Paralelização do Algoritmo LCS usando OpenMPI

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

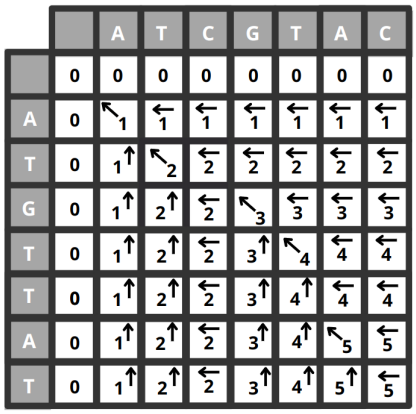


Figura 1: Matriz preenchida para A = **ATCGTAC** e B = **ATGTTAT**.

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.

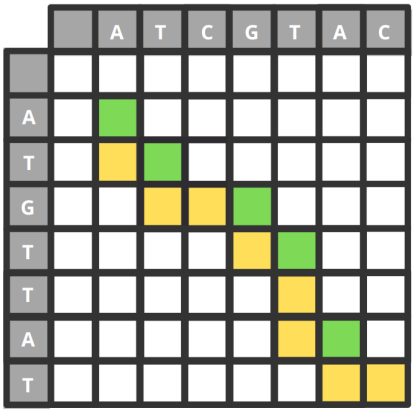


Figura 2: Caminho para A = **ATCGTAC** e B = **ATGTTAT**.

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 decomposição 2D da matriz de pontuação com comunicação *wavefront* entre processos. O código abaixo ilustra a abordagem:

```
// Configuracao da grid 2D
MPI_Dims_create(...); // Define dimensoes da grid
MPI_Cart_create(...); // Cria comunicador da grid
MPI_Cart_coords(...); // Obtem coordenadas (pi, pj)

// Wavefront
for (int wave = 0; wave < wave_total; wave++) { // Para cada diagonal
    for (int i = 0; i < dims[0]; i++) { // Varre linhas da grade
        for (int j = 0; j < dims[1]; j++) { // Varre colunas da grade
            // Processos na diagonal atual executam
            if (pi + pj == wave && pi == i && pj == j) {
                // Receber bordas dos vizinhos
                if (pi > 0) MPI_Recv(...);
                if (pj > 0) MPI_Recv(...);
                if (pi>0 && pj>0) MPI_Recv(...);

                // Calcular bloco local
                for (int ii = 1; ii <= size_i; ii++) {
                    for (int jj = 1; jj <= size_j; jj++) {
                        // Calculo do LCS usando P4 e bordas recebidas
                    }
                }

                // Enviar bordas para vizinhos
                if (pi < dims[0]-1) MPI_Send(...);
                if (pj < dims[1]-1) MPI_Send(...);
                if (pi<dims[0]-1 && pj<dims[1]-1) MPI_Send(...);
            }
        }
    }
}
```

A. Lógica de Operação

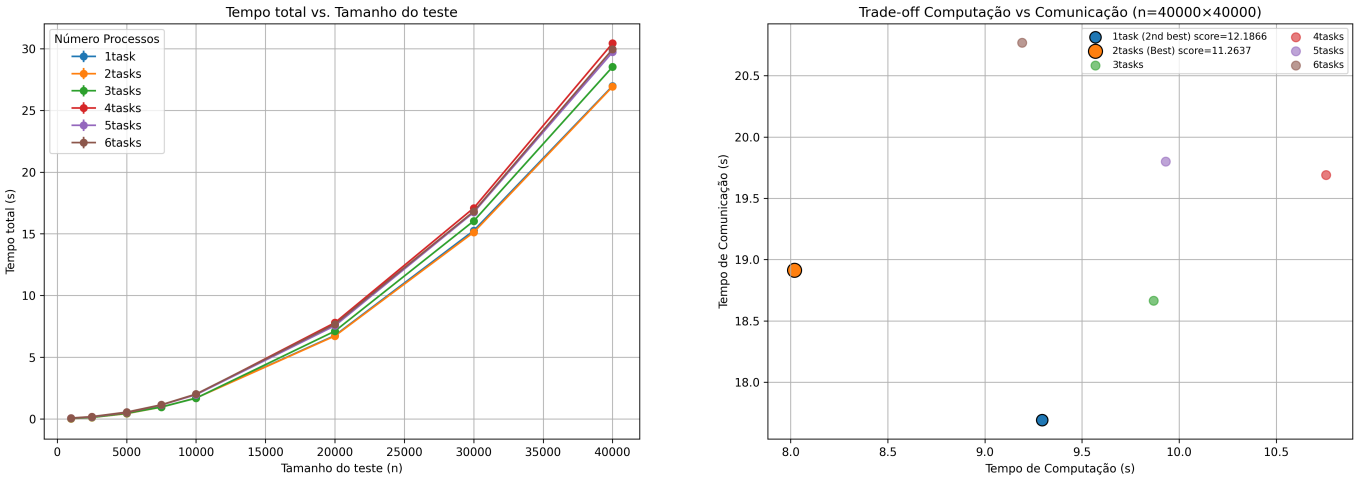
O algoritmo processa a matriz de pontuação em quatro etapas principais:

- **Construção da tabela P4:** Antes de iniciar a DP paralela, constrói-se um índice invertido $P4[b][j]$ que, para cada base $b \in \{A, C, G, T\}$ e cada coluna j da sequência A, armazena a última posição em que b apareceu até j. Esse pre-cálculo permite consultas em $O(1)$ durante o DP, substituindo buscas lineares pela última ocorrência.
- **Decomposição da Matriz:** A matriz global de dimensões $(|B|+1) \times (|A|+1)$ é particionada entre os processos em uma malha 2D de tamanho $\text{dims}_0 \times \text{dims}_1$. Cada processo de coordenadas (p_i, p_j) recebe um bloco de linhas $[\text{start}_i, \ldots, \text{end}_i]$ e colunas $[\text{start}_j, \ldots, \text{end}_j]$, de tamanho $\text{size}_i \times \text{size}_j$, incluindo as bordas de índice zero.
- **Comunicação Wavefront:** O preenchimento dos blocos segue ondas diagonais. Antes de calcular, cada processo recebe os resultados dos seus vizinhos: `top_border`, `left_border` e `top_left_corner` (se existirem). Após o cálculo, os processos devem passar adiante a sua última linha, última coluna e contador `LCS` parcial para seu vizinho à direita, abaixo e em diagonal, respectivamente.
- **Cálculo LCS Local:** Cada processo preenche o seu bloco logal de forma totalmente sequencial, respeitando os três vizinhos já calculados: o valor imediatamente acima (vizinho de cima); o valor imediatamente à esquerda (vizinho da esquerda); e valor na diagonal superior-esquerda (vizinho diagonal). Para cada par de símbolos das sequências, o processo faz o seguinte:
  - Se os símbolos forem iguais, ele pega o valor do vizinho diagonal e soma 1.
  - Caso contrário, ele escolhe o maior entre:
    - O valor do vizinho de cima;
    - O valor do vizinho da esquerda;
    - Um "salto" até a última ocorrência daquele símbolo na outra sequência — usando a tabela pré-computada P — e soma 1 ao valor daquele ponto.

B. Melhor Configuração

A melhor configuração de número de processos por nó foi validada pelos testes descritos na Seção IV-B. Como ilustrado na Figura 3a, à medida que a quantidade de processos por nó aumenta, observa-se um incremento no tempo de execução do programa. Isso ocorre porque, com o aumento do número de processos, a matriz de entrada é particionada em blocos menores, o que gera um maior número de ondas (*wavefronts*) e, consequentemente, eleva o *overhead* de comunicação e sincronização entre os processos.

Dessa forma, as configurações que apresentaram os melhores tempos foram as de 1 processo por nó e 2 processos por nó. Para anular o empate entre elas, realizou-se uma análise baseada no *trade-off* entre tempo de computação e tempo de comunicação para cada configuração, considerando especificamente a entrada de 40000×40000 (Figura 3b).



(a) Tempo total de execução em função do número de processos.

(b) Trade-off entre tempo de computação e comunicação.

Figura 3: Comparação de desempenho e análise de trade-off.

Com base nessa análise de score harmônico, definiu-se que:

- Melhor configuração: `2 processos` (score harmônico = 11.263722).
- Segunda melhor: `1 processo` (score harmônico = 12.186583).

Portanto, a configuração de `2 processos` por nó foi escolhida como a ótima, por apresentar o menor score harmônico e, assim, oferecer o melhor comprometimento entre os tempos de computação e comunicação.

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 um cluster com as seguintes características de hardware:

- **Arquitetura:** x86_64 (Little Endian)
- **Processador:** AMD FX™-6300 Six-Core Processor
  - 1 socket, 3 núcleos por socket, 2 threads por núcleo (total de 6 CPUs lógicas)
  - Frequência nominal de 1,4 GHz a 3,5 GHz (modo boost habilitado)
  - Cache: L1d 96 KiB, L1i 192 KiB, L2 6 MiB, L3 8 MiB
- **Virtualização:** AMD-V
- **NUMA:** 1 nó (CPUs 0–5)

O sistema operacional utilizado foi Linux, compilado com suporte a MPI e bibliotecas padrão de comunicação.

B. *Procedimento Experimental*

Para avaliar o desempenho da implementação paralela, seguiu-se este protocolo:

- O executável paralelo foi compilado com a flag `-O3`, otimização que ativa inline e outras transformações para reduzir tempo de execução;
- Os scripts `sbatch` empregaram a diretiva *–cpu-freq=3500000-3500000:performance*, garantindo frequência fixa de 3.5 GHz para reduzir variações de clock;
- Utilizaram-se 2 nós fixos, variando de 1 a 6 processos por nó (2 a 12 processos MPI no total);
- Entradas dos testes eram de dimensão $N \times N$, para $N \in \{1000, 2500, 5000, 7500, 10000, 20000, 30000, 40000\}$.

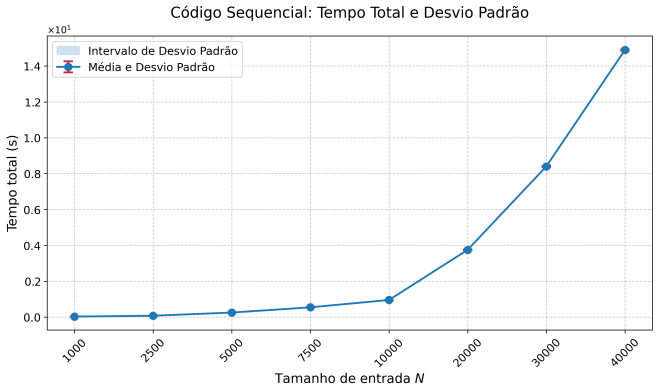
Para cada combinação de parâmetros, o teste foi executado **21 vezes consecutivas**, registrando os tempos em arquivos separados. Além disso, os arquivos de entrada foram gerados previamente usando um utilitário (`generate_input`) para criar sequências aleatórias, garantindo reprodutibilidade dos testes.

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.

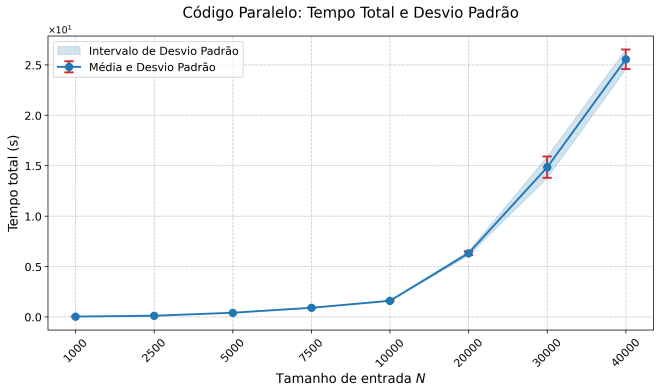
Por fim, para as tabelas de medição de tempo, *speedup* e eficiência foi utilizado a estratégia vitoriosa para casos gerais criados aleatoriamente pelo utilitário `generate_input` de matrizes NxN, com N variando de 20.000 a 40.000.

V Medições de Tempo

A porcentagem de tempo médio que o algoritmo passa na seção paralela foi de 97.4462%. 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 2.5538%. A comparação entre o tempo de execuções das versões puramente sequenciais e paralelas é representada pelas Figuras 4a e 4b.



(a) Tempo médio da versão sequencial.



(b) Tempo médio da versão paralela (2 processos).

Figura 4: Comparações de tempo entre versões sequencial e paralela.

Tabela I: Tempos de execução e desvio padrão – Versão Sequencial.

N	**Tempo (s)**	**Desvio Padrão (s)**
1000	0,0230	0,0002
2500	0,0709	0,0045
5000	0,2477	0,0041
7500	0,5395	0,0051
10000	0,9495	0,0050
20000	3,7477	0,0111
30000	8,4005	0,0344
40000	14,8954	0,0345

Tabela II: Tempos de execução e desvio padrão – Versão Paralela (2 processos).

N	**Tempo (s)**	**Desvio Padrão (s)**
1000	0,0194	0,0010
2500	0,1035	0,0027
5000	0,3989	0,0023
7500	0,8880	0,0028
10000	1,5843	0,0508
20000	6,3065	0,2009
30000	14,8343	1,0603
40000	25,5513	0,9753

VI Lei de Amdahl

Para medir o tempo puramente sequencial e a parte paralelizável do algoritmo LCS utilizou-se a função `clock()` em quatro pontos: imediatamente antes da leitura das sequências, após a alocação e inicialização da matriz de pontuações, logo após o cálculo da LCS e ao término de toda a execução. O tempo de inicialização t_{init} é obtido pela diferença entre os marcadores de leitura e de pós-inicialização, o tempo de cálculo t_{lcs} pela diferença entre os marcadores de fim de inicialização e fim do LCS, e o tempo total t_{total} entre os marcadores de início e fim. Assim, define-se $\beta = t_{\text{init}}/t_{\text{total}}$ e $f_p = 1 - \beta$, aplicando depois a Lei de Amdahl:

$$S(n) = \frac{1}{\beta + \frac{1-\beta}{n}}$$

para cada n núcleos e, no limite $n \to \infty$, $S(\infty) = 1/\beta$.

Tabela III: Speedup teórico máximo pela Lei de Amdahl.

Núcleos	β **(parte sequencial)**	**Speedup Teórico**
2	0,025538	1,95
4	0,025538	3,72
8	0,025538	6,79
...	...	...
∞	0,025538	39,16

VII Speedup e Eficiência

Nesta seção, apresentamos os resultados de desempenho em termos de *speedup* e eficiência relativa ao número de processos utilizados. As Tabelas IV e V mostram os valores obtidos para diferentes tamanhos de entrada, executando a estratégia vitoriosa: 2 processos por nó.

Tabela IV: Speedup em função do número de processos totais para diferentes tamanhos de problema.

Tamanho (N)	**1 Proc.**	**2 Proc.**	**4 Proc.**	**6 Proc.**	**8 Proc.**	**10 Proc.**	**12 Proc.**
10	1.0000	0.5983	0.5994	0.5769	0.5085	0.5185	0.5034
20	1.0000	0.5911	0.5943	0.5474	0.5072	0.5139	0.5158
40	1.0000	0.5886	0.5830	0.5677	0.5014	0.5209	0.5103

Tabela V: Eficiência em função do número de processos para diferentes tamanhos de problema.

Tamanho (N)	1 Proc.	2 Proc.	4 Proc.	6 Proc.	8 Proc.	10 Proc.	12 Proc.
10	1.0000	0.2991	0.1498	0.0962	0.0636	0.0518	0.0420
20	1.0000	0.2956	0.1486	0.0912	0.0634	0.0514	0.0430
40	1.0000	0.2943	0.1457	0.0946	0.0627	0.0521	0.0425

VIII Escalabilidade dos Algoritmos

Análise de Escalabilidade Forte: Os valores de *speedup* abaixo de 1 para todos os tamanhos (Tabela IV) confirmam que a versão paralela é mais lenta do que a sequencial, e esse cenário se agrava a medida que aumentamos a quantidade de processos no experimento. Assim, caracteriza-se a ausência de Escalabilidade Forte.

Análise de Escalabilidade Fraca: Comparando a eficiência entre diferentes entradas para um mesmo número de processos (Tabela V), observa-se que ela varia apenas um pouco ao aumentarmos N de 10 para 40. Por exemplo, para 4 processos, a eficiência fica em torno de 14–15% independentemente do tamanho. Esse comportamento indica sinal de Escalabilidade Fraca: à medida que aumentamos o problema proporcionalmente ao número de processos, o custo relativo de comunicação se mantém estável, embora elevado.

Conclusão sobre escalabilidade:

- Não há escalabilidade forte: toda configuração paralela fica atrás do desempenho sequencial.
- Há indícios de escalabilidade fraca: a eficiência não cai drasticamente com o aumento do problema e dos processos, permanecendo em patamares semelhantes para N variando de 10 a 40.

IX Análise dos Resultados

Os experimentos mostraram que a implementação paralela do algoritmo LCS com OpenMPI não apresentou ganhos de desempenho frente à versão sequencial. Em todos os tamanhos testados (N variando de 1 000 a 40 000), a versão paralela foi consistentemente mais lenta, como ilustram as Figuras 4a e 4b.

Principais observações:

1) **Overhead de comunicação dominante:** A sobrecarga imposta pelas trocas de mensagens em cada onda (*wavefront*) e remontagem da matriz no processo 0 superou qualquer benefício de dividir o trabalho. Para N=40 000, a execução paralela com dois processos por nó foi cerca de 71% mais lenta do que a versão sequencial (Tabelas I e II).
2) **Ausencia de escalabilidade forte:** Em vez de acelerar, o aumento do número de processos degradou o desempenho total (ver Figura 3a). Isso indica que o custo de comunicação cresce mais rápido do que a redução do tempo de cálculo local.
3) **Eficiência paralela reduzida:** A eficiência máxima observada foi apenas 29,9% (Tabela V), caindo para menos de 5% quando se utilizam 12 processos simultâneos.

Em outras palavras, a maior parte do tempo é consumida pela comunicação de bordas e pelo envio final dos blocos para o processo 0, não pelo processamento de cada bloco em si. A Figura 5 ilustra que, conforme o número de processos cresce, a fração de tempo dedicada à comunicação aumenta de forma acentuada. Isso decorre de dois fatores principais:

- Mais processos significam mais ondas (*wavefronts*) e, consequentemente, mais trocas de mensagens entre vizinhos no grid.
- A etapa de coleta final — em que todos os processos enviam seus blocos para o processo 0 — passa a representar uma fatia crescente do tempo total à medida que o número de processos aumenta.

No entanto, se desconsiderarmos o tempo gasto na remontagem da matriz (isto é, excluirmos o custo de comunicação puro), observamos um comportamento bem diferente (Tabelas VI e VII).

Tabela VI: Speedup sem remontagem da matriz.

Tamanho (N)	1 Proc.	2 Proc.	4 Proc.	6 Proc.	8 Proc.	10 Proc.	12 Proc.
10	1.0000	1.8669	2.2600	1.8922	1.5152	1.5692	1.6114
20	1.0000	1.8313	2.2655	1.6589	1.5084	1.6218	1.7961
40	1.0000	1.8389	2.1301	1.9028	1.5060	1.6652	1.7475

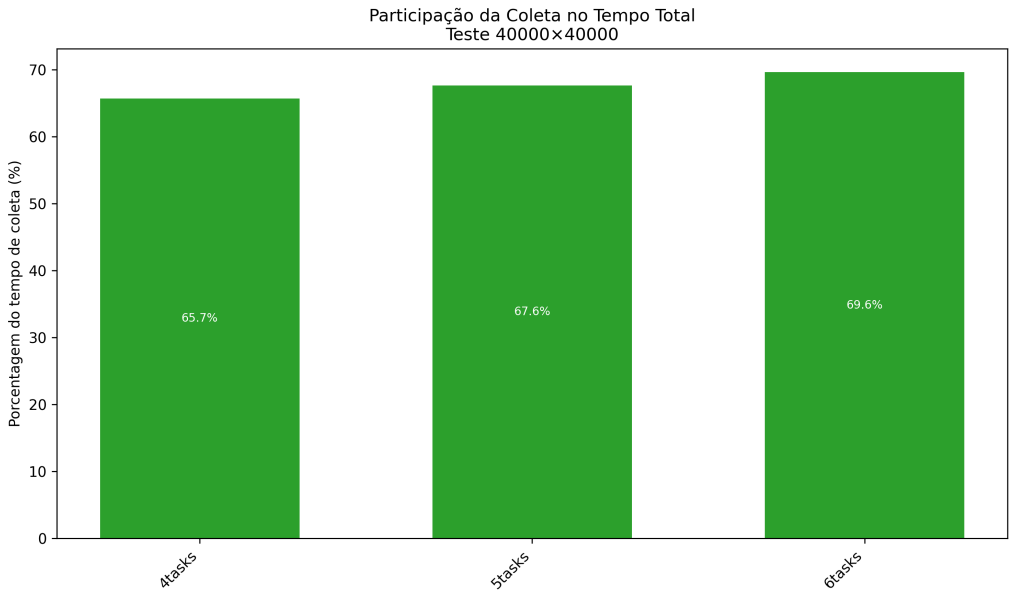


Figura 5: Porcentagem média do tempo de execução dedicada à comunicação (troca de bordas e coleta final).

Tabela VII: Eficiência sem remontagem da matriz.

Tamanho (N)	1 Proc.	2 Proc.	4 Proc.	6 Proc.	8 Proc.	10 Proc.	12 Proc.
10	1.0000	0.9334	0.5650	0.3154	0.1894	0.1569	0.1343
20	1.0000	0.9157	0.5664	0.2765	0.1885	0.1622	0.1497
40	1.0000	0.9195	0.5325	0.3171	0.1883	0.1665	0.1456

Nesses resultados, observa-se um speedup de até 2,26 com quatro processos e eficiência próxima a 93% com a configuração de dois processos. Apesar de ainda não haver escalabilidade forte, esses números mostram que a abordagem paralela exibe *escalabilidade fraca*: a eficiência se mantém razoavelmente estável quando aumentamos o tamanho de entrada mantendo o mesmo número de processos.

X Conclusão

Neste trabalho, investigamos a paralelização do algoritmo da Maior Subsequência Comum (LCS) usando OpenMPI e o comparamos à implementação sequencial clássica por programação dinâmica. Apesar de a abordagem 2D da matriz e o esquema de comunicação em "wavefront" permitirem distribuir o trabalho por vários processos, nossos experimentos mostraram que a execução paralela nunca superou o desempenho sequencial para os tamanhos testados (até N=40.000). O speedup permaneceu sempre abaixo de 1, evidenciando que o overhead de troca de mensagens e sincronização acaba anulando potenciais ganhos de paralelização.

Adicionalmente, constatou-se que a eficiência de paralelização se manteve em patamares relativamente estáveis — entre 14% e 30% — ao aumentar N de 10 a 40 para um mesmo número de processos, característica típica de um fraco escalonamento. Por outro lado, a escalabilidade forte não se materializou, pois acrescentar mais processos degradou o desempenho total, principalmente devido ao acréscimo no volume de mensagens trocadas.

Por fim, ao isolar o custo de coleta final dos blocos no processo principal — removendo-o da análise — observamos que o cálculo em si pode atingir speedup de até 2,26 e eficiência de cerca de 93% com quatro processos (Tabelas VI e VII). Esse resultado confirma que a fase de coleta centralizada é o principal gargalo à escalabilidade, sugerindo que futuras otimizações devem focar na redução desse custo de comunicação.

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