Data Structure In Real World: Part III Report

Wuwei Yuan, 2020011000

1. Introduction

In this project, we present some variants of bloom filter and evaluate the performance on the given dataset by ourselves.

2. Approaches

We implement the ”standard” bloom filter (std) and some cache-efficient variants (blo[1], pat[1, 1]) of standard bloom filter described in the paper “Cache-, Hash- and Space-Efficient Bloom Filters”. For ‘std’ and ‘blo[1]’, we present its multi-threading variant by atomic operations. We also present two multi-threading variants of ‘pat[1, 1]’ using atomic operations and mutex locks.

The dataset has $n = 10^7$ insertions. Among all implementations, we set $m = 2^{27}$ to make division faster. We also set $k = \frac{m}{n} \ln 2$ according to the paper.

3. Optimizations

3.1. Workload Dispatcher

The preallocated workload is not balanced, so we propose some dispatch method.

The basic one distribute each operation to the task list of thread $hash(key)\%num\_threads$. However, the data conforms the Zipf’s law, and some keys will have more occurence than others. This makes some task list much longer than others.

Considering the data distribution, we try to allocate the tasks dynamically. We first separate the keys into $t = 2^{20}$ buckets. While iterating the full task list, we first distribute each task to bucket $hash(key)\%t$. When a bucket is put into the first task, we select the thread with shortest task list at that time as the corresponding thread for that bucket. Although this strategy can make the length of task lists of each threads have nearly the same length, the execution time of each thread is not balanced. Some threads only have a few distinct keys, and the memory access latency will be much lower due to better time locality.

Now we take account into the memory access latency. We still separate the keys into t buckets in the same way discribed above. In the case of choosing the execution thread, we add weights to the number of threads in each bucket. Numerically, when a thread has buckets with $a_1, \ldots, a_l$ tasks, the weighted length of that task list is choosen to be $s = \sum_{i=1}^{l} a_i^{0.9}$. Finally we also choose the thread with minimal weighted length.

3.2. Thread Scheduling

When the number of threads is large, the OS always allocate some threads to the same core and result in poor efficiency. Instead, we schedule the threads manually to ensure that each thread is executing on the different core.

4. Experiments

The experiment results are shown below.

We also evaluate the performances of different workload dispatchers. The advanced dispatcher separate the task list to more balanced length, and the more advanced dispatchse separate to more balanced execution time.

5. Future Works

The workload dispatch algorithm cannot evenly distribute the dataset, and we can still make some improvements here. Also the computer works as a black box to me, and we need to learn more about computer architecture to understand it better.

6. Acknowledgements

Thanks Yixuan Xu, Yan Dai and Renfei Zhou for discussion and effective advices. We also thank Prof. Zhang and the TAs Yipeng Liu and Hengrui Huang for providing powerful servers and helps.