Paralelização do Algoritmo LCS usando OpenMP

João Pedro V. Ramalho (GRR20224169)
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil
jpvr22@inf.ufpr.br

I Introdução

O problema da Maior Subsequência Comum (LCS) é aplicado a bioinformática e processamento de textos. Sua solução sequencial por programação dinâmica possui complexidade $O(n^2)$, tornando-se proibitiva para grandes entradas. Este trabalho explora a paralelização do algoritmo usando OpenMP, avaliando diferentes estratégias e métricas de desempenho.

II Kernel

O núcleo do algoritmo busca pela maior subsequência comum entre as sequências $A = (a_0, a_1, \ldots, a_n)$ e $B = (b_0, b_1, \ldots, b_m)$ de tamanhos *n* e *m*, respectivamente, seguindo:

$$c[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1][j-1] + 1 & \text{se } i, j > 0 \text{ e } a_i = b_j \\ \max(c[i-1][j], c[i][j-1]) & \text{caso contrário} \end{cases} \quad (1)$$

Com isso, a primeira linha e coluna da matriz *c* são zeradas para tratar os casos em que as sequências são vazias. Durante o preenchimento da matriz, se os caracteres na coluna *i* (a_i) e linha *j* (b_j) forem iguais, o valor da célula *c[i][j]* recebe o valor da diagonal superior esquerda (*c[i-1][j-1]*) + 1. Isso significa que encontramos mais um elemento em comum entre as sequências (*match*).

Caso os caracteres sejam diferentes, propagamos o maior valor entre a célula superior (*c[i-1][j]*) e a célula à esquerda (*c[i][j-1]*), garantindo que cada posição armazene o maior tamanho de subsequência local.

Ao final, a matriz *c* nos permitirá determinar a maior subsequência comum entre as sequências A e B, bem como o seu tamanho, que é armazenado na última célula (*c[m][n]*). Esse resultado pode ser visualizado no exemplo da Figura 1, que ilustra o caso das sequências A = ATCGTAC e B = ATGTTAT

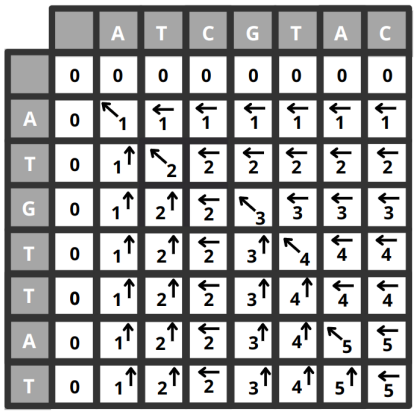


Figura 1: Matriz preenchida para A = **ATCGTAC** e B = **ATGTTAT**.

Na Figura 1, podemos observar que a matriz apresenta caminhos de sentido horizontal, vertical e diagonal:

- **Caminhos diagonais:** Representam *matches* entre caracteres. A diagonal incrementa o contador de subsequência comum. Exemplo: O caminho de $c[1][1]$ para $c[2][2]$ indica o *match* entre a_2 e b_2.
- **Caminhos horizontais:** Indicam avanço na sequência A sem correspondência em B. Exemplo: Movimento de $c[2][2]$ para $c[2][3]$, onde não há *match* entre a_3 e b_2.
- **Caminhos verticais:** Indicam avanço na sequência B sem correspondência em A. Exemplo: Movimento de $c[3][2]$ para $c[4][2]$, sem *match* entre a_1 e b_2.

Esses caminhos nos permitem reconstruir a maior subsequência comum. Para isso, basta seguir o caminho inverso a partir da célula (*c[m][n]*), seguindo a direção das flechas e adicionando à subsequência apenas os caminhos diagonais, como ilustrado na Figura 2.

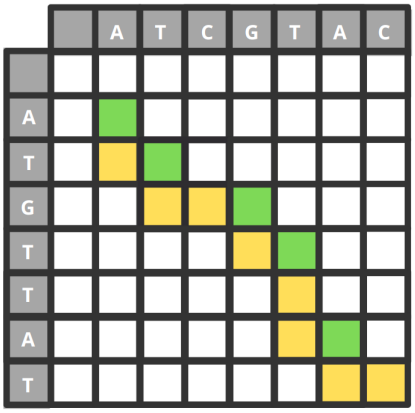


Figura 2: Caminho para A = **ATCGTAC** e B = **ATGTTAT**.

Dessa forma, o algoritmo revelou que a maior subsequência comum entre as sequências A e B é: $A - T - G - T - A$, de tamanho 5.

III Estratégia de Paralelização

A implementação paralela do algoritmo LCS adotou o padrão *wavefront*, explorando o paralelismo nas diagonais da matriz de pontuação. O código abaixo ilustra a abordagem:

```
#pragma omp parallel num_threads(8)
{
    for (int diag = 2; diag <= size_A + size_B; diag++) {
        int start = (diag > size_A) ? (diag - size_A) : 1;
        int end = (diag > size_B) ? size_B : (diag - 1);
        #pragma omp for schedule(guided, 128)
        for (int i = start; i <= end; i++) {
            int j = diag - i;
            if (seq_A[j - 1] == seq_B[i - 1]) {
                score_matrix[i][j] = score_matrix[i - 1][j - 1] + 1;
            } else {
                score_matrix[i][j] = max(score_matrix[i - 1][j], score_matrix[i][j - 1]);
            }
        }
    }
}
```

A. Configuração Ótima e Fundamentação

A seleção de 8 threads com escalonamento `guided` e `chunk=128` foi validada experimentalmente (Seção IV), apresentando desempenho superior em matrizes quadradas ($N \times N$) e retangulares ($N \times M$). Esta combinação equilibra três fatores:

- **Balanceamento de carga dinâmico**: O escalonamento `guided` aloca inicialmente blocos grandes às threads e reduz progressivamente o tamanho dos `chunks`, adaptando-se à variação de carga entre diagonais.
- **Redução de overhead**: O `chunk=128` minimiza a frequência de sincronização entre threads, reduzindo o custo de alocação de tasks;
- **Otimização de cache**: Blocos maiores melhoram a localidade espacial, reduzindo *cache misses*.

A barreira implícita após cada `#pragma omp for` garante a dependência de dados entre diagonais, preservando a correção do algoritmo.

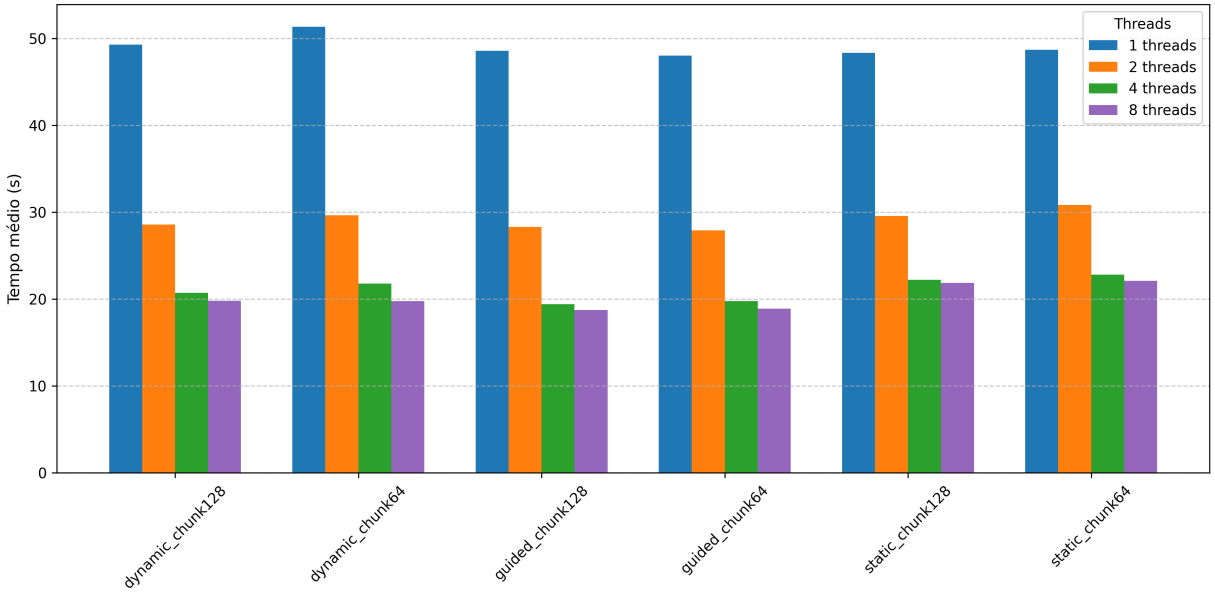
B. Análise Comparativa de Configurações

O escalonamento `static` demonstrou limitações significativas no balanceamento de carga, especialmente em matrizes assimétricas. Por dividir as iterações em `chunks` fixos antes da execução, threads frequentemente recebiam partes desproporcionais do trabalho. Por exemplo, em matrizes retangulares com 8 threads, o `static` obteve tempos 17% superiores ao

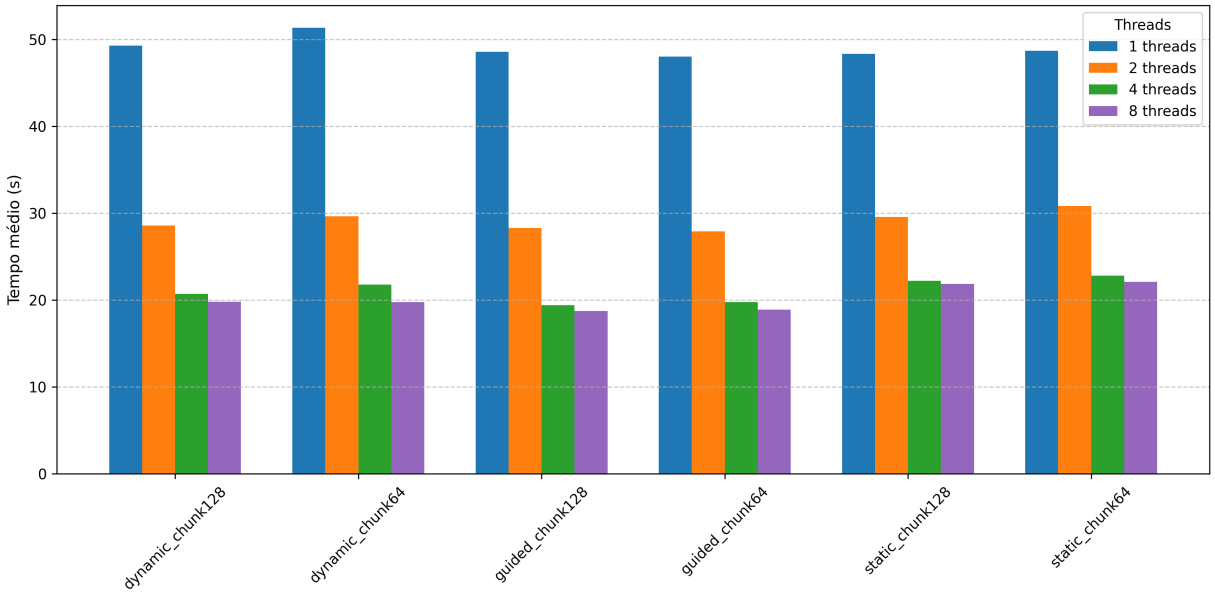
`guided` (4,59s vs 3,91s), já que não se adapta à variação de complexidade entre diagonais. Essa rigidez levou a ociosidade em threads com `chunks` menores, enquanto outras processavam regiões mais densas.

Por outro lado, o escalonamento `dynamic` melhorou o equilíbrio de carga, mas introduziu overhead de sincronização. Com `chunks` pequenos (ex: 64), as threads requisitavam novas tarefas com frequência. Além disso, `chunks` reduzidos fragmentaram o acesso à memória, aumentando cache misses.

A configuração `guided` superou essas limitações. Para matrizes quadradas, o `guided 128` foi 0,8% mais rápido que o `guided 64` (18,74s vs 18,90s), pois `chunks` maiores reduziram o custo de alocação inicial e otimizaram a localidade espacial na cache. Já em matrizes retangulares, o `guided 64` obteve vantagem de 0,7% (3,91s vs 3,94s), ajustando-se dinamicamente a diagonais irregulares. Ao final, escolheu-se adotar o `guided 128` como estratégia vitoriosa porque foi a configuração que lidou melhor com as entradas maiores - veja Figura 3a e 3b.



(a) Entradas: matrizes quadradas (NxN)



(b) Entrada: matrizes retângulares (NxM).

Figura 3: Comparação entre configurações de escalonamento, chunk e threads.

IV Metodologia Experimental

Nesta seção, descrevemos o ambiente e a metodologia adotados para a execução dos experimentos de desempenho.

A. Ambiente de Execução

Os experimentos foram realizados em uma máquina com as seguintes configurações:

- **Sistema Operacional:** Linux Mint 21.2 (Victoria)
- **Kernel:** 5.15.0-117-generic
- **Processador:** Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
- **Número de núcleos físicos:** 4 (com 2 threads por núcleo, totalizando 8 threads)
- **Compilador:** GCC 11.4.0

A compilação do programa foi realizada com as seguintes flags:

- `-Wall` para avisos de compilação
- `-O3` para otimizações de desempenho
- `-fopenmp` para habilitar a paralelização com OpenMP

B. Procedimento Experimental

Para avaliar o desempenho da implementação paralela, foram realizados testes com diferentes tamanhos de entrada e diferentes configurações de paralelização. Os scripts de teste automatizam a variação dos seguintes parâmetros:

- Tamanho das sequências de entrada: variando de 1.000 até 100.000 elementos, a passos diferentes conforme o experimento.
- Número de threads: `1`, `2`, `4` e `8`.
- Estratégia de escalonamento: `static`, `dynamic` e `guided`.
- Tamanho de chunk: `64` e `128`.

Duas abordagens de experimentação foram utilizadas:

1) **Sequências de mesmo tamanho:** ambas as entradas possuem o mesmo tamanho, variando de 1.000 a 100.000.
2) **Sequências de tamanhos distintos:** todos os pares diferentes (A, B) foram testados, onde $A \neq B$ e $A, B \in \{1.000, 2.500, 5.000, 7.500, 10.000, 25.000, 50.000, 75.000, 100.000\}$.

Para cada combinação de parâmetros, o teste foi executado **21 vezes consecutivas**, registrando os tempos em arquivos separados. Além disso, os arquivos de entrada foram gerados previamente usando um utilitário (`generate_input`) para criar sequências aleatórias, garantindo reprodutibilidade dos testes.

Todos os arquivos utilizados nos testes, bem como os resultados obtidos, estão disponíveis no GitHub e podem ser acessados no final deste relatório para referência.

Por fim, para as tabelas de medição de tempo, *speedup* e eficiência foi utilizado a estratégia vitoriosa para casos gerais criados aleatoriamente pelo utilitário `generate_input` de matrizes NxN, com N variando de 50.000 a 100.000.

V Medições de Tempo

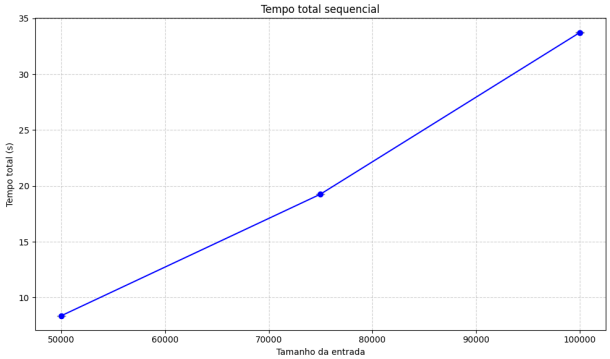
A porcentagem de tempo que o algoritmo passa na seção paralela foi de 99,9829%. Na seção sequencial, há apenas a leitura das entradas (sequências dos arquivos), a alocação e inicialização da matriz de pontuação — operações relacionadas à manipulação de memória e preparação dos dados. Por isso, essa região representa um tempo muito pequeno em todos os casos, cerca de 0,0171% do tempo total da execução paralela. A comparação entre o tempo de execuções das versões puramente sequenciais e paralelas é representada pelas Figuras 4a, 4b e 4c.

VI Lei de Amdahl

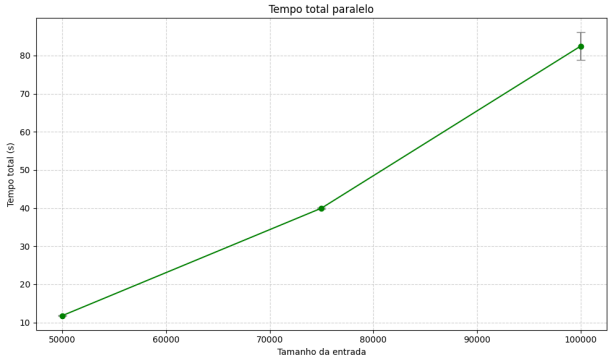
Tabela I: Speedup teórico máximo pela Lei de Amdahl

Núcleos	β **(parte sequencial)**	**Speedup Teórico**
2	0,000041	1,999918
4	0,000059	3,999830
8	0,000058	7,999660
∞	0,000052	24 303,009860

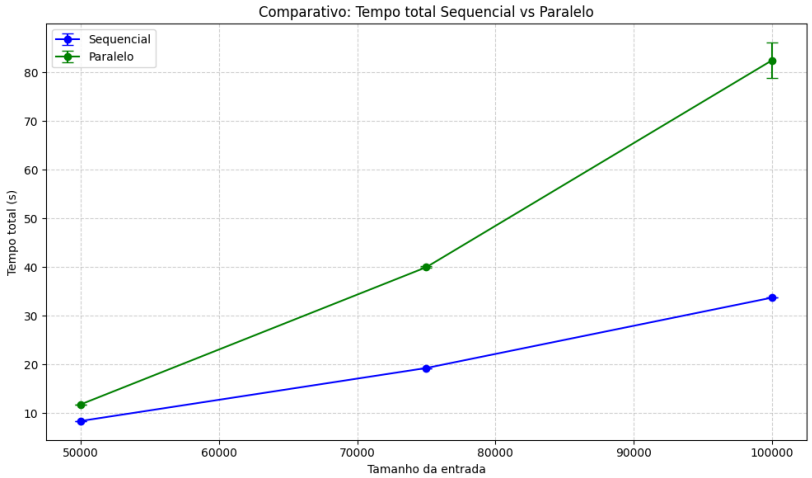
Para analisar o desempenho da implementação paralela do algoritmo LCS, foi necessário distinguir entre o tempo puramente sequencial e o tempo paralelizável. O tempo sequencial engloba operações fundamentais que não podem ser paralelizadas, como a leitura dos arquivos de entrada, a alocação e inicialização da matriz de scores, além da exibição dos resultados finais. Por outro lado, o tempo paralelizável concentra-se no núcleo do cálculo da matriz LCS, que foi otimizado. A medição precisa dos tempos foi realizada através da função `omp_get_wtime()`. Como o preenchimento da matriz representa a parcela mais significativa do tempo de execução e foi adequadamente paralelizado, os resultados demonstram uma aceleração considerável quando comparado à versão sequencial.



(a) Tempo total da versão sequencial



(b) Tempo total da versão paralela



(c) Comparativo entre sequencial e paralela

Figura 4: Comparações de tempo entre versões sequencial e paralela

VII Speedup e Eficiência

Nesta seção, apresentamos os resultados de desempenho em termos de *speedup* e eficiência relativa ao número de threads utilizadas. As Tabelas II e III mostram os valores obtidos para diferentes tamanhos de entrada.

Tabela II: Speedup obtido para diferentes números de threads

Tamanho da Entrada	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**
50 000	1,00	1,84	2,91	2,89
75 000	1,00	0,34	0,48	0,48
100 000	1,00	1,70	2,44	2,55

Tabela III: Eficiência (%) para diferentes números de threads

Tamanho da Entrada	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**
50 000	100,00	91,84	72,81	36,12
75 000	100,00	16,85	12,08	6,03
100 000	100,00	84,87	61,04	31,85

VIII Análise dos Resultados

Observa-se, inicialmente, que para o tamanho de entrada $N = 50\,000$ o speedup cresce de forma quase linear até 4 threads (atingindo $S = 2{,}91$), mas quase estaciona ao passar para 8 threads ($S = 2{,}89$), refletindo uma queda significativa na eficiência,

que cai de 72,81 % para apenas 36,12 %. Para $N = 100\,000$, o ganho é mais consistente: o speedup alcança 1,70 com 2 threads, 2,44 com 4 threads e 2,55 com 8 threads, resultando em eficiências de 84,87 %, 61,04 % e 31,85 %, respectivamente. Isso indica que, embora o aumento do tamanho da entrada amortize melhor o overhead de paralelização, a escalabilidade continua limitada além de 4 threads.

O caso de $N = 75\,000$ apresenta comportamento atípico, com speedup inferior a 1 para 2, 4 e 8 threads, sugerindo que o overhead de sincronização e de criação de tarefas não foi compensado pelo trabalho paralelo. Isso pode ter ocorrido por variabilidade na medição de tempos ou interferência de processos de sistema, visto que estes testes foram realizados em um momento em que o autor não podia disponibilizar a máquina apenas para executar os testes. Para confirmar o resultado, seria necessário repetir esses experimentos sob condições controladas e de uso exclusivo da máquina.

Em comparação com o speedup linear ideal ($S = p$), todos os pontos observados ficam aquém do ideal, especialmente quando p cresce. Da mesma forma, ao aplicar a Lei de Amdahl, considerando uma fração sequencial $\beta \approx 0,000058$, o speedup teórico máximo para 8 threads seria próximo de 8,0, muito acima dos 2,89 medidos em $N = 50\,000$. Logo, percebe-se que há outras fontes de ineficiência — como contenção de memória, cache misses e barreiras frequentes — que restringem o ganho paralelo.

IX Escalabilidade dos Algoritmos

A escalabilidade forte — em que se mantém N fixo e se aumenta p — mostra ganhos visíveis até 4 threads, porém decrescentes além desse ponto. Por exemplo, dobrar de 4 para 8 threads em $N = 100\,000$ resulta em speedup de 2,44 para apenas 2,55, um incremento marginal de 4 %. Já a escalabilidade fraca — em que N cresce na mesma proporção que p — também não se sustenta: comparar $(N = 50\,000, p = 4)$ e $(N = 100\,000, p = 8)$ revela tempos próximos de 3,44 s e 7,84 s, quase dobrando, quando o ideal seria manter tempos constantes. Assim, o algoritmo não apresenta boa escalabilidade fraca.

X Discussão

De modo geral, o paralelismo por *wavefront* consegue explorar adequadamente as dependências de dados presentes nas etapas iniciais da execução. No entanto, à medida que o número de threads aumenta, observam-se limitações de desempenho causadas principalmente por overheads de sincronização (barreiras) e efeitos adversos relacionados à hierarquia de memória, como a contenção e o aumento das taxas de *cache misses*.

Uma abordagem promissora para mitigar esses problemas seria a adoção de técnicas de *cache blocking* (ou *tiling*). Essa estratégia consiste em subdividir a matriz de dependências em blocos menores, de forma que os dados processados em uma dada etapa caibam em níveis mais rápidos da hierarquia de cache. Como consequência, espera-se uma melhora significativa na localidade espacial e temporal dos acessos à memória, reduzindo a incidência de *cache misses* — um gargalo relevante no algoritmo de LCS devido à sua natureza iterativa e ao padrão de acesso disperso.

Repositório: https://github.com/joaop-vr/lcs_parallel/