Comparing Intel TBB `concurrent_hash_map` and `libcuckoo`

Manu Goyal, Dave Andersen, Michael Kaminsky

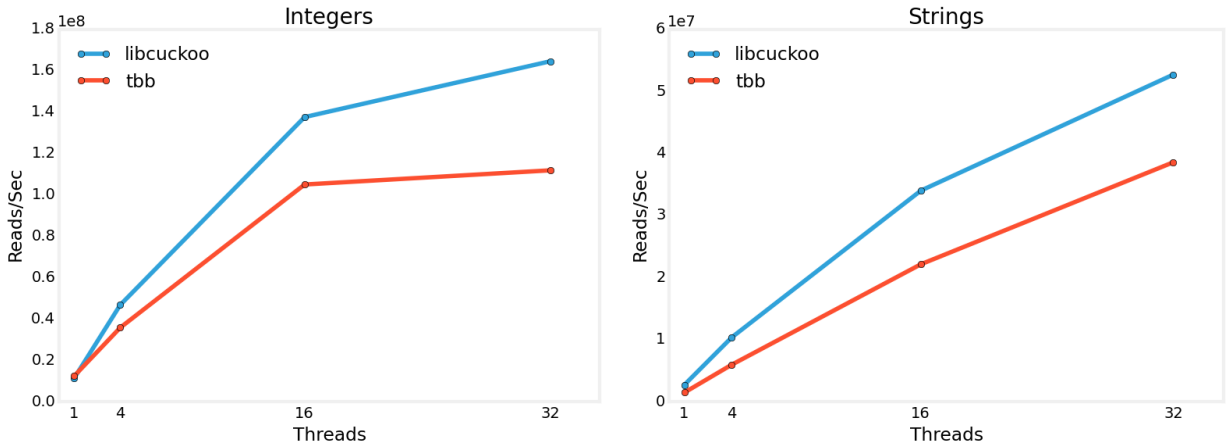
October 29, 2015

Overview

In this benchmark, we compare `libcuckoo`, our high-performance, memory-efficient hash table, with the Intel Thread Building Blocks `concurrent_hash_map`. We compare the performance of the two tables across different types of workloads and different numbers of cores, and also compare the memory usage for different table sizes. We ran the benchmarks on a machine with 36 Intel Xenon 2.9 GHz cores and 60GB memory. The cores were split evenly into two NUMA nodes.

Pure Read

Our read benchmark fills a table up to 90% of its allocated capacity, then concurrently runs reads for data that is in the table and data that isn't. It counts the number of reads executed over 10 seconds. Figure 1 compares the read throughput of the two tables with integer and string keys. For both types of keys, `libcuckoo` outperforms `concurrent_hash_map`, with the difference getting slightly larger as we increase the number of threads. With 32 threads, `libcuckoo` outperforms `concurrent_hash_map` by 47% for integers, and 37% for string keys.

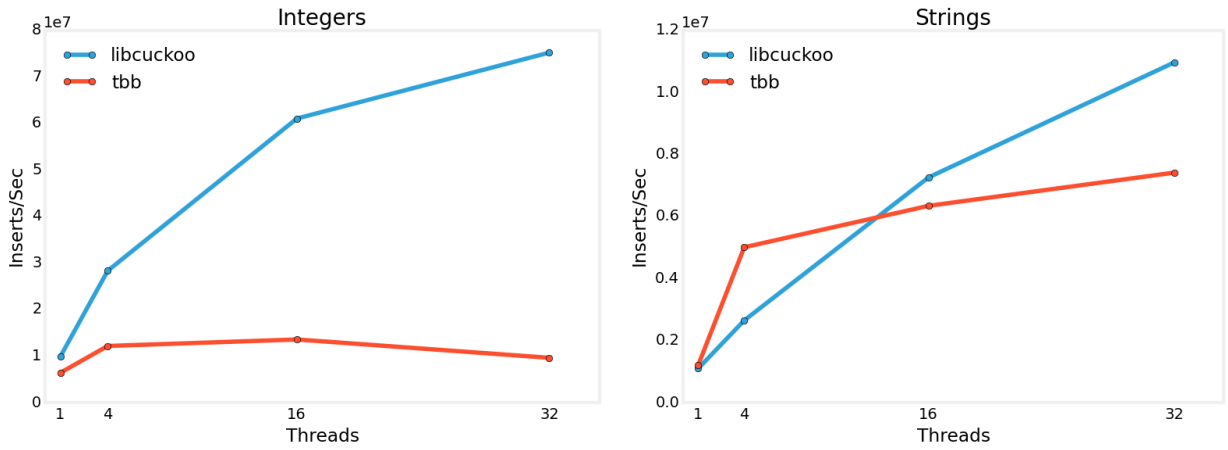


(a) (b)

Figure 1: Pure read throughput for integer and string keys

Pure Insert

Our insert benchmark measures the time taken to fill up a table from 0% to 90% of its allocated capacity. Figure 2 compares the insert throughput with integer and string keys. For integers, `libcuckoo` greatly outperforms `concurrent_hash_map`, by over 680%, and for strings, it outperforms `concurrent_hash_map` by 50%. We suspect that TBB's low performance on integers was due to the fact that it doesn't deal well with multiple NUMA clusters. This would explain the poor scaling with large numbers of threads on different NUMA clusters. With strings, since the cost of copying strings likely dominates the runtime, this effect is less apparent.



(a) (b)

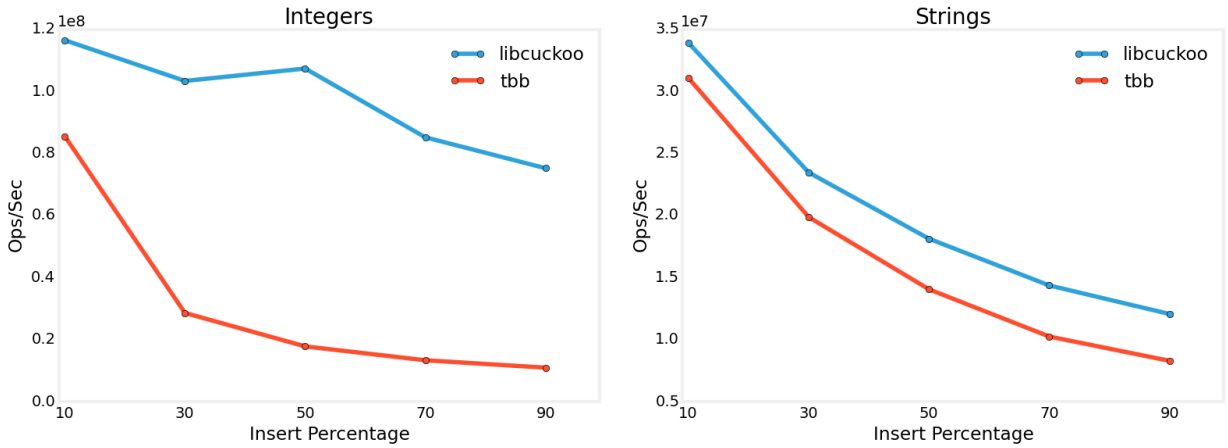
Figure 2: Pure insert throughput for integer and string keys

Mixed Workload

Our mixed benchmark runs a mixed workload of inserts and reads at a configurable ratio, and measures the time and number of operations taken to fill up the table from 0% to 90% of its allocated capacity. Figure 3 compares the performance of the two tables at different ratios of inserts (all with 32 threads), with `libcuckoo` doing better with both integer and string keys. We see again that the difference between the two tables is much greater at higher insert percentages compared to lower percentages (588% compared to 36%, respectively), because the difference between `libcuckoo` and `concurrent_hash_map` is more pronounced for inserts than it is for reads.

Memory Usage

Finally, we compare the approximate memory usage of the two tables. While not a completely accurate measure of the amount of memory used by each table, we measured the maximum resident set size as determined by Ubuntu's `time` command for the insert benchmark. For integers, `libcuckoo` scales far better than `concurrent_hash_map`, using 72% less memory than `concurrent_hash_map` with the largest table.



(a) (b)

Figure 3: Mixed read-insert throughput for integer and string keys

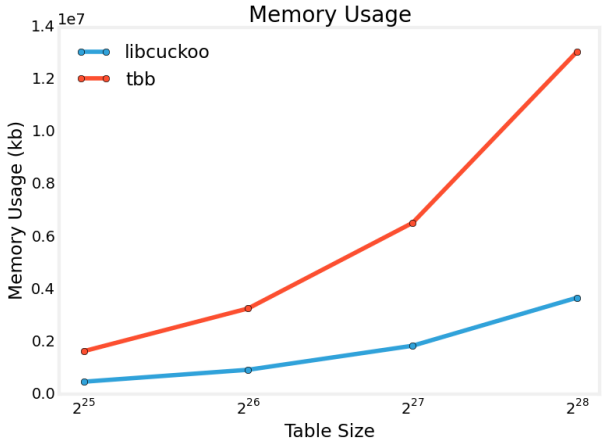


Figure 4: Memory usage for insert benchmark with integer keys. Table size is the number of elements each table has capacity for

Conclusion

`libcuckoo` has a number of features that cause it to perform better and use less memory than `concurrent_hash_map`. `libcuckoo` stores data in a cache-optimized form and avoids false sharing between CPU's, which allow it to scale inserts and reads very well to a large number of CPU's with low memory overhead. Furthermore, the cuckoo hashing algorithm lets it achieve very high table load factors before needing to expand, which significantly reduces memory usage.