LCS Paralelo

Luiz Henrique Murback Wiedmer
GRR20221234

1.Introdução

Este documento tem como intenção mostrar um método para a paralelização do algoritmo LCS(Longest Common Subsequence) utilizando MPI, métodos de teste do algoritmo, resultados dos testes e conclusões com relação ao nível de eficiência do algoritmo apresentado em comparação com o algoritmo serial e com os resultados esperados baseados em cálculos ideais.

2.Algoritmo

O algoritmo para cálculo da LCS utilizado como base é baseado em programação dinâmica, evitando o custo computacional alto de $O(2^{min(n,m)})$ da versão ingênua(algoritmo recursivo), por um custo de $O(n*m)$. O algoritmo em C é:

```
int LCS(mtype ** scoreMatrix, int sizeA, int sizeB, char * seqA, char *seqB) {
    int i, j;
    double start_time, endtime;
    for (i = 1; i < sizeB + 1; i++) {
        for (j = 1; j < sizeA + 1; j++) {
            if (seqA[j - 1] == seqB[i - 1]) {
                /* if elements in both sequences match,
                 the corresponding score will be the score from
                 previous elements + 1*/
                scoreMatrix[i][j] = scoreMatrix[i - 1][j - 1] + 1;
            } else {
                /* else, pick the maximum value (score) from left and upper elements*/
                scoreMatrix[i][j] =
                        max(scoreMatrix[i-1][j], scoreMatrix[i][j-1]);
            }
        }
    }
    return scoreMatrix[sizeB][sizeA];
}
```

Texto 1: Código LCS Serial em C

O funcionamento é simples: é utilizada uma matriz para contar o comprimento da subsequência em cada ponto das strings(o valor na posição scoreMatrix[i][j] da matriz representa o comprimento da maior subsequência comum entre os prefixos seqA[0..i-1] e seqB[0..j-1]), caso os elementos analisados sejam iguais, o valor de scoreMatrix[i][j] na tabela será o valor de (scoreMatrix[i-1][j-1] + 1), que é o tamanho da subsequência até aquele momento, mais o caractere analisado. Se forem diferentes, é escolhido o maior valor entre scoreMatrix[i-1][j] e scoreMatrix[i][j-1], o que faz com que os caracteres diferentes sejam ignorados, e o algoritmo continue calculando a subsequência mais longa.

3.Estratégia de Paralelização

A maior dificuldade ao tentar paralelizar esse algoritmo é o fato de que o cálculo de cada célula da matriz depende das células na diagonal esquerda superior, na esquerda e acima, o que faz com que uma paralelização ingênua, calcular os elementos de cada linha de maneira paralela, por exemplo, não funcione. Por conta disso, a tática mais básica a ser aplicada é calcular cada diagonal de maneira paralela, descendo a partir do elemento scoreMatrix[1][1](já que a linha e a coluna 0 já foram enchidos com 0 para possibilitar o cálculo), assim, as dependências sempre estarão prontas quando necessárias.

O problema de usar esse método no contexto de MPI é que a memória dos processos executando em paralelo não é compartilhada, logo, deve haver comunicação explícita entre eles, e quanto mais comunicação, maior o atraso do programa. Por isso, soluções com glanularidade fina não são ideais. Levando isso em conta, a matriz original foi separada em blocos, de maneira que cada processo lida com uma coluna de blocos, sempre com o paralelismo em diagonal. Quando existe um bloco a direita do que acabou de ser processado, a coluna final desse bloco será enviado para o da direita, e como cada processo lida com uma coluna, as dependências da linha de cima de um bloco sempre vão estar na memória do próprio processo.

Ao fim do cálculo apenas o bloco com o resultado final é trazido para o root. Caso toda a matriz fosse remontada em memória, o tempo de execução do algoritmo ficaria pior do que o tempo sequencial, já que seria necessário mais comunicação ainda.

4.Metodologia

Para a realização dos testes do algoritmo, foram utilizadas as seguintes máquinas:
Máquina 1(rodou o teste sequencial também):
SO: Linux Mint 22.1 Cinnamon
Kernel do Linux: 6.8.0-62-generic
Processador: 12th Gen Intel© Core™ i5-12400F × 6
Cache L1: 80KB(por núcleo)
Cache L2: 1,25MB(por núcleo)
Cache L3: 18MB(compartilhada)
Máquina 2:
SO: Linux Mint 22.1 Cinnamon
Kernel do Linux: 6.8.0-62-generic
Processador: 10th Gen Intel© Core™ i5-10300H x 4
Instruction Cache L1: 32 KB(por núcleo)
Data Cache L1: 32 KB(por núcleo)
Cache L2: 256 KB(por núcleo)
Cache L3: 8 MB(compartilhada)
Programa:

Compilador: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Flags: -O3
MPI: Open MPI 4.1.6
Latência da rede: Aproximadamente 0.5ms
Ferramenta para contagem do tempo do programa paralelo: MPI_Wtime()
Ferramenta para contagem do tempo do programa sequencial: omp_get_wtime()
O tamanho do blocking utilizado foi 256x256.
Cada teste foi realizado 20 vezes, e os seguintes resultados foram observados:

Tabela 1: Tempo Médio/Desvio padrão de cada teste

	Processos						
Tam	Serial	1	2	4	6	8	10
20000	0.66/0.00	1.07/0.00	0,69/0.00	0.54/0.00	0.49/0.00	0.45/0.00	0.44/0.00
40000	2.63/0.00	3.55/0.00	1.98/0.00	1.39/0.01	1.12/0.00	0.88/0.01	0.79/0.04
60000	5.90/0.01	7.60/0.02	4.13/0.00	2.78/0.02	2.16/0.04	1.58/0.02	1.35/0.02
80000	10.65/0.01	13.09/0.06	7.17/0.03	4.80/0.05	3.75/0.22	2.62/0.07	2.14/0.02
100000	16.54/0.01	20.29/0.09	10.97/0.01	7.27/0.04	5.59/0.04	3.96/0.23	3.22/0.13

5. Região Sequencial e Speedup Ideal

Para possibilitar o cálculo do Speedup máximo ideal, foi necessário medir o tempo que o algoritmo sequencial gastou para realizar tarefas que não seriam paralelizadas. Para isso, em cada teste sequencial foi calculado o tempo total do programa(como mostrado na Tabela 1), e o tempo apenas do algoritmo LCS(com a função omp_get_wtime()), subtraindo esses dois valores foi possível obter o tempo puramente sequencial, e utilizando esses dados, a porcentagem média de tempo puramente sequencial entre as diferentes entradas foi de 4,37%. Com isso, é possível alcançar a seguinte tabela de speedup seguindo a Lei de Amdahl:

Tabela 2: Speedup ideal para diferentes entradas e número de processadores

		Processos					
Tam	Fração Sequencial	2	4	6	8	10	Infinitos
20000	0.058	1.89	3.40	4.65	5.69	6.57	17.24
40000	0.043	1.91	3.54	4.93	6.14	7.20	23.25
60000	0.037	1.92	3.60	5.06	6.35	7.50	27.02
80000	0.048	1.90	3.50	4.83	5.98	6.98	20.83
100000	0.041	1.92	3.56	4.97	6.21	7.30	24.39

E com ela, é possível também construir a seguinte tabela de eficiência ideal máxima:

Tabela 3: Eficiência ideal para diferentes entradas e número de processadores

	Processos					
Tam	1	2	4	6	8	10
20000	1	0.95	0.85	0.77	0.71	0.65
40000	1	0.96	0.89	0.82	0.77	0.72
60000	1	0.96	0.90	0.84	0.79	0.75
80000	1	0.95	0.87	0.80	0.75	0.69
100000	1	0.96	0.89	0.82	0.78	0.73

6. Speedup real e eficiência

Utilizando os resultados mostrados na Tabela 1, é possível calcular o speedup real do algoritmo paralelo desenvolvido, produzindo assim a seguinte tabela:

Tabela 4: Speedup real para diferentes entradas e número de processadores

		Processos					
Tam	Sequencial	1	2	4	6	8	10
20000	1	0.61	0.95	1.22	1.34	1.46	1.5
40000	1	0.74	1.32	1.89	2.34	2.98	3.32
60000	1	0.77	1.42	2.12	2.73	3.73	4.37
80000	1	0.81	1.48	2.21	2.84	4.06	4.97
100000	1	0.81	1.48	2.23	2.95	4.17	5.13

E com ela, podemos construir uma tabela de eficiência real, e comparar os resultados com a Tabela 3 para melhor entendimento dos dados:

Table 5: Eficiência real para diferentes entradas e número de processadores

		Processos					
Tam	Sequencial	1	2	4	6	8	10
20000	1	0.61	0.47	0.30	0.22	0.18	0.15
40000	1	0.74	0.66	0.47	0.39	0.37	0.33
60000	1	0.77	0.71	0.53	0.45	0.46	0.43
80000	1	0.81	0.74	0.55	0.47	0.50	0.49
100000	1	0.81	0.74	0.55	0.49	0.52	0.51

Ao comparar a eficiência ideal máxima com a eficiência real obtida a partir dos testes, é possível verificar que o algoritmo está longe da eficiência teórica. Esse resultado está dentro do esperado, por conta do custo de overhead do algoritmo(gerenciamento de processos e latência da comunicação). O comportamento fora do esperado foi o fato de 8 e 10 processadores terem tido eficiência melhor do que 6 processadores. Isso foi inesperado porque para utilizar 8

e 10 processadores, foram utilizadas duas máquinas conectadas pela rede, logo o atraso deveria ser maior. O que pode ter ocorrido é que, por conta da máquina 1 ser quem está gerenciando a rede, o processamento feito unicamente por ela foi pior do que o dividido. Outra possibilidade é que houveram inconsistências no comportamento delas por conta do Turbo Boost da Intel, que infelizmente não pode ser desativado para os testes por conta da Máquina 2 ter a BIOS muito restritiva.

Com esses resultados, podemos verificar que, para essas entradas, o algoritmo não é escalável, já que mesmo aumentando o tamanho da entrada, a eficiência aumenta muito pouco comparando o mesmo número de processos.

Baseado nisso, surge a dúvida se a paralelização desse algoritmo utilizando o método apresentado, faz sentido em contextos reais, especialmete para grandes números de processos.

7.Conclusão

Baseando-se nos dados aqui apresentados, é possível afirmar que, devido a não escalabilidade do algoritmo, nem sempre é interessante utilizar o máximo possível de threads ao paralelizar o algoritmo LCS da forma apresentada.

Além disso, os tempos apresentados para o algoritmo foram apenas para encontrar a score da LCS, o que seria útil apenas em cenários bem limitados. Pensando em uma implementação que deixa a matriz DP em memória, o tempo acaba sendo maior que o tempo do algoritmo sequencial, justamente porque a reconstrução da matriz no processo root exige que todos os blocos sejam enviados pela rede até a root e reorganizados.

Logo, podemos dizer que não é interessante paralelizar o algoritmo LCS utilizando Blocking e Wavefront por antidiagonais por meio do MPI.