Relatório - Paralelização do LCS com MPI

Vinícius Fontoura de Abreu
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil

***Index Terms*—LCS; Paralelização; MPI; Dependência de Dados; Blocking.**

I. Introdução

Neste trabalho foi necessário: Paralelizar o algoritmo de busca da maior subsequência comum entre dois vetores (*LCS*) usando OpenMPI, testar a diferença de desempenho entra a versão sequencial e paralela com MPI, e analisar os resultados a partir dos dados de teste.

II. Estratégia de paralelização

O algoritmo que encontra a maior subsequência comum (*LCS*) utilizando programação dinâmica é altamente dependente de dados, o que torna a paralelização mais desafiadora.

Essa reformulação foi essencial para facilitar a divisão de trabalho entre os processos MPI. Cada processo calcula um bloco de linhas da matriz LCS e só precisa receber a última linha do processo anterior antes de iniciar seus cálculos.

A lógica para os cálculos continua a mesma que no trabalho anterior, mudando apenas a quantidade de linhas que guardamos em memória.

O núcleo do código pode ser visto à seguir:

```
for (int i = 0; i < my_rows; i++) {
    int global_i = my_start + i;

    // Se DEBUG_MODE estiver ativado, usa a matriz local
    // caso contrario, aloca memoria dinamicamente para a linha atual
    #if DEBUG_MODE
    mtype *curr_row = local_matrix[i];
    #else
    mtype *curr_row = (mtype*) calloc(sizeB + 1, sizeof(mtype));
    #endif

    // Calculas as linhas da matriz LCS
    for (int j = 1; j <= sizeB; j++) {
        // se os caracteres sao iguais, incrementa o valor da diagonal anterior
        if (seqA[global_i - 1] == seqB[j - 1]) {
            curr_row[j] = prev_row[j - 1] + 1;
        // se os caracteres sao diferentes, pega o maximo entre a linha anterior e a coluna anterior
        } else {
            curr_row[j] = max(prev_row[j], curr_row[j - 1]);
        }
    }
    // substitui a linha anterior pela linha atual
    // para o proximo loop ou processo calcular
    memcpy(prev_row, curr_row, (sizeB + 1) * sizeof(mtype));

    #if !DEBUG_MODE
    free(curr_row);
    #endif
}
```

III. Experimentos e Metodologia

A. Máquina escolhida para testes

O hardware e sistema operacional do cluster utilizado para os testes são os seguintes:

- 2 nodos contendo um AMD FX(tm)-6300 com 6 cores cada
- Conexão ETH 1Gbps entre os nodos
- 16GB de RAM (compartilhada)
- 2TB de armazenamento (compartilhada)
- Kernel: 6.8.0-62-generic
- OS: Ubuntu 24.04.2 LTS x86_64

A topologia da CPU é a seguinte:

- Arquitetura: x86_64
- 6 núcleos
- 2 threads por núcleo
- Frequência máxima: 3,5GHz (4.1GHz com *turbo Boost* ativo)
- Cache L2: 6 MiB (3 instâncias)
- Cache L3: 8 MiB (1 instância)

B. Testes utilizados

Um script em python foi utilizado para gerar arquivos com entradas de qualquer tamanho.

Foram usados arquivos no formato N_A e N_B para cada um dos testes, onde N é o tamanho da entrada.

Os testes foram realizados com os tamanhos das entradas entre 20.000 e 150.000 caracteres, com um incremento de 10.000 entre os testes. Essa faixa foi escolhida para garantir um tempo mínimo de 10s e evitar o uso excessivo de recursos do cluster.

C. Metodologia

Os experimentos foram todos executados em uma mesma máquina, especificada acima.

O código foi compilado com as flags:

```
-O3 -Wall -Wextra -march=native -ftree-vectorize
```

. O número de *processos* foi fixado em 2, 4, 8, 10 e 12.

Além disso, todos os testes foram executados com as seguintes especificações no arquivo *.sbatch*:

```
#SBATCH --job-name=lcs_batch
#SBATCH --output=slurm_batch_%j.out
#SBATCH --error=slurm_batch_%j.err
#SBATCH --nodes=2
#SBATCH --ntasks=12
#SBATCH --time=4:00:00
#SBATCH --cpu-freq=high
```

IV. Desempenho

Para medir a eficiência do código, utilizamos:

- Comparações entre os tempos de execuções sequenciais e os tempos de execuções paralelas para as entradas de teste.
- Comparações entre os tempos de execuções paralelas para diferentes números de processos.
- Comparações do *speedup* da execução paralela sobre a sequencial para diferentes entradas e números de *processos*.
- Comparação da eficiência entre *processos* para a mesma entrada (escalabilidade Forte)
- Comparação da eficiência entre conforme aumentamos a entrada e as *processos* proporcionalmente (escalabilidade Fraca)

Esses dados foram coletados e organizados em tabelas e gráficos a seguir.

A. Identificando os trechos sequenciais

Para identificar os trechos sequenciais do código, foram medidos os tempos de execução do código sequencial original com entrada de tamanho $100k$. A média de 20 execuções nos trouxe os seguintes valores:

TotalTime: 12.149757 seconds SeqTime: 0.19239 seconds ParTime: 11.957367 seconds

B. Speedup teórico usando a lei de Amdahl

Speedup teórico usando a lei de Amdahl para 2, 4, 8, 10, 12 e N processadores e 1.6% de código sequencial:

Tabela I
Speedup teórico baseado na Lei de Amdahl

Processadores	Speedup Teórico
2	1.97
4	3.70
8	6.52
10	7.34
12	8.02
∞	62.89

C. Tabela de Speedup e Tempo reais

Após executar os testes, as seguintes tabelas foram geradas:

Médias de tempo:

Tabela II
Tempos médios de execução (em segundos) para diferentes números de processos

Entrada	Seq	2	4	8	10	12
20000	0.5209	0.2517	0.2684	0.3359	0.3301	0.3242
30000	1.1548	0.5544	0.5857	0.6971	0.7039	0.7106
40000	2.0416	0.9816	1.0369	1.1811	1.2216	1.2621
50000	3.1639	1.5354	1.6188	1.8261	1.8924	1.9586
60000	4.5471	2.2089	2.3316	2.6531	2.6958	2.7385
70000	6.4604	3.0059	3.1641	3.5491	3.6389	3.7288
80000	8.3797	3.9356	4.1395	4.6597	4.7472	4.8348
90000	10.4631	4.9799	5.2368	5.8896	6.0640	6.2384
100000	12.9005	6.1408	6.4738	7.1702	7.3669	7.5635
110000	15.5801	7.4395	7.8149	8.8473	9.0804	9.3135
120000	18.5052	8.8457	9.2997	10.3629	10.5982	10.8334
130000	21.7833	10.3735	10.9078	12.0726	12.4157	12.7587
140000	25.1346	12.0303	12.6571	14.1917	14.5773	14.9628
150000	28.7135	13.7812	14.5125	15.9311	16.4402	16.9492

Desvios Padrão:

Tabela III
Desvios padrão (em segundos) para diferentes números de processos

Entrada	Seq	2	4	8	10	12
50000	0.0152	0.5441	0.7674	1.0973	1.1741	1.2509
60000	0.0243	0.7826	1.1057	1.5960	1.6700	1.7439
70000	0.0266	1.0645	1.5002	2.1283	2.2532	2.3781
80000	0.0391	1.3947	1.9618	2.7951	2.9349	3.0746
90000	0.0518	1.7645	2.4839	3.5349	3.7541	3.9733
100000	0.0631	2.1751	3.0798	4.2833	4.5420	4.8007
110000	0.0686	2.6390	3.7046	5.3084	5.6175	5.9266
120000	0.0851	3.1402	4.4082	6.1990	6.5389	6.8788
130000	0.0739	3.6744	5.1657	7.2141	7.6559	8.0976
140000	0.1046	4.2606	5.9951	8.4983	9.0037	9.5090
150000	0.0511	4.8767	6.8745	9.5065	10.1373	10.7680

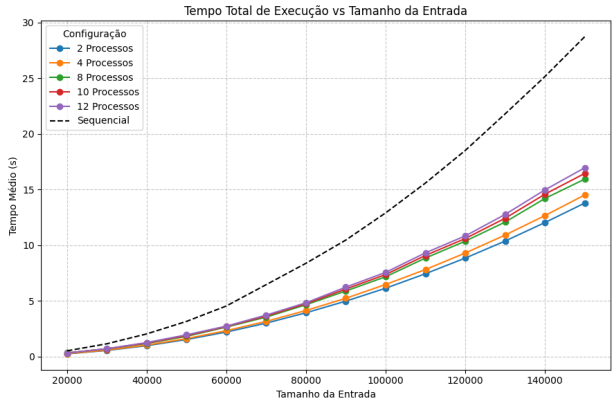
Speedup:

Tabela IV
Speedups para diferentes números de processos

Entrada	2	4	8	10	12
50000	2.0606	1.9545	1.7326	1.6740	1.6154
60000	2.0586	1.9502	1.7139	1.6872	1.6605
70000	2.1492	2.0418	1.8203	1.7764	1.7326
80000	2.1292	2.0243	1.7983	1.7658	1.7332
90000	2.1011	1.9980	1.7765	1.7269	1.6772
100000	2.1008	1.9927	1.7992	1.7524	1.7056
110000	2.0942	1.9937	1.7610	1.7169	1.6729
120000	2.0920	1.9899	1.7857	1.7470	1.7082
130000	2.0999	1.9970	1.8044	1.7559	1.7073
140000	2.0893	1.9858	1.7711	1.7255	1.6798
150000	2.0835	1.9785	1.8024	1.7483	1.6941

Melhor visualizadas através dos gráficos:

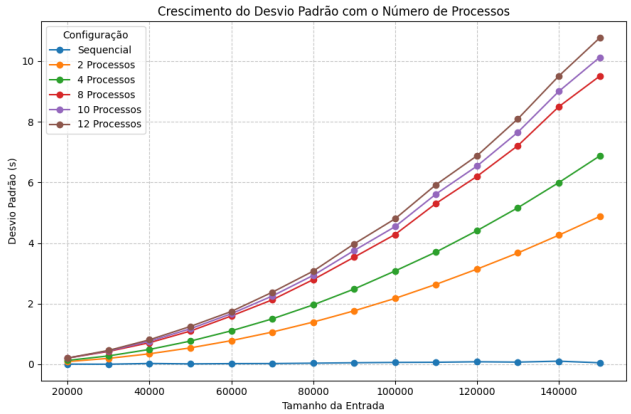
Médias de Tempo:



É fácil perceber que para todas os tamanhos de entrada o algoritmo MPI foi mais rápido que o sequencial, principalmente conforme o tamanho da entrada **aumenta**.

Entretanto, o algoritmo utilizando 2 processos é mais rápido que utilizando mais processos. Isso se deve ao crescente custo de comunicação entre os processos. Esse ponto será abordado a frente.

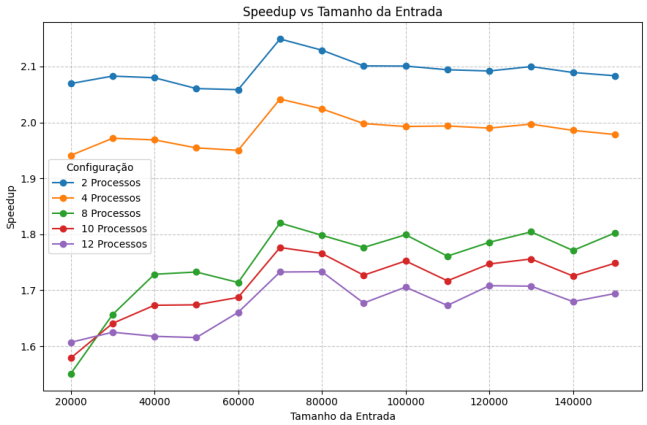
Desvio Padrão:



É possível visualizar que o desvio padrão aumenta rapidamente conforme o número de processos também aumenta. Isso é um indicativo de **inconsistência no tempo de execução** entre os testes.

Isso se deve principalmente a dois fatores: Uso compartilhado e mal gerenciado dos recursos do cluster & Comunicação crescente entre os processos dado o meio utilizado (rede Ethernet). Para evitar esse problema seria necessário uma melhor distribuição dos recursos entre os usuários do cluster e/ou uma infraestrutura mais pontente.

Speedup:



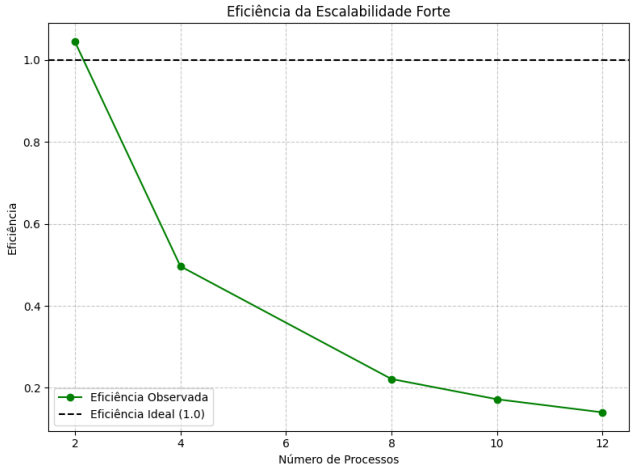
Como o gráfico de $Tempo\ vs\ Entrada$ indica, o maior *speedup* é utilizando dois processos. Ainda sim, com todos os números de processos utilizados o *speedup* foi maior que 1.0.

V. Escalabilidade

A. *Forte*

Para testar a escalabilidade Forte, foi selecionada a entrada de tamanho 140.000 e variando apenas o número de *processos*.

A partir das execuções, temos o seguinte:



É possível ver que o algoritmo tem uma eficiência menor conforme o número de *processos* aumenta, enquanto deveria se manter próximo de 1.0. Logo, o algoritmo **não é** fortemente escalável.

Essa queda na eficiência demonstra que apenas aumentar o número de processos para resolver uma entrada de tamanho X não terá o resultado esperado (o dobro de processos = metade do tempo)

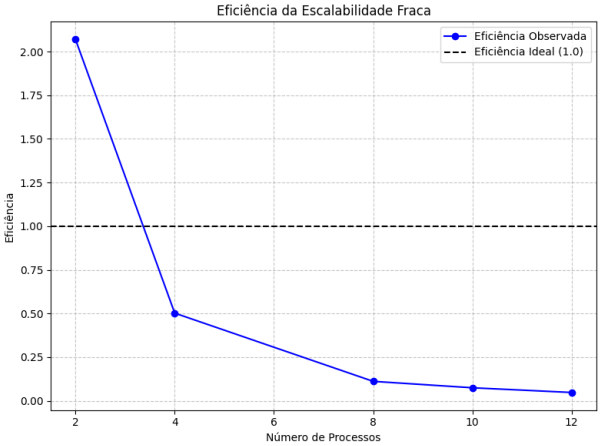
B. *Fraca*

Para testar a escalabilidade Fraca, foram selecionadas entradas proporcionais à quantidade de *processos*. portanto, para a entrada 20.000 - 2 *processos*; 40.000 - 4 *processos*; e assim por diante. Executando, a seguinte tabela foi gerada:

Tabela V
EFICIÊNCIA EM % PARA DIFERENTES NÚMEROS DE PROCESSOS

Entrada	2	4	8	10	12
20000	206.95				
40000		50.24			
80000			11.18		
100000				7.50	
120000					4.81

Melhor visualizada por:



É possível visualizar que o tempo triplica conforme as entradas dobram. Portanto, o algoritmo **não** é fracamente escalável. Visto que ao aumentar o número de processos e o tamanho do problema proporcionalmente, o algoritmo não foi capaz de ”absorver”o aumento do problema.

VI. ANÁLISE DE RESULTADOS

A. Método de paralelização

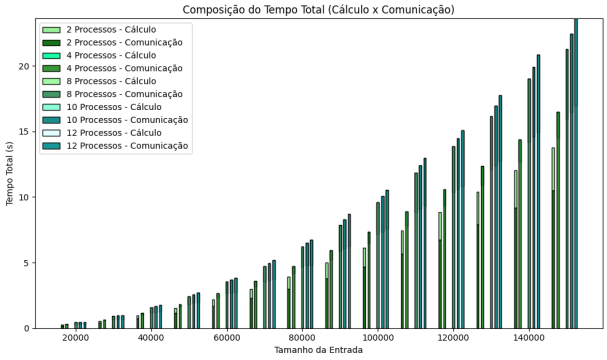
Outras alternativas de melhorar a paralelização do código foram utilizadas, por exemplo:

- Usar a técnica proposta em ”An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs”. Mas este consumia tempo de mais em comunicação e os testes não foram promissores
- Utilizar blocking no algoritmo de programação dinâmica (anti-diagonais) feito no T1 mas este também não se mostrou promissor

B. Tempo gasto em comunicação

A comunicação entre os processos é essencial para algoritmos implementados utilizando MPI. Em uma única máquina isso não é um problema, visto que a velocidade de comunicação entre núcleos ou processos acontece no mesmo barramento ou SO respectivamente.

Essa situação muda quando estamos trabalhando com nós distribuidos. A seguir um gráfico com o tempo $gastoemcomunicação + cálculosdoalgoritmo$ para cada tamanho de entrada e quantidade de processos



É possível visualizar que a maior parte do tempo total de execução foi na comunicação entre os processos.

C. Escalabilidade

Visto que o *LCS* é um problema altamente dependente de dados e sua paralelização gera um *overhead* muito custoso de comunicação entre os processos.

Além disso, é possível ver que: aumentar proporcionalmente o número de processos conforme a entrada aumenta, **não mantém a eficiência** esperada, assim como aumentar o número de processos para uma mesma entrada também perde eficiência. Portanto o algoritmo implementado **não é** Forte **nem** Fracamente escalável.

D. Metodologia dos testes

Para realizar testes mais extensivos seria necessário mais memória *cores* e mais memória *RAM* no cluster, visto que o algoritmo tem um consumo alto de memória. Além disso, aqui estão algumas sugestões para haver mais abrangência nos testes:

- Testes alterando os métodos de memória virtual
- Utilizar *GPU*

VII. CONCLUSÃO

Por fim, os resultados obtidos demonstram que o maior gargalo na paralelização usando MPI parece ser o alto custo de comunicação, está pode ser melhorado alterando o algoritmo para enviar dados em blocos ou melhorando a infraestrutura do cluster.