Project 4 Report

Hisen Zhang <zhangz29>

Experimental results show your implementation's performance (both encoding and query performance). When measuring the performance, do not count the time of loading file to memory and writing file to SSD. The performance results must contain:

1. encoding speed performance under different number of threads,
2. single data item search speed performance of your vanilla baseline, dictionary without using SIMD, and dictionary with SIMD,
3. prefix scan speed performance of your vanilla baseline, dictionary without using SIMD, and dictionary with SIMD.

The test was performed on the platform described below:

OS Ubuntu 18.04.5 LTS
Memory 23.3 GiB
Processor AMD Ryzen73800X8-core processor×8
Graphics SVGA3D;build:RELEASE;LLVM;
GNOME 3.28.2
OS type 64-bit
Virtualization Oracle
Disk 42.0 GB

To minimize the disk IO bottleneck, we created a file system in memory to hold the input and output files:

```
mkdir /mnt/tmp
mount -t tmpfs -o size=2g tmpfs /mnt/tmp
```

Three methods mentioned below are compiled with following commands:

vanilla: `g++ test_vanilla.cpp -o build/test_vanilla -O2`
dictionary: `g++ test_main.cpp trie.cpp -o build/test_main -O2 -pthread`
dictionary_SIMD: `g++ test_main.cpp trie.cpp -o build/test_main_SIMD -DSIMD -mavx2 -O2 -pthread`

The value of percentage boost = (old-new) / new * 100%

Measurement

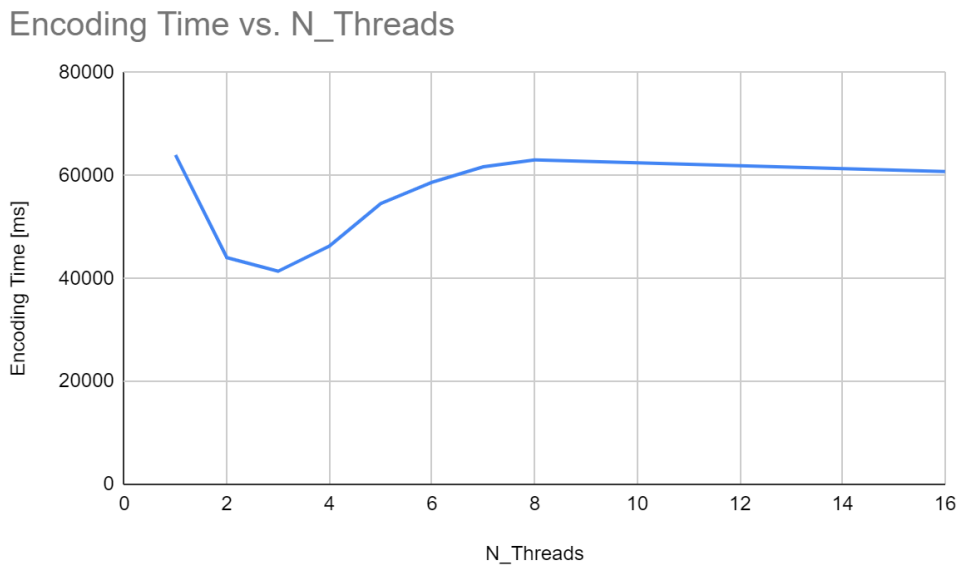

Figure 1. Encoding Time vs. Number of Threads

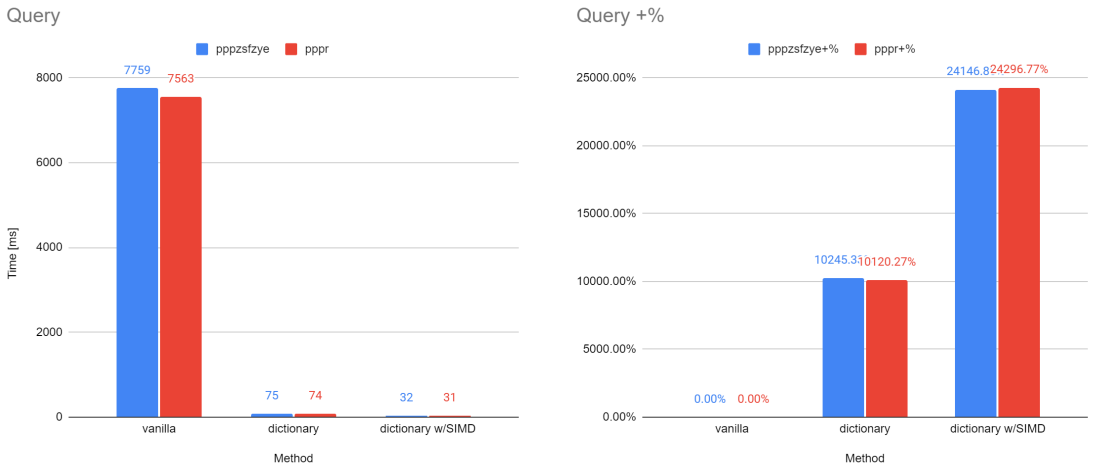

Figure 2. Query Time in ms (left) and Performance Boost Relative to Vanilla (right)

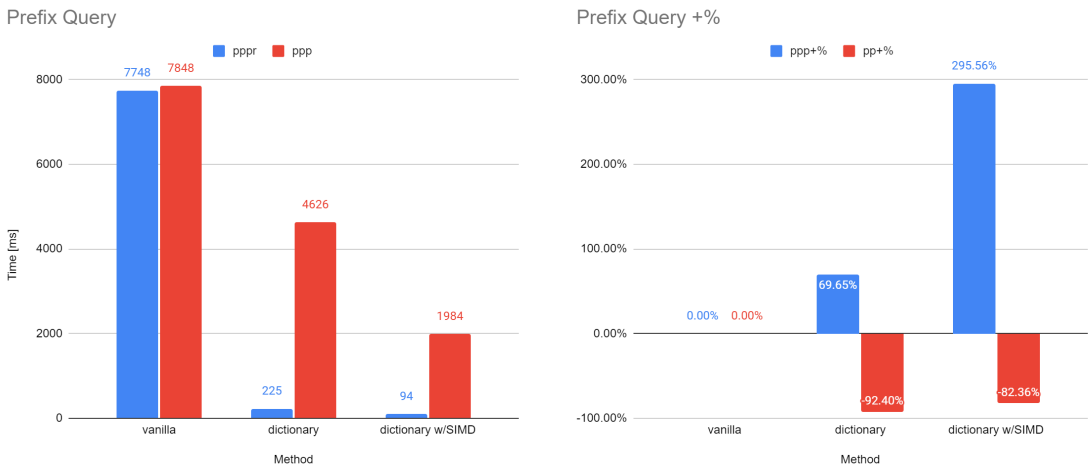

Figure 3. Prefix Query Time in ms (left) and Performance Boost Relative to Vanilla (right)

Analysis

In Figure 1, the time for encoding drops from 1 thread to 3 threads. Then as the number of threads grows, the benefit decreases gradually. While we expect the peak performance at 8 threads (hardware concurrency), this behavior can be explained by the implementation.

The main thread reads CHUNK_SIZE lines (1000 by default) at a time into a vector, dispatching the encoding task to the thread pool. Because the file is in ramdisk, the parsing is very fast, and the encoding is the bottleneck. At thread number equals 3, the parsing rate is approximately the same as the encoding rate. For fewer threads, the encoding worker consumes tasks slower than the parsing rate, leaving data in the queue. For more threads, workers compete for tasks, leaving most of the threads idling, and the overhead of mutex, as a result, becomes significant. Therefore, the number of threads to reach peak boost is a function of the data IO rate and the CHUNK_SIZE.

In Figure 2, we see a major performance boost by the dictionary codec. Vanilla implementation reads files one line at a time, keeping the indices of matching elements. The dictionary codec works the same way, but because the encoded vector is much shorter and kept in the memory already, the search takes only 10% of the time that baseline would consume, regardless of key length. SIMD implementation further halves the time by looking at 8 integer elements at a time, which is only feasible on the encoded integer rather than the raw string inputs. Therefore we can assert that dictionary codec accelerated search to a significant level.

Figure 3 compares the performance of prefix search on different methods. Generally, the vanilla linear search has a linear time complexity of O(n) regardless of the key length. The dictionary codec with the prefix tree, however, is heavily influenced by the key length. This is due to the implementation: the program first looks for all keys with the given prefix, then performs SIMD

linear search in the encoded entries one by one. A shorter key generally means more found matches, slowing down the query. Therefore, the program favors a longer key prefix. In some extreme cases (e.g., prefix “pp”), the codec performs worse than the baseline. While this can be done by limiting the minimum key length, a better SIMD (AVX) prefix search algorithm may overcome this issue.