Relatorio Trabalho 1 OpenMP

Aluno: Leonardo Bueno Nogueira Kruger GRR20180130

github projeto

1. Introdução

Trabalho sobre a paralelização do Algoritmo LCS (Longest Common Subsequence) utilizando a biblioteca OpenMP, e a variação do método LCS em Row-wise independent algorithm. O projeto completo com arquivos de entrada e script utilizado na execução dos resultados estão no github do projeto.

2. Funcionamento Core LCS

Algoritmo LCS (Longest Common Subsequence) Algoritmo utilizado para encontrar a maior subsequencia presente em duas sequencias (Strings), uma subsequencia é caracteriza como uma sequencia que aparece na mesma ordem relativa mas não necessariamente continua.

O Algoritmo trabalha em cima de uma matriz de tamanho sizeA x sizeB onde sizeA é o tamanho da string A e sizeB é o tamanho da string B, com a primeira linha e coluna inicializadas em 0.

```
for (j = 0; j < (sizeA + 1); j++)
        scoreMatrix[0][j] = 0;

for (i = 1; i < (sizeB + 1); i++)
    scoreMatrix[i][0] = 0;
```

A partir dessa estrutura ocorre o LCS sobre a matriz, percorrendo a matriz caso encontre um 'match' de character (char de seqA e seqB correspondem) é pego o valor na diagonal anterior e somado 1 para o tamanho da subsequencia, caso contrario é maior valor do campo superior(cima) ou anterior(esquerda), ao final teremos o valor da maior subsequencia.

```
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];
}
```

3. Estratégia de paralelização

A partir da estrutura core do LCS foi possivel observar uma dependencia de dados que impossibilitava a paralelização do código, a partir do aritgo An OpenMP-based tool for finding longest common subsequence in bioinformatics que propõe soluções sobre problema, decidi por implementar a versão 1 proposta no artigo, Yang et al propõe o 'Row-wise independent algorithm' uma solução diferente das tradicionais anti-diagonal e bit-parallel e a escolha para o meu trabalho. O algoritmo consiste na modificação da equação tradicional do LCS:

$$R[i,j] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ R[i-1,j-1] + 1 & if\ A[i] = B[j] \\ \max\big(R[i-1,j], R[i,j-1]\big) & otherwise \end{cases} \tag{1}$$

Pela versão 'Row-wise':

$$R[i,j] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ R[i-1,j-1] + 1 & if\ A[i] = B[j] \\ \max\big(R[i-1,j], R[i-1,j-k-1] + 1\big) & if\ A = B[j-k] \\ \max\big(R[i-1,j], 0\big) & if\ j - k = 0 \end{cases}$$

Onde k é o numero de passos necessarios para encontrar uma match como seqA[i] == seqB[j-k] ou j-k == 0, para isso é utilizada outra matriz on calculamos os valores de j-k para toda iteração i, dado por:

$$P[i,j] = \begin{cases} 0 & if\ j = 0 \\ j - 1 & if\ B[j-1] = C[i] \\ P[i,j-1] & otherwise \end{cases}$$

```
mtype lcs_yang_v1(mtype **DP, mtype **P, char *A, char *B, char *C, int m,
int n, int u)
{
    {

        for (int i = 1; i < m + 1; i++)
        {
            int c_i = get_index_of_character(C, A[i - 1], u);
```

```
#pragma omp parallel for schedule(static)
            for (int j = 0; j < n + 1; j++)
            {
                if (A[i - 1] == B[j - 1])
                {
                    DP[i][j] = DP[i - 1][j - 1] + 1;
                }
                else if (P[c_i][j] == 0)
                {
                    DP[i][j] = max(DP[i - 1][j], 0);
                }
                else
                {
                    DP[i][j] = max(DP[i - 1][j], DP[i - 1][P[c_i][j] - 1] +
1);
                }
            }
        }
    }
    return DP[m][n];
}
```

```
void calc_P_matrix_v1(mtype **P, char *b, int len_b, char *c, int len_c)
{
#pragma omp parallel for
    for (int i = 0; i < len_c; i++)
    {
        for (int j = 2; j < len_b + 1; j++)
        {
            if (b[j - 2] == c[i])
            {
                P[i][j] = j - 1;
            }
            else
            {
                P[i][j] = P[i][j - 1];
            }
        }
    }
}
```

4. Informações e Metologia de testes

Informação	Descrição
S.O/kernel	Ubuntu 5.15.0-43-generic #46~20.04.1-Ubuntu SMP Thu Jul 14 15:20:17 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux

Informação	Descrição
versão do compilador	gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Flags de compilação	-O3 -Wall -pedantic -pthread -fopenmp
Processador	Intel(R) Core(TM) i3-9100F CPU @ 3.60GHz
Número de execuções	30
Metodologia	Teste de escalabilidade forte
Memoria RAM	16GB 3000MHZ
Compilar	gcc -O3 -Wall -pedantic -pthread -fopenmp parallel.c -o parallel
parametros	Recebe como entrada valores 'AB' '12' 'CD' '34' que definem os arquivos utilizados na execução

Arquitetura processador

```
Arquitetura:                    x86_64
Modo(s) operacional da CPU:     32-bit, 64-bit
Ordem dos bytes:                Little Endian
Address sizes:                  39 bits physical, 48 bits virtual
CPU(s):                         4
Lista de CPU(s) on-line:        0-3
Thread(s) per núcleo:           1
Núcleo(s) por soquete:          4
Soquete(s):                     1
Nó(s) de NUMA:                  1
ID de fornecedor:               GenuineIntel
Família da CPU:                 6
Modelo:                         158
Nome do modelo:                 Intel(R) Core(TM) i3-9100F CPU @ 3.60GHz
Step:                           11
CPU MHz:                        800.112
CPU MHz máx.:                   4200,0000
CPU MHz mín.:                   800,0000
BogoMIPS:                       7200.00
Virtualização:                  VT-x
cache de L1d:                   128 KiB
cache de L1i:                   128 KiB
cache de L2:                    1 MiB
cache de L3:                    6 MiB
CPU(s) de nó0 NUMA:             0-3
```

Check List

Aplicação paralela

- 1. Está correta? Algumas vezes retorna segfault?
- Sim, retorna o resultado correto Entrada
- 2. Tenho um N que roda em pelo menos 10s?
- Não, Acima de 80k linhas trava o computador e menos não chega perto de 10s
- 3. Tenho tamanhos variados? (2N, 4N, 6N)
- Sim, 10k, 20k, 30k, 60k Máquina
- 4. É um servidor virtualizado? Nem pense em rodar na amazon cloud.
- Não
- 5. Estou usando modo usuário? Grub.
- Testes feitos em ambos os ambientes não demonstra variação

5. Resultados

Tempo	Init Matrix	LCS
Serial(1 thread) 60k	0,001314	3,649211
Porcentagem do tempo total	1%	99%

Argumento de amdahl

Tabela de amdahl dada por: O argumento de Amdahl é utilizado para prever o máximo speedup teórico usando múltiplos processadores, a partir da porcentagem do programa paralelizavel. A partir do speedup dado por:

$$S = \frac{T(1)}{T(N)}$$

temos as seguintes formulas, onde podemos obter o T(n) teoricamente e assim calcular o valor de S(n) em função do número de threads e da fração do algoritmo serial/paralelizavel. Através disso podemos montar a tabela teorica de Speedup.

Fórmulas:

- $n \in \mathbb{N}$, o número de threads de execução,
- $B \in [0,1]$, fração de um algoritmo estritamente serial,

O tempo $T(n)$ que um algoritmo demora para terminar a execução utilizando n thread(s) de execução, corresponde a:

$$T(n) = T(1)\left(B + \frac{1}{n}(1-B)\right)$$

Portanto,o speedup teórico $S(n)$ que pode ser obtido pela execução de um dado algoritmo, em um sistema capaz da execução de n threads de execução, é:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} = \frac{1}{B + \frac{1}{n}(1-B)}$$

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

No limite, como "N" tende ao infinito, o speedup máximo tende ser 1 / (1 - P).

Tabela Lei de Amdahl

Lei de Amdahl	1	2	4	8	N
Eficiencia	1	1,98	3,88	7,47	100

Resultados das execuções Seriais e Paralelas Cada algoritmo foi executado pelo menos 30 vezes com as entradas dos tamanho descritos para coleta de dados e amostragem.

Tabela Resultados

em segs	Serial	1 Thread	2 Thread	4 Thread	8 Thread
Média 10k:	0,17669	0,16903	0,09683	0,070976	0,071005
Desv Pad 10k:	0,00188	0,00275	0,00509	0,02414	0,02307
Média 20k:	0,700294	0,650065	0,355678	0,207929	0,208012
Desv Pad 20k:	0,00453	0,00914	0,00535	0,04439	0,04318
Média 30k:	1,594543	1,44017	0,777968	0,521799	0,519987
Desv Pad 30k:	0,00935	0,01746	0,01501	0,18416	0,18115
Média 60k	6,353989	6,32141		1,601886	1,602086
Desv Pad 60k	0,034371	0,01919		0,050562	0,049892

Tabela Speedup

	Tempo Serial ->	1 Speedup ->	2 Speedup	4 Speedup
Média 10k:	0,17669	1,045	1,74	2,38
Média 20k:	0,700294	1,077	1,82	3,12
Média 30k:	1,594543	1,107	1,85	2,76
Média 60k	6.353989	1,005	---	3,94

6. Conclusão

Os resultados dos testes foram satisfatorios, o desvio padrão baixo mostrou que houve poucas interrupções/variações nos tempos de execução dos testes com mesmas variaveis, comparando os resultados da tabela prática de Speedup e a tabela teorica do Argumento de Amdahl demonstra uma relação forte entre os resultados previstos e obtido, provando que a porcentagem de código paralelizavel previsto foi atingido.

O algoritmo apresentou escalabilidade forte, tendo em vista que o Speedup atingiu os valores mais proximos ao argumento de Amdahl com maiores entradas de dados, o código não chegou a atingir execuções de 10's ou mais pois a partir de 80k ocorria o travamento da máquina, para melhores equipamentos pode se tornar mais escalavel ainda.

Os resultados obtidos na perspectiva geral foram ótimos, ocorreu uma pequena variação negativa no Speedup de 4 Threads entre 20k -> 30k porém todos os outros resultados são positivos, o valor de score bateu com o esperado da execução serial em todos os casos mostrando também a corretude do algoritmo.