Smith-Waterman Profiling Sophie Liu_FA21

Objective: In this assignment, the goal is to find the length of the longest common substring of the two strings. Assume the strings are hundreds of thousands of characters, such as DNA chains.

Methodology: There are multiple ways to find the common substrings. I chose to go by Smith-Waterman (SW) algorithm, a common DNA profiling method which finds the longest common subsequence. Smith-Waterman algorithm can tolerate local mismatches and gaps. It finds the longest common subsequence by keeping track of the highest scores, where mismatches and gaps are penalized, and matches are rewarded. The reason for SWA, rather than other algorithms which find the common substring, is because in DNA sequencing, finding the longest subsequence is more practical.

The following methods showed performance speed up compared to the naïve Smith-Waterman algorithm: thread parallelization, critical/atomic synchronization, and increasing memory size when the strings get too large (≥ 100k). The method expected to show speed up but did not were data prefetching and SIMD. It might be due to the complicated data accessing pattern, which prevents the compiler from efficiently executing prefetching and SIMD. The method having an ambiguous effect is thread affinity (restricting the access to a few threads when there are plenty available). It seems that setting thread affinity has some performance speedup when there are a lot of simultaneous CPU tasks. But this is not prominent when the SWProfiling.c was the only file running on the computer.

In addition, cache simulation is performed to understand how different caching factors affect the miss rate. Factors that affect missing rates are data size (string length) and L1 cache size. Factors that did not help are associativity and L2 cache size.

Execution: to compile, make sure openmp is available.

General compile: gcc -O2 ./≪filename≫ -fopenmp -g3
Specify number of threads: OMP_NUM_THREADS=≪threadnumber≫ ./a.out
Enable prefetch: icc ./≪filename≫ -qopenmp -O2 -qopt-prefetch
Enable aggressive SIMD: gcc ./≪filename≫ -fopenmp -ftree-vectorize -ftree-slp-vectorize -ffast-math -funsafe-loop-optimizations -ftree-loop-if-convert-stores -march=native -mtune=native -Ofast
Enable thread affinity: export GOMP_CPU_AFFINITY=0-3
Export OMP_PROC_BIND=true

Results

Figure 1 shows the runtime comparison between naïve SW, parallel critical SW and parallel atomic SW. The naïve SW is faster when the data size (string length) is smaller. This is because smaller data sizes cannot overcome the overhead of parallelism. The second method implemented was parallel SW with the critical section to assign the max scores. However, this method does not show any speedup compared to the naïve, which is speculated to be due to the synchronization overhead. The third method implemented was parallel SW with atomic section to assign the max scores. This method only shows speed up when the data size (string length) is large (≥ 10,000). The three methods were initially tried on a 4-thread computer for 100,000 data size (string length), but the performances were not as expected. A closer inspection shows that the memory (DRAM) was drained. An alternative is to tile the scoring matrix to save up the memory footprint of the algorithm. However, due to the time constraints and the complexity of the algorithm (banding/striping), I am not able to test this hypothesis. The eventual solution was to move the code to a larger machine with 16 threads available. As figure 2 shows, with more memory space, the advantage of parallel atomic SW on large data size (100,000 string length) is prominent compared to naïve SW. The performance seems to have a proportional scale-up for 2,4, and 8 threads. As the thread number increases beyond (16 threads), the performance doesn't seem to scale up. This observation can be expected to be improved if the data size (string length) is larger than 100,000, which manifests the full potential of thread parallelism.

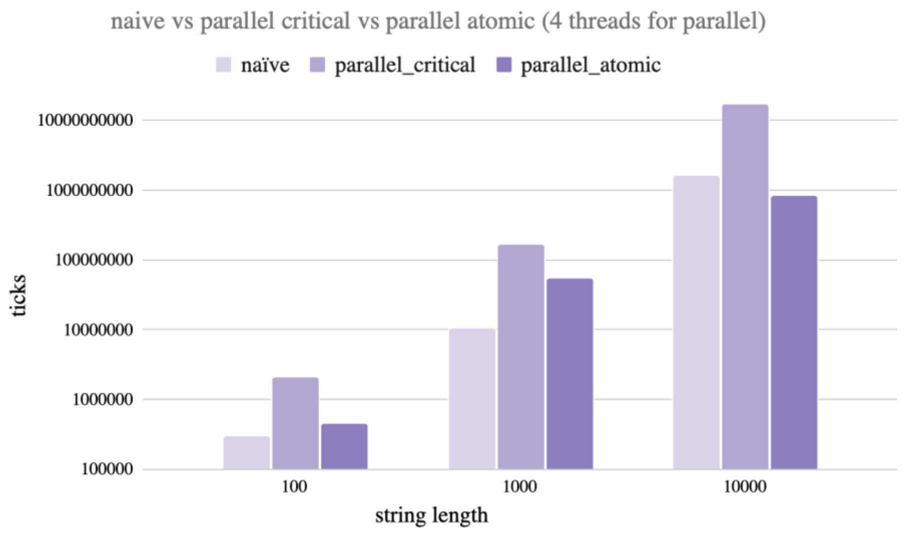

Figure 1. Runtime comparison between naïve, parallel critical and parallel atomic

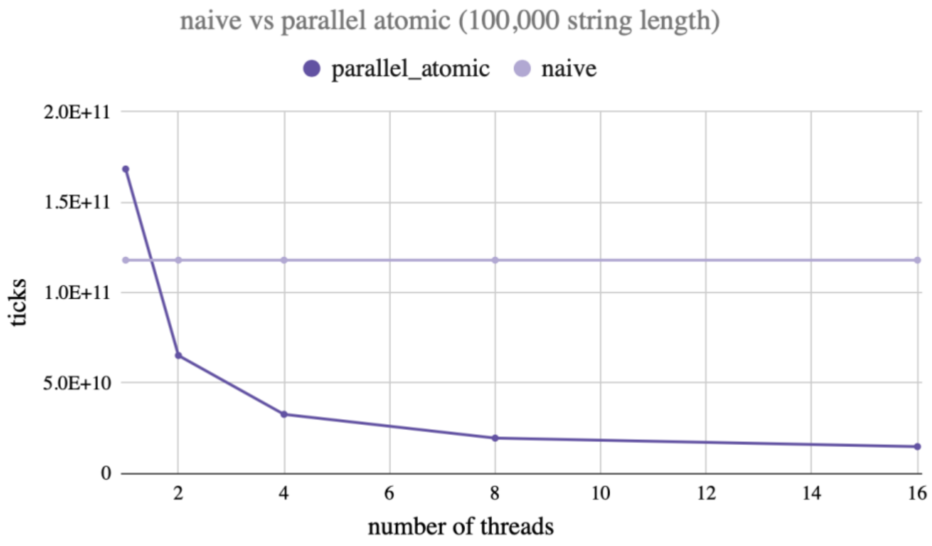

Figure 2. Effect of increasing threads in parallel atomic algorithm

Cache simulator

Different cache factors were simulated through the pin tool. Only small sizes (less than 500 string length) were tested due to simulation overheads, which made testing larger data sizes impractical. Figure 3 shows that miss rate drops as cache size (L1) increases, but plateaus after cache reaches a certain size. This is expected. Once all the data fits into the cache, increasing the cache size further does not offer any benefit. In addition, the miss rate was calculated for different data sizes (string length) with the same cache size. Figure 4 shows that the miss rate initially increases, but soon decreases as data size becomes larger for a given cache size. This result is consistent after repeating the simulation, which suggests that SW reuses data in the cache for larger data size.

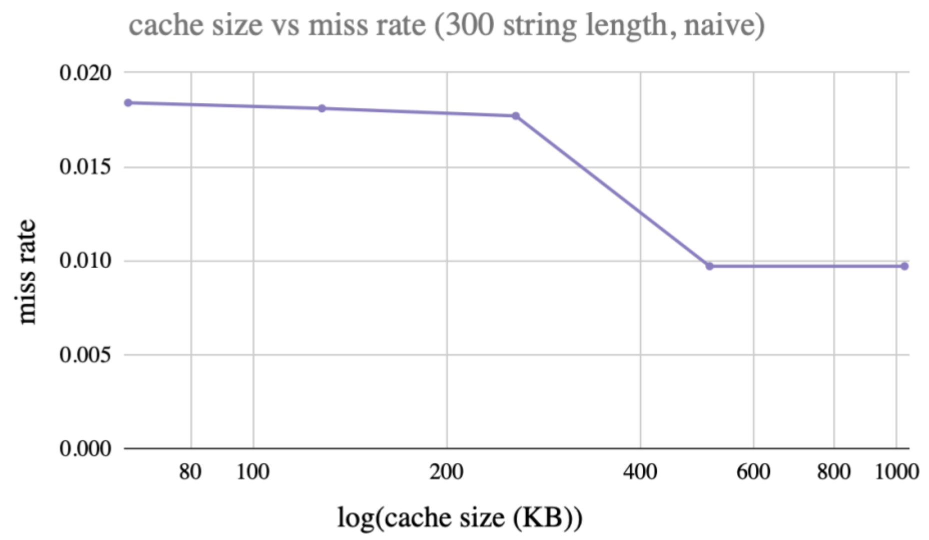

Figure 3. Effect of different cache size on miss rate (string length is 300)

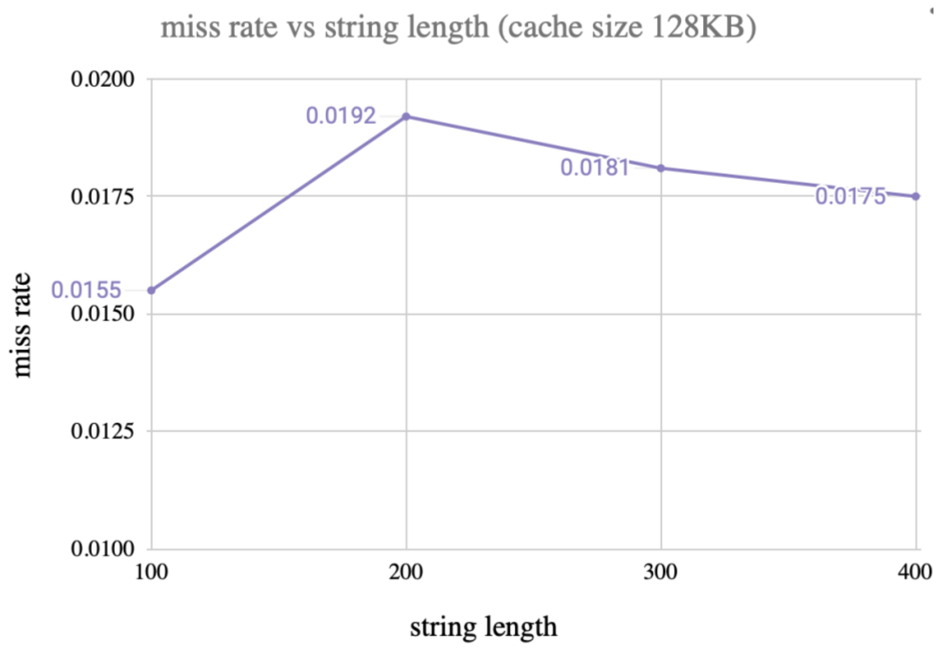

Figure 4. the effect of string length on miss rate for a defined cache size

Discussion for other expected optimizations but had no improvement
Prefetch – Data prefetching was implemented through the command line options (compiler flags). The results did not suggest that prefetching speeds up any of the SW naïve, SW parallel critical and SW parallel atomic. Further inspection on assembly code suggests that the compiler only generated very few prefetch instructions. This can be explained as the complex data accessing pattern of SW algorithm hinders the compiler to implement efficient data prefetching.

SIMD – an aggressive SIMD was implemented through the command line options (compiler flags). The results did not suggest it speeds up any of the SW naïve, SW parallel critical and SW parallel atomic. Further inspection on assembly code suggests that the compiler does not generate more SIMD instructions than without using aggressive SIMD commands. This can be explained as the compiler is not able to recognize SIMD opportunities in the SW algorithm, which is possibly due to the complex data accessing pattern.

Thread affinity – Thread affinity was enabled through the command line. The results were ambiguous in the way that it heavily depends on the simultaneous CPU tasks. It was briefly seen that when there are many CPU tasks running aside of SW, thread affinity speeds up SW. But it is not so clear since the concurrent CPU tasks are very dynamic and are hard to track down.

L2 - L2 cache was modified on top of the optimal L1 (the largest L1) to inspect whether L2 can further improve the cache performance. The result shows that L2 does not improve any of the cache performance metrics given an optimal L1. This proves that if the L1 size is sufficiently large, L2 cache size becomes irrelevant.

Future Work
SW algorithm has been studied in literature at different architectural levels. The most relevant further improvement on top of the current implementation is to implement the memory tilling so that the algorithm can handle longer data size in smaller computer machines. There are also other cache metrics that weren't fully investigated, such as associativity, capacity, etc, which can be expected to be tested.

https://link.springer.com/article/10.1007/s12539-021-00473-0