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), 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 desse método é que, por acesso de memória em diagonal, ocorrem muitos cache-misses, o que faz com que ele tenha velocidades próximas à versão serial, e muitas vezes até piores. Pensando nisso, o seguinte algoritmo em C foi produzido:

```
void processaBloco(mtype** scoreMatrix, int sizeA, int sizeB, int i_block, int j_block, const
char* seqA, const char* seqB) {
    int i_start = i_block * blockSize;
    int j_start = j_block * blockSize;
    int i_end = (i_start + blockSize < sizeB + 1) ? i_start + blockSize : sizeB + 1;
    int j_end = (j_start + blockSize < sizeA + 1) ? j_start + blockSize : sizeA + 1;
    for (int i = i_start; i < i_end; ++i) {
        for (int j = j_start; j < j_end; ++j) {
            if (i == 0 || j == 0) {
                scoreMatrix[i][j] = 0;
            } else if (seqB[i - 1] == seqA[j - 1]) {
                scoreMatrix[i][j] = scoreMatrix[i - 1][j - 1] + 1;
            } else {
                scoreMatrix[i][j] = max(scoreMatrix[i - 1][j], scoreMatrix[i][j - 1]);
            }
        }
    }
}

int LCS(mtype ** scoreMatrix, int sizeA, int sizeB, char * seqA, char *seqB, int
numThreads) {
            int bi = (sizeB + blockSize) / blockSize;
    int bj = (sizeA + blockSize) / blockSize;
    for (int d = 0; d <= bi + bj - 2; ++d) {
        #pragma omp parallel for num_threads(numThreads)
        for (int i = 0; i <= d; ++i) {
            int j = d - i;
            if (i < bi && j < bj) {
                processaBloco(scoreMatrix, sizeA, sizeB, i, j, seqA, seqB);
            }
        }
    }
            return scoreMatrix[sizeB][sizeA];
}
```

Texto 2: Algoritmo Paralelo com Blocking

A ideia é que ao invés de simplesmente calcular cada célula de uma diagonal em paralelo, a matriz é dividida em blocos. Blocos na mesma diagonal são calculados em paralelo, assim como no método anterior, porém o cálculo de cada célula no interior do bloco ocorre de maneira serial(do mesmo jeito explicado no capítulo anterior). Essa mudança melhora muito o uso de cache, especialmente se o tamanho do bloco for escolhido corretamente, além de que faz com que uma thread realize mais trabalho, reduzindo o número de vezes em que novas threads são criadas e o trabalho é divido entre elas, por conta disso, essa versão apresentou tempos melhores, como será mostrado a seguir.

4.Metodologia

Para a realização dos testes do algoritmo, foram utilizadas as seguintes
SO: Linux Mint 22.1 Cinnamon
Kernel do Linux: 6.8.0-59-generic
Processador: 12th Gen Intel© Core™ i5-12400F × 6 (Intel Turbo Boost desligado)
Cache L1: 80KB(por núcleo)
Cache L2: 1,25MB(por núcleo)
Cache L3: 18MB(compartilhada)
Compilador: gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Flags: -O3
Ferramenta para contagem do tempo do programa: perf
Ferramenta para contagem do tempo da parte paraleizada: omp_get_wtime()

O tamanho do blocking utilizado foi 1024x1024, já que esse tamanho apresentou os melhores resultados em testes anteriores, provavelmente por melhor fazer uso da cache (1024x1024 é o maior bloco com base 2 que cabe na cache L2 de um núcleo individual físico do processador dos testes), além de permitir um bom nível de paralelização.

Cada teste foi realizado 20 vezes, e os seguintes resultados foram observados:

Tabela 1: Tempo Médio/Desvio padrão médio de cada teste

	Threads					
Tam	Serial	1	2	4	8	12
20000	1,12/0,025	1,22/0,001	0,68/0,002	0,50/0,001	0,39/0,001	0,34/0,001
40000	4,47/0,072	4,90/0,009	2,64/0,005	1,87/0,003	1,43/0,030	1,21/0,005
60000	9,86/0,103	11,03/0,018	5,86/0,010	4,11/0,015	3,10/0,060	2,58/0,024
80000	17,86/0,097	19,75/0,026	10,55/0,022	7,47/0,012	5,61/0,035	4,73/0,011
100000	27,66/0,127	30,73/0,056	16,28/0,025	11,46/0,016	8,50/0,059	7,06/0,025

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 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 os dados coletados nos testes do algoritmo sequencial original, 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

		Processadores				
Tam	Fração Sequencial	2	4	8	12	Infinitos
20000	0,058	1,890	3,407	5,690	7,326	17,241
40000	0,043	1,918	3,543	6,149	8,147	23,256
60000	0,037	1,929	3,600	6,354	8,529	27,027
80000	0,048	1,908	3,497	5,988	7,853	20,833
100000	0,041	1,921	3,562	6,216	8,270	24,390

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

	Processadores				
Tam	1	2	4	8	12
20000	1	0,95	0,85	0,71	0,60
40000	1	0,96	0,89	0,77	0,68
60000	1	0,96	0,90	0,79	0,71
80000	1	0,95	0,87	0,75	0,65
100000	1	0,96	0,89	0,78	0,69

6. Speedup real e eficiência

Utilizando o algoritmo 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 números de threads

	Threads					
Tam	Sequencial	1	2	4	8	12
20000	1	0,91	1,64	2,24	2,87	3,29
40000	1	0,91	1,69	2,39	3,12	3,69
60000	1	0,89	1,68	2,39	3,18	3,82
80000	1	0,9	1,69	2,39	3,18	3,77
100000	1	0,9	1,69	2,41	3,25	3,91

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

	Threads					
Tam	Sequencial	1	2	4	8	12
20000	1	0,91	0,82	0,56	0,36	0,27
40000	1	0,91	0,85	0,60	0,39	0,31
60000	1	0,89	0,84	0,60	0,40	0,32
80000	1	0,90	0,85	0,60	0,40	0,31
100000	1	0,90	0,85	0,60	0,41	0,33

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(como a criação e gerenciamento da threads). Além de que, no caso de 8 e 12 threads, os resultados necessariamente seriam diferentes, já que o processador utilizado tem 6 núcleos físicos e 12 threads, logo, quando se usa mais de 6 threads, é esperado que a eficiência caia bastante, já que o processador não é fisicamente capaz de paralelizar mais de 6 threads.

Com esses resultados, podemos verificar que, para essas entradas, o algoritmo não é escalável, já que mesmo aumentando o temanho da entrada, a eficiência aumenta muito pouco comparando o mesmo número de threads. Isso provavelmente ocorre porque em várias partes do processo(início e fim, por processar em diagonal), nem todas as threads são aproveitadas. O uso de tasks, criadas por uma thread única, e consumidas pelas outras, provavelmente ajudaria nisso, já que faria com que, cada vez que uma tarefa é terminada, a thread que acabou de terminar a mesma, começaria outra tarefa.

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

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.

Se o usuário quiser o máximo possível de desempenho, vale a pena, já que usar mais threads trará mais ganhos, ainda que modestos, mas para aplicações que devem dividir threads com outras tarefas, faria mais sentido limitar o uso de threads do algoritmo, já que o ganho de desempenho ao utilizar várias threads provavelmente seria menor do que ao entregar as threads para tarefas completamente diferentes.