Homework 4

Yongmao Luo yl4893 | Leo Qian yq2292 | Zifan Chen zc2628

1 Introduction

In this homework, we are optimizing the program to calculate the longest common substring (LCS) of two input files. We will use optimization strategies on algorithms, parallelism, and hardware.

2 Baseline

Our baseline implementation is using brute force algorithm with worst-case complexity $O(nm^2)$ where n is the length of longer string, and m is the length of the shorter. However, this is just an upper bound, and in reality, the performance will be better because the upper bound will never be reached, and the performance will be close to $O(nm)$. The results of baseline performance is in Table 1.

3 Dataset

We have 3 different datasets. Each dataset has 2 input files with 40,000 and 60,000 random characters. The first dataset has LCS length of 6, and we call this dataset 1. The second dataset has LCS length of 30,000, and we call this dataset 2. The third dataset has LCS length of 35 and only consists of characters `a` and `b` which results in a lot of common substrings with smaller lengths. We call this dataset 3. We will use these 3 datasets on the following experiments.

4 Optimizations on Algorithms

4.1 KMP

KMP is a famous string-comparing algorithm used to verify if the word is in a string. However, from the theory of this algorithm, even if the word is not fully in the string, we can find

	Baseline	**KMP**	**DP**	**ST**
Dataset 1	3692	7666	2639	1802
Dataset 2	4199	2105	2629	1332
Dataset 3	18670	26124	12101	1982

Table 1: Execution time (ms) of different algorithms and datasets. The compiler flag used is `-O3`.

the longest matching prefix of the word. When finding two strings' longest common substring, we can remove some prefix of the word to generate a new word, then compare it to the string. During the process, we record the length of longest matching prefixes, among which the biggest one is the length of the common longest substring.

The time complexity of KMP is $O(m + n)$. Thus, in this modified KMP algorithm, the time complexity is $O(m(m + n))$, where m is the length of the shorter string, and n is the length of the longer string. Thus, we can save a lot of time compared to baseline.

The actual results are presented in Table 1. For strings having a short common string length, the operation time is even longer than baseline! For strings that have a long LCS length, the operation time is much shorter than baseline.

Having such an interesting result is because when two strings sharing just short common substring, the time complexity to the operation of KMP to find the most common substring will be nearly the same as the baseline, that is, comparing two strings one by one without skipping any element.

4.2 Dynamic Programming

We use a space-optimized dynamic programming algorithm to calculate LCS. The runtime complexity of the algorithm is $O(mn)$. In theory, since we always need around m*n times comparisons, the overall time will be the same.

The actual results for different datasets are presented in Table 1. We can find that for Dataset 1 and 2, the performance is around the same, but for Dataset 3, the performance degrades to around 0.2.

The reason why DP algorithm running on Dataset 3 has such a big performance degrading is that Dataset 3 has many repeated substrings. In DP algorithm, if a match is found, it will have 4 executions rather than 1, thus the performance will degrade.

4.3 Suffix Tree

In order to find out the length of LCS for $str1$ and $str2$, we built an Ukkonen's suffix tree to store the combination of these two strings (i.e. the combination string stored in the suffix tree is $str1 + '\#' + str2 + '\$'$). In this suffix tree the length of LCS for $str1$ and $str2$ is simply the height of the deepest non-leaf node with $\#$ and $\$$ in this tree.

The time complexity to build an Ukkonen's suffix tree is $O(m+n)$, and the time complexity to traverse the tree is also $O(m+n)$ as the number of nodes in the tree is no larger than $2(|M|+|N|)$. Thus, the total time complexity of looking for LCS for $str1$ and $str2$ is $O(m + n)$.

To store all $(n + m)(n + m - 1)/2$ suffixes for the combination string that combines two input strings, suffix tree need space of $O(n + m)$.

	Running Time (ms)
50000 B	4108.1979
100000 B	8045.7151
150000 B	17721.4645
200000 B	31325.6979

Table 2: The running time of suffix tree algorithm for different sum sizes of 2 inputs

The actual results are presented in Tables 1 and 2. The suffix tree can achieve satisfying result compared to Dataset 1, 2, 3. And with sum of two string length increasing, the overall running time increase approxiamtely linearly. The reason why there are some extra time may because the characteristic of input strings.

5 Parallelism

5.1 Vectorization

We only implement vectorization on baseline, because the other algorithms (DP, KMP, and Suffix Tree) doesn't fit into vectorization. We vectorize so that 16 characters can be compared at once. The results of vectorization is presented in Table 3.

5.1.1 Expected Benefits

If 2 strings that need to be compared have many common substrings which are longer than 16, vectorization can speed up the process dramatically. Otherwise, the performance may degrade, even dramatically because we compare many unnecessary characters.

5.1.2 Actual Result

	Baseline
Dataset 1	23824 (0.15)
Dataset 2	23793 (0.17)
Dataset 3	10802 (1.72)

Table 3: Execution time (ms) and speedup of vectorized baseline and datasets. The compiler flag used is `-O3 -msse2`.

5.1.3 Explanation and Discussion

We see here that for strings with very short common substrings, vectorizing the code will incur additional overhead (comparing more characters) and slow down the execution. Only strings with frequent common substrings which length is greater than 16 can apply vectorization of this kind.

5.2 Multi-Threading

The library we used for multi-threading is OpenMP [1]. We use multiple threads to optimize the performance of the for loops in the program so that we can utilize parallelism of the multi-core CPU. OpenMP will run different parts of the loop with different threads in parallel as long as the loops do not have dependencies.

The machine we use have 8 Intel CPU cores, and the performance of algorithms run with 8 threads is recorded in Table 4. (Suffix Tree is not capable for multi-threading.)

	Baseline	KMP	DP
Dataset 1	766 (4.82)	8332 (0.92)	836 (3.16)
Dataset 2	947 (4.43)	6282 (0.34)	856 (3.07)
Dataset 3	3109 (6.01)	9353 (2.79)	2574 (4.70)

Table 4: Execution time (ms) and speedup (in parenthesis) of different algorithms and datasets with multi-threading. The compiler flag used is `-O3 -fopenmp`.

Discussion – We can see that with multi-threading on an 8-core machine, the performance can have a great improvement at about 4x speedup on baseline and DP. We don't yet know why KMP algorithm doesn't have a speedup. Probably this is because of the characteristic of datasets, the undeterminism of the algorithm itself, the compiler optimization, or the computer architecture (See Section 6.2).

Not all algorithms gains same performance improvement from parallelism, and from the data we conclude that the baseline algorithm gains most from multi-threading.

5.3 Putting Together

	Baseline
Dataset 1	3992 (0.92)
Dataset 2	4101 (1.02)
Dataset 3	1902 (9.81)

Table 5: Execution time (ms) and speedup of vectorized and multi-threaded baseline and datasets. The compiler flag used is `-O3 -msse2 -fopenmp`.

We can see that for some situations (Dataset 3) putting multi-threading and vectorization together can have $\sim$10x speedup. However, in other situations, the performance is worse than baseline. We conclude that fully utilizing parallism could have huge performance improvement on performance, and vectorization can only be effective for some scenarios and should be careful when dealing with.

6 Hardware Optimizations

6.1 Cache

We use Pin Tool [2] to run cache simulator on our algorithms. We used strings of length 400 and 600, and adjusted different parameters for cache line size, associativity, and size. We use cache miss rate as the objective. We do not run Suffix Tree on the simulator, because the simulation takes too long for unknown reason. Results are recorded in Tables 6, 7, and 8.

	Baseline	KMP	DP
32 B	0.46	0.38	0.72
64 B	0.26	0.21	0.4
128 B	0.15	0.12	0.24
256 B	0.09	0.08	0.15
512 B	0.07	0.05	0.10
1024 B	0.06	0.05	0.09
2048 B	0.11	0.08	0.15

Table 6: Cache miss rate (%) of different algorithms and cache line size on cache simulator. The cache associativity is 8 and cache size is 32 KB.

	Baseline	KMP	DP
1	0.34	0.28	0.53
2	0.26	0.22	0.41
4	0.26	0.21	0.41
8	0.26	0.21	0.40
16	0.26	0.21	0.40
32	0.26	0.21	0.40

Table 7: Cache miss rate (%) of different algorithms and cache associativity on cache simulator. The cache line size is 64 B and cache size is 32 KB.

	Baseline	KMP	DP
32 KB	0.26	0.21	0.4
64 KB	0.24	0.20	0.38
128 KB	0.22	0.18	0.34
256 KB	0.22	0.18	0.34

Table 8: Cache miss rate (%) of different algorithms and cache size on cache simulator. The cache line size is 64 B and associativity is 8-way.

	Baseline	KMP	DP
Optimized	0.02	0.02	0.04

Table 9: Cache miss rate (%) of different algorithms on optimized cache (1024 B cache block size, 128 KB capacity, 8-way associative) on cache simulator.

Finally, we choose the best parameter of each, and run the simulator on the optimal parameters. The cache miss rates reported are in Table 9.

Discussion – From Table 6, we can find that with cache line size increasing from 32 B to 1024 B, the cache miss rate of all three algorithms decreases. The reason is that our algorithms have comparatively good spacial locality. However, if we set cache line size to 2KB, the miss rate increases because the cache will only have 2 sets, which will largely increase cache replacement miss.

From Table 7, we observe that with associativity increasing, the cache miss rate decreases. This is mainly because it decreases the cache mapping miss rate.

From table 8, we observe that with cache size increasing, the cache miss rate decreases. This is because increasing cache size will increase cache sets, which will reduce cache replacement miss rate.

6.2 Architecture

We ran the same algorithms on Apple M1 microarchitecture and obtained different results in Table 10 and 11.

	Baseline	**KMP**	**DP**
Dataset 1	2365 (1.56)	5071 (1.52)	1588 (1.66)
Dataset 2	2602 (1.61)	1475 (1.43)	1594 (1.65)
Dataset 3	10414 (1.79)	15013 (1.74)	7774 (1.56)

Table 10: Execution time (ms) of different algorithms and datasets on 8-core Apple M1. The compiler flag used is `-O3`.

	Baseline	**KMP**	**DP**
Dataset 1	691 (3.42)	1221 (4.15)	1768 (0.90)
Dataset 2	735 (3.54)	1259 (1.17)	1716 (0.93)
Dataset 3	2335 (4.46)	3203 (4.69)	3561 (2.18)

Table 11: Execution time (ms) and speedup (in parenthesis) of different algorithms and datasets with multi-threading on 8-core Apple M1. The compiler flag used is `-O3 -fopenmp`.

Discussion – Based on Table 1 and 10, we find that each algorithm will be sped up for around 1.6 times. The reason is because the per-core performance of Apple M1 is higher than Intel CPU core.

More importantly, in theory with multi-threading, the KMP algorithm will be sped up, and we can find the same result as the theory inferring. However, for multi-threading implementation of DP algorithm, the performance degrades for Dataset 1 and Dataset 2, which is also strange because it is against the theory.

7 Conclusion

The most speedup we get is from parallelism, especially multi-threading. In specific scenarios, vectorization can also be applied to reach additional 2x speedup. The maximum speedup we are able to reach from parallelism is 9.81x with 8 threads on an Intel machine. Not all algorithms are suitable for parallelism, and we find that the naïve implementation gains the most speedup from parallelism.

Different algorithms can apply to different scenarios. We compared different algorithms under different datasets. Suffix Tree has the lowest time complexity, but it's hard to parallelize. With parallelism, baseline and DP can achieve better performance than Suffix Tree.

We also argue that using a 8-way associative and 128KB cache with 1024 B blocks, we can reach minimum cache miss rate in the simulator, and we expect that in reality this could also improve performance by a lot.

On different architectures, since the environments are different, we couldn't reach a definitive conclusion, but we find that the performance will be different when run on an Intel machine and on Apple machine. This deserves further investigation.

Logs

- Nov. 23rd: Yongmao Luo and Leo Qian discussed the architectural optimization method about the homework.
- Nov. 27th: Yongmao Luo and Leo Qian discussed the algorithm optimization, and realized the baseline and kmp.
- Dec. 2nd: Yongmao Luo and Leo Qian tested the efficiency of different algorithms and discussed the fixes and optimization of algorithms to finalize the algorithm used in the homework.
- Dec. 3rd: Yongmao Luo and Leo Qian created different test cases to verify the characteristics of algorithms, discussed the way to architecturally optimize the dynamic programming algorithm and wrote the report.
- Dec. 6th: All discussed different algorithm and optimization methods. Zifan Chen implement the algorithm via Suffix Tree.
- Dec. 11th: Leo and Yongmao worked on vectorization. Zifan worked on Suffix Tree.

References

[1] L. Dagum and R. Menon, OpenMP: An industry standard api for shared-memory programming, *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. DOI: 10.1109/99.660313.

[2] Pin - a dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.