Para j = 1 até N_A + 1
        se A[j-1] == B[i-1]
            AB[i][j] = AB[i-1][j-1] + 1
        senão
            AB[i][j] = max(AB[i-1][j], AB[i][j-1])
```

Figura 1. Pseudocódigo abordagem sequencial

B. Abordagem Paralela

Analisando o código sequencial, observa-se que o cálculo da pontuação em cada posição da matriz depende de três células: a à esquerda, a acima e a na diagonal superior à esquerda. Essa dependência de dados impõe restrições na ordem de cálculo e precisa ser considerada na paralelização do algoritmo.

Para contornar essa limitação, foi adotada a técnica de paralelização conhecida como *wavefront*. Com essa abordagem, as células localizadas na mesma diagonal secundária (de cima para baixo e da esquerda para a direita) podem ser processadas simultaneamente, pois todas dependem apenas de células pertencentes a diagonais anteriores. Isso garante o respeito às dependências de dados e permite a execução paralela eficiente.

Para a abordagem paralela, é necessário definir o número de *threads* que serão utilizadas na execução. Esse valor é fornecido pelo usuário na entrada do programa e será utilizado para configurar a paralelização com OpenMP. O pseudocódigo apresentado na Figura 2 ilustra a lógica implementada para distribuir a carga de trabalho entre as *threads*.

```
#pragma omp parallel
{
    Para d = 2 até N_A + M_B
        #pragma omp for
        Para i = 1 até M_B +1
            se A[j-1] == B[i-1]
                AB[i][j] = AB[i-1][j-1] + 1
            senão
                AB[i][j] = max(AB[i-1][j], AB[i][j-1])
}
```

Figura 2. Pseudocódigo abordagem paralela

O algoritmo percorre as diagonais secundárias da matriz (também chamadas de diagonais anti-principais), que vão da parte superior esquerda para a inferior direita. Para cada valor de d, correspondente ao índice da diagonal atual, os elementos dessa diagonal são processados em paralelo. A diretiva $pragma\ omp\ parallel$ inicializa o ambiente paralelo, e $pragma\ omp\ for$ distribui a iteração da diagonal entre as *threads* disponíveis.

Cada célula $AB[i][j]$ da matriz é preenchida de acordo com a mesma regra do algoritmo sequencial.

III. Metodologia

Para a realização dos experimentos de forma sistemática e segura, foi utilizado o script $script.py$, que automatiza a execução do programa com diferentes configurações de número de *threads* e tamanhos de entrada N. Para simplificação, o programa considera $N_A = M_B = N$, ou seja, ambas as sequências possuem o mesmo comprimento.

- *Threads*: 1, 2, 4 e 8
- N: 1000, 5000, 10000, 20000, 30000, 40000 e 50000

Limitações de hardware impediram a realização de testes com valores de N superiores a 50000. Para garantir a confiabilidade dos resultados, cada combinação de número de *threads* e tamanho de entrada foi executada 20 vezes, permitindo calcular médias e reduzir o impacto de variações pontuais.

Os experimentos foram conduzidos em uma máquina com as seguintes especificações:

- **Sistema Operacional**: Ubuntu 24.04.2 LTS x86_64
- **Kernel**: 6.11.0-25-generic
- **Compilador**: gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0
- **Processador**: 11th Gen Intel i5-1135G7 (8) @ 4.200GHz
- **Núcleos**: 4
- ***Threads* por Núcleo**: 2

As *flags* de compilação utilizadas foram $-O3$, para otimizações de desempenho, e $-fopenmp$, para habilitação do suporte à paralelização com OpenMP.

IV. Resultados

A. Dados

A seguir, é apresentado um resumo dos resultados obtidos nos experimentos descritos anteriormente. Os valores correspondem à média de tempo de execução do código paralelo em 20 execuções realizadas para cada configuração experimental.

N	1 Thread	2 Threads	4 Threads	8 Threads
1000	0.0027	0.0030	0.0037	0.0478
5000	0.1023	0.1163	0.0948	0.1683
10000	0.5286	0.6934	0.6602	0.9569
20000	2.8557	3.8863	4.0261	5.6945
30000	10.3379	13.9905	12.1458	18.5878
40000	35.0962	41.7084	50.0934	41.5524
50000	68.3891	94.1735	124.8553	81.9682

Tabela I
Dados Obtidos

B. Análise de Dados

A Figura 3 mostra a relação entre o tempo de execução do código paralelo e o número de *threads* para diferentes tamanhos de N. Observa-se que o tempo de execução não diminui com o aumento do número de *threads*. Na maioria dos casos, o tempo aumenta à medida que mais *threads* são utilizadas.

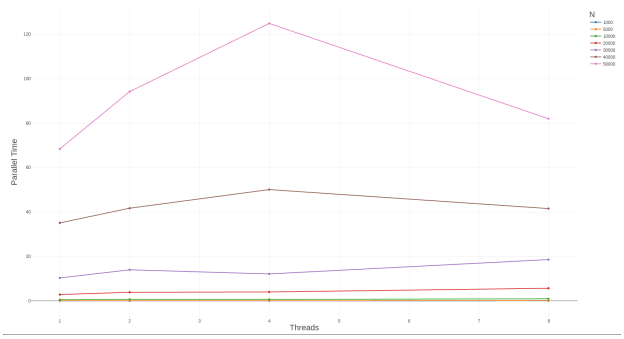

Figura 3. Tempo Paralelo x Threads

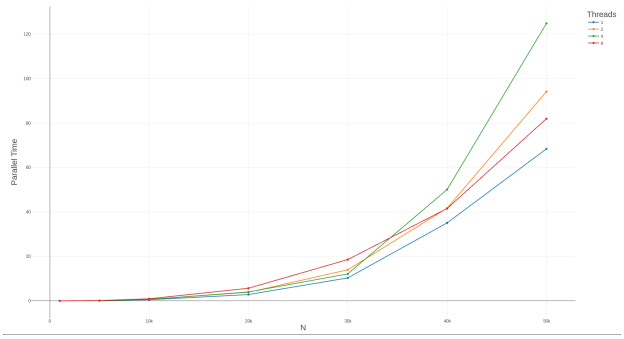

Figura 4. Tempo Paralelo x N

O tempo de execução atinge seu pico com 4 *threads* e apresenta uma leve melhora com 8 *threads*, mas ainda assim permanece superior ao tempo registrado com apenas 1 *thread*.

A Figura 4 apresenta os mesmos dados sob outra perspectiva, relacionando o tempo de execução paralelo ao tamanho de N. O tempo cresce significativamente com o aumento de N, no entanto, apresenta um comportamento incomum: para $N > 30000$, o uso de mais *threads* não implica em menor tempo de execução. Em alguns casos, a execução com 4 *threads* é mais lenta do que com 2 ou até mesmo com 1 *thread*. Esse padrão sugere que o custo da paralelização não escala bem para problemas maiores.

N	1 Thread	2 Threads	4 Threads	8 Threads
1000	0.6810077095	0.5349966701	0.4434104763	0.03451343903
5000	0.4420804141	0.3862005325	0.4780017607	0.2679942997
10000	0.3590619369	0.2745635814	0.2887112462	0.1987347587
20000	0.2736815617	0.20113878	0.1950265635	0.1377964413
30000	0.171126969	0.1269894023	0.1462863943	0.09602807469
40000	0.1053502082	0.07612865192	0.0632402543	0.07626667781
50000	0.07242816003	0.05256088716	0.03964387712	0.06042237953

Tabela II
Tabela de *SpeedUp*

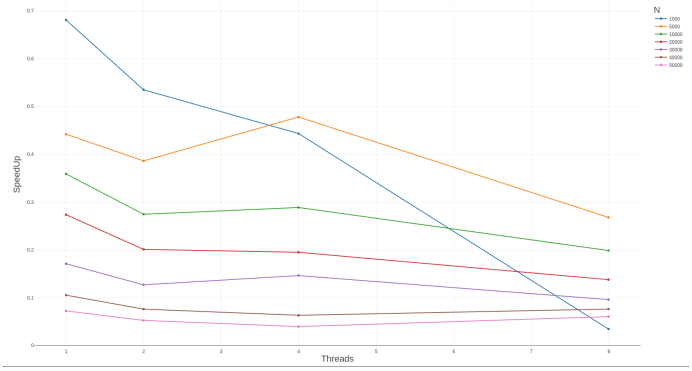

Figura 5. SpeedUp x Threads

A Tabela II apresenta os valores de *SpeedUp* obtidos de acordo com a Equação (3) e o número de *threads*, e a Figura 5 apresenta a relação entre o *SpeedUp* e o número de *threads*.

$$SpeedUp = \frac{Tempo\ Sequencial}{Tempo\ Paralelo} \tag{3}$$

O gráfico mostra que o *SpeedUp* diminui conforme o aumento do número de *threads*, especialmente para valores de N pequenos.

Para $N = 1000$ o desempenho piora drasticamente com mais *threads*, mesmo para valores maiores de N o ganho é pequeno e tende a se manter ou cair com mais de 4 *threads*.

A Lei de Amdahl, expressa na Equação (4), descreve a limitação do aumento de velocidade de um sistema quando apenas uma parte dele é paralelizada, e pode ser utilizada para calcular a eficiência da paralelização.

$$S = \frac{1}{(1-P) + \frac{P}{Threads}} \tag{4}$$

Na qual

- S: *SpeedUp* teórico máximo
- P: fração do código paralelizável obtida através de $\frac{Tempo\ Sequencial}{Tempo\ Total - Tempo\ Paralelo}$
- $Threads$: número de *threads* utilizadas.

O *SpeedUp* teórico máximo obtido segundo a Lei de Amdahl está representado na Tabela III a seguir.

N	1 Thread	2 Threads	4 Threads	8 Threads
1000	4.438482968	3.601457186	2.80429258	2.241967964
5000	8.406740086	8.599504366	8.843597961	6.635945335
10000	17.24683791	16.20409023	15.16384918	12.93108887
20000	32.22341297	29.70378713	25.75187149	20.20872872
30000	46.21397175	43.48374821	37.06349851	28.92214811
40000	56.20822742	57.41130794	50.46552075	35.80454454
50000	76.73715431	71.6216656	62.11899114	39.51476052

Tabela III
Tabela de Eficiência

V. Conclusão

A partir dos resultados apresentados nas Figuras 3, 4 e 5, bem como nas Tabelas I, II e III, observa-se que a paralelização do algoritmo de LCS com OpenMP não apresentou os ganhos esperados em termos de desempenho. Como mostrado na Tabela I e na Figura 3, o tempo de execução do código paralelo não diminui com o aumento do número de *threads*; pelo contrário, em muitos casos, ele aumenta, atingindo picos de lentidão com 4 *threads*. A Figura 4 reforça esse comportamento, indicando que, mesmo com o crescimento do tamanho da entrada N, o aumento do número de *threads* não necessariamente resulta em maior eficiência.

Esse comportamento é refletido na Tabela II e na Figura 5, onde o *SpeedUp* obtido é baixo e decrescente à medida que mais *threads* são utilizadas, especialmente para valores pequenos de N. A Tabela III, por sua vez, apresenta o *SpeedUp* teórico máximo segundo a Lei de Amdahl, evidenciando uma diferença considerável em relação aos valores práticos, o que sugere que o potencial de paralelismo do algoritmo não foi plenamente explorado.

Diante desses resultados, conclui-se que o algoritmo apresenta escalabilidade fraca, uma vez que o aumento no número de *threads* não resulta em ganhos proporcionais de desempenho — mesmo para entradas maiores. O *overhead* associado à paralelização e a divisão de carga ineficiente limitam os benefícios da execução paralela. Isso evidencia a necessidade de estratégias mais eficientes de paralelização e balanceamento de carga para que os recursos computacionais possam ser aproveitados de forma mais eficaz.