UNIVERSIDADE FEDERAL DO PARANÁ

VIVIANE DA ROSA SOMMER

TRABALHO 1 - OpenMP

CURITIBA

2022

1. Parte principal do código (Kernel)

A ideia principal do cálculo da Longest Common Subsequence (LCS) é encontrar a maior subsequência comum entre algumas sequências (no caso deste trabalho, 2 sequências), não precisando ocupar posições consecutivas dentro das sequências originais.

A maneira que foi construído esse código se chama programação dinâmica, onde para calcular o próximo valor, existe uma dependência de cálculo que precisa ser previamente computado.

Dessa maneira, criamos uma matriz (scoreMatrix) de tamanho (sizeA + 1) * (sizeB + 1) (onde sizeA é o tamanho da sequência A e sizeB é o tamanho da sequência B) e preenchemos a primeira linha e coluna com zeros.

Para preencher a matriz, seguimos a seguinte lógica:

- Se o char correspondente à linha atual e à coluna atual forem correspondentes, preencha o campo atual adicionando um ao campo da diagonal.
- Caso contrário, pegue o valor máximo da coluna anterior e do elemento de linha anterior para preencher o campo atual.

Segue-se esse passo a passo, até preencher toda a matriz. No final da execução, o campo da última linha com última coluna terá o valor da maior LCS.

2. Estratégia final de paralelização

Como estratégia, foi optado por alterar a leitura das sequências, de maneira a salvar em uma matriz já separada pelo número de threads que serão executadas. Além disso, se considerou que as colunas da segunda sequência seriam quebradas de 100 em 100, para tentar paralelizar esse trecho do código. Assim, ao executar o código, cada thread vai calcular uma submatriz.

Exemplo:

- Sequência: aabbccdd
- Número de threads: 2
- Salvo como:

  [a a b b]

  [c c d d]

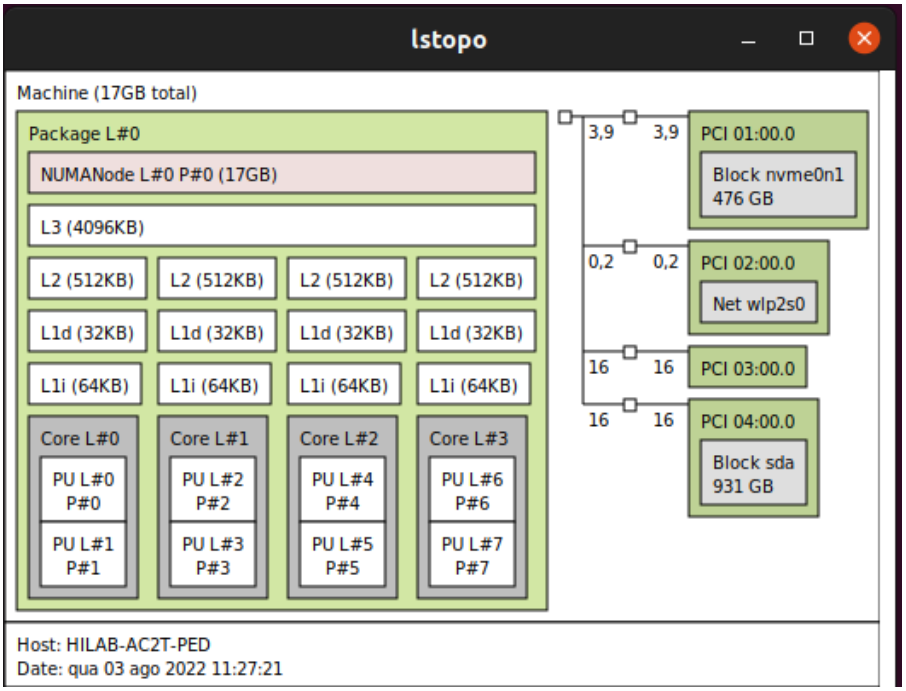
Dessa maneira, ao executar a função LCS, se focou em paralelizar o trecho de laços externos em vez dos dois for's principais, pois como o código é essencialmente feito para ter dependência, não foi encontrada uma maneira de quebrar essa dependência.

3. Metodologia para os experimentos

Para gerar os experimentos, foram criados scripts em python, que faziam chamadas consecutivas do programa, para testar as diferentes entradas e número de threads.
Todas as execuções foram feitas com o Wifi e Bluetooth desligados, apenas o VSCode aberto.

4. Informações da máquina

O trabalho foi executado em uma máquina com as seguintes configurações:



Compilado com: gcc -fopenmp paralelo.c -O3 -lm

Versão do SO:

- PRETTY_NAME="Ubuntu 21.10"
- NAME="Ubuntu"
- VERSION_ID="21.10"
- VERSION="21.10 (Impish Indri)"

Informações do processador

- Arquitetura: x86_64
- Modos operacional da CPU: 32-bit, 64-bit
- Ordem dos bytes: Little Endian
- Address sizes: 43 bits physical, 48 bits virtual
- CPU(s): 8
- Lista de CPU(s) on-line: 0-7
- Thread(s) per núcleo: 2
- Núcleo(s) por soquete: 4
- Soquete(s): 1
- Nó(s) de NUMA: 1
- ID de fornecedor: AuthenticAMD
- Família da CPU: 23
- Modelo: 24
- Nome do modelo: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx
- Step: 1

- Frequency boost: enabled
- CPU MHz: 2328.290
- CPU MHz máx.: 2300,0000
- CPU MHz mín.: 1400,0000
- BogoMIPS: 4591.41
- Virtualização: AMD-V
- cache de L1d: 128 KiB
- cache de L1i: 256 KiB
- cache de L2: 2 MiB
- cache de L3: 4 MiB
- CPU(s) de nó 0 NUMA: 0-7

5. Porcentagem de tempo que o algoritmo demora em trechos não paralelizados

O algoritmo foi dividido nas seguintes funções:

- read_seq - Trecho Serial
- allocateScoreMatrix - Trecho Serial
- initScoreMatrix - Trecho Paralelo
- printMatrix - Trecho Serial - Não usado na execução
- LCS - Trecho Paralelo
- freeScoreMatrix- Trecho Paralelo

Após realizar 20 execuções, com entrada de tamanho 50.000 e 8 threads, a média dos Trechos Serial é de 0.000205 segundos, com desvio padrão de 0.000037 segundos.

N = 50000 Threads = 8	Tempo Total **	Tempo dos Trechos Seriais **	Porcentagem de tempo serial
Execução 1	6.552499	0.000190	0.002899657%
Execução 2	6.553326	0.000208	0,003174%
Execução 3	6.539309	0.000143	0,002180108%
Execução 4	6.561866	0.000141	0,002148779%
Execução 5	6.544938	0.000205	0,003132192%
Execução 6	6.569676	0.000204	0,003105176%
Execução 7	6.549626	0.000211	0,003221558%
Execução 8	6.568928	0.000216	0,003288208%
Execução 9	6.543374	0.000194	0,002964831%
Execução 10	6.559028	0.000242	0,003689571%
Execução 11	6.551455	0.000195	0,002976438%

Execução 12	6.601172	0.000140	0,002120836%
Execução 13	6.558268	0.000225	0,003430784%
Execução 14	6.545858	0.000203	0,003101198%
Execução 15	6.547437	0.000205	0,003130996%
Execução 16	6.598979	0.000211	0,003197467%
Execução 17	6.587281	0.000217	0,003294227%
Execução 18	6.559687	0.000209	0,003186128
Execução 19	6.587935	0.000239	0,003627844%
Execução 20	6.544486	0.000305	0,004660412%

** todos os tempos estão em segundos

6. Tabelas de speedup e eficiência

Eficiência		1 CPU	2 CPUs	4 CPUs	8 CPUs
	N = 20.000	1	0.000000,970	0.000000,960	0.000001,031
	N = 30.000	1	0.000000,975	0.000000,952	0.000001,000
	N = 60.000	1	0.000000,967	0.000000,968	0.000000,992

7. Análise final dos Resultados

Após fazer os testes da tabela de speedup e eficiência, é evidente que o algoritmo paralelo contém erros, e acaba sendo executado em 1 thread + o tempo de overhead para criação das N threads. Por isso, não foi concluída a análise da Lei de Amdahl e Escalabilidade.
Foi encontrado no seguinte artigo com uma possível implementação paralela com OpenMP para o problema de LCS , porém optei por não copiar o código semi pronto e tentar realizar a paralelização por conta própria.

Artigo: Shikder R, Thulasiraman P, Irani P, Hu P. An OpenMP-based tool for finding longest common subsequence in bioinformatics. BMC Res Notes. 2019 Apr 11;12(1):220. doi: 10.1186/s13104-019-4256-6. PMID: 30971295; PMCID: PMC6458724. (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6458724/)