Programação Paralela

Longest Common Subsequence

Vinicius Tikara Venturi Date

Introdução

Neste relatório iremos discutir sobre a paralelização do algoritmo de maior subsequência em comum(LCS).

Veremos a estratégia escolhida, os dados obtidos a partir dos experimentos e as conclusões que podemos tirar deles.

Algoritmo Sequencial

A parte principal do algoritmo sequencial é esta:

```
int LCS(mtype ** scoreMatrix, int sizeA, int sizeB, char * seqA, char
*seqB) {
   int i, j;
   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];
}
```

Esta parte utiliza a técnica de programação dinâmica para sua resolução, pois o problema possui a dita estrutura ótima, isto é, o problema da maior subsequência em comum pode ser quebrado em subproblemas

menores que, quando juntos, podem responder o problema original, ou seja, achar a subsequência em comum de trechos de ambas strings.

Temos também, a sobreposição desses subproblemas, permitindo, por fim, a utilização da programação dinâmica.

Entretanto, esse algoritmo possui muitas dependências na matriz de pontuação, uma célula dessa matriz depende dos resultados da linha anterior e da coluna anterior, algo que causa problemas ao tentar aplicar a paralelização diretamente.

Estratégia Vitoriosa

A estratégia vitoriosa se deu pela implementação do algoritmo discutido por Yang et al.[1], onde a partir da manipulação da recorrência da seção anterior conseguimos remover a dependência da linha anterior. A manipulação se dá por perceber que após uma dada quantidade de passos, ou nós não vimos ainda a letra igual, ou achamos a posição anterior dela.

Com tal ideia em mente é criada uma matriz P que codifica essa ideia de última posição vista. A matriz tem em um eixo os caracteres únicos das strings a serem comparadas - o alfabeto - e no outro uma das strings a serem comparadas, neste caso a string B. A matriz então codifica quando foi visto pela última vez tal caractere do alfabeto nessa string B.

```
void calculatePMatrix(string unique, string bString,
vector<vector<mtype>> &p){
   #pragma omp parallel for
   for(int i = 0; i < unique.length(); ++i){
       for(int j = 1; j < bString.length()+1; ++j){
           if(bString[j-1] == unique[i]){
               p[i][j] = j;
           } else {
               p[i][j] = p[i][j-1];
           }
       }
   }
}
```

Tendo em mão esta matriz P, conseguimos recuperar a posição em que foi vista o último caractere da string A na outra, conseguindo recuperar o índice em que se encontra dado caractere atual string A na string que contém o alfabeto.

```
int firstEqual(string str, char c){
      for(int i = 0; i < str.length(); ++i){
          if(str[i] == c) return i;
      }
      return -1;
}
```

Desta forma se torna possível o processamento paralelo da linha anterior da matriz de pontuação, uma vez que cada elemento da linha é independente de outro elemento da mesma linha, apenas dependendo do elemento da linha anterior.

```
mtype LCS(vector<vector<mtype>> &scoreMatrix, string a, std:: string b,
vector<vector<mtype>> p, string unique) {
   for (int i = 1; i < a.length()+1; i++) {
       // acha a posição do caractere atual de A em relação
       // ao alfabeto, utilizado na indexação da matriz P
       mtype c = firstEqual(unique, a[i-1]);
       #pragma omp parallel for
       for (int j = 0; j < b.length()+1; j++) {
           // checa os casos da função de transição da programação
dinâmica para a LCS
           // utilizando a matriz P para checar aonde foi visto por
último o iésimo caractere
           // da string A em relação a string B
           if(i == 0 || j == 0){
               scoreMatrix[i][j] = 0;
           } else if(p[c][j] == 0){
               scoreMatrix[i][j] = max(scoreMatrix[i-1][j], 0);
           } else {
               scoreMatrix[i][j] = max(scoreMatrix[i-1][j],
scoreMatrix[i-1][p[c][j]-1] + 1);
           }
       }
   }
   return scoreMatrix[a.length()][b.length()];
}
```

Lei de Amdahl

Executando uma série de vezes na maior entrada que a máquina suporta, foi possível observar que cerca de 22% do programa se encontra na parte sequencial e os restantes 78% se encontram na parte paralelizável.

Utilizando a equação $s(p) = 1/(\beta + (1 - \beta)/p)$, onde β é a porcentagem de tempo sequencial, temos os seguintes speedups teóricos para 1, 2, 3, 4 ,5, 6, 7, 8 e infinitos processadores na tabela a seguir:

Threads	Speedup
2	1,63
3	2,08
4	2,41
5	2,66
6	2,86
7	3,02
8	3,15
∞	4,54

Experimentos

Hardware e ambiente

Os experimentos foram rodados no seguinte processador:

```
AMD Ryzen 5 3600X 6-Core Processor
     cache de L1d:                  192 KiB
     cache de L1i:                  192 KiB
     cache de L2:                   3 MiB
     cache de L3:                   32 MiB
```

Com a seguinte memória RAM:

`16 GiB RAM DDR4 3200Mhz` em dual channel, 8GiB por canal.

Utilizando o sistema operacional Ubuntu na versão 20.04. instalado num HD de 1TiB, com cerca de 490 MiB de swap.

O turbo-boost foi desabilitado na BIOS antes dos experimentos.

O compilador utilizado foi o `g++` versão 9.4.0. O programa foi compilado com as flags `-O3` e `-fopenmp`.

Execução

O programa paralelo foi executado 50 vezes para cada combinação de tamanho de entrada e número de threads possível.

A execução sequencial foi feita no programa com as otimizações paralelas, pois além de outra linguagem(C++), foram feitos alguns passos

extras antes do cálculo do LCS, de tal forma que a comparação com o programa original não é sensata.

Foram utilizadas sequências geradas por um programa escrito em Python, onde uma sequência é uma string de caracteres minúsculos e maiúsculos do alfabeto, sem acentos. As entradas tem tamanho de 10000 caracteres até 60000 caracteres, com incrementos de 10000.
Acima de 60000 caracteres o programa utilizava toda memória do sistema e era consequentemente morto, fazendo com que o tempo de execução máximo obtido fora de cerca de 8 segundos.

Executei com até 6 threads, pois após 6 threads houve uma piora de execução de tempo e, consequentemente, de eficiência. O motivo para tal piora parece ser a utilização dos núcleos virtuais do processador, uma vez que até 6 threads eram usados apenas os núcleos físicos.

Com isso em mente, a partir dos experimentos foi obtida a seguinte média de tempo de execução, em segundos:

N	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads
10000	0.216	0.154	0.132	0.115	0.110	0.110
20000	0.819	0.541	0.446	0.392	0.358	0.358
30000	1.808	1.162	0.940	0.812	0.752	0.740
40000	3.184	2.016	1.620	1.408	1.303	1.261
50000	4.948	3.112	2.483	2.164	1.989	1.917
60000	7.092	4.436	3.538	3.079	2.837	2.723

Com o seguinte desvio padrão:

N	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads
10000	0.003	0.002	0.004	0.003	0.006	0.002
20000	0.002	0.002	0.006	0.007	0.007	0.005
30000	0.007	0.004	0.013	0.008	0.009	0.010
40000	0.005	0.004	0.011	0.013	0.016	0.009
50000	0.012	0.005	0.013	0.012	0.016	0.013
60000	0.012	0.009	0.015	0.012	0.027	0.032

Podemos também obter as seguintes métricas, o speedup

N	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads
10000	1	1.405	1.640	1.876	1.974	1.966
20000	1	1.514	1.836	2.088	2.286	2.289
30000	1	1.557	1.924	2.226	2.405	2.444
40000	1	1.579	1.966	2.261	2.443	2.525
25000	1	1.543	1.895	2.204	2.362	2.403
50000	1	1.590	1.992	2.287	2.487	2.581
60000	1	1.599	2.004	2.304	2.500	2.604

E a eficiência

N	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads
10000	1	0.703	0.547	0.469	0.395	0.328
20000	1	0.757	0.612	0.522	0.457	0.382
30000	1	0.778	0.641	0.556	0.481	0.407
40000	1	0.790	0.655	0.565	0.489	0.421
50000	1	0.795	0.664	0.572	0.497	0.430
60000	1	0.799	0.668	0.576	0.500	0.434

Análise

O desvio padrão da média de tempos foi baixo, nos indicando que os experimentos, na maioria dos casos, tiveram um tempo de execução parecida, ou seja, houve pouca interferência externa na execução dos experimentos.

Analisando os outros dados obtidos na seção anterior podemos concluir da tabela de eficiência que não existe escalabilidade forte no

algoritmo, pois a eficiência decai conforme o número de threads aumenta, mesmo para a maior entrada da tabela. Este fato pode ser observado uma vez que os tempos médios para 5 e 6 threads tem valores muito similares, não ocorre uma diminuição significativa ao ser adicionada outra thread.

Por outro lado o algoritmo admite a escalabilidade fraca, uma vez que se for mantido o número de threads e variado o tamanho da entrada, a eficiência se mantém parecida, até aumentando um pouco para entradas maiores. Ou seja, os valores de eficiência de uma mesma coluna são similares.

Por fim, podemos também comparar a Lei de Amdahl com os experimentos práticos. Utilizando o speedup calculado para a maior entrada(60000) temos a seguinte comparação:

Threads	Speedup Teórico	Speedup Calculado
2	1,63	1.599
3	2,08	2.004
4	2,41	2.304
5	2,66	2.5
6	2,86	2.604

Portanto, a Lei de Amdahl, para este algoritmo e ambiente específicos, nos dá uma estimativa razoável

Referências

[1] Yang J, Xu Y, Shang Y. An efficient parallel algorithm for longest common subsequence problem on gpus. In: Proceedings of the world congress on engineering. 2010. p. 499–504