CS250B Homework 1: Optimizing Sorting

Due November 25

In this homework, you will attempt to apply what we discussed during the lectures to implement the best possible sorting function. You are free to try whatever approach you want, while being unsure of what machine your code will eventually be deployed to. (So, a typical development scenario)

The provided code includes two complete implementations, one using std::sort, and one using quicksort. The data type to sort is a key-value pair. The provided version uses a 64-bit key and 32-bit value, but this may change (but not so much as will break the quicksort implementation. Meaning, key and value will be 64 bits large at maximum).

Furthermore, while the existing two sorting implementations are both single-threaded, you are free to use more threads, as specified by the "threads" argument in the user_sort function.

Deployment System

The only assurance given about the deployment/evaluation system is that it is an x86 server new enough to support AVX2. (So, perhaps not the best idea to develop on an ARM machine.) There is no assurance on DRAM performance or details about the cache hierarchy. Although it is likely it will also share some typical characteristics such as the size of the L1 cache.

Recommended Approaches

There are three things you can probably optimize: the algorithm, cache accesses, and parallelism.

As for the algorithm, the simplest approach you can do is probably mergesort, since it has a nicely predictable access pattern while still being O(nlogn) on average.

As for cache access optimizations, you will probably want to use something similar to the lazy k-merger covered during class. But since developing a generic and flexible lazy k-merger is complicated, you can probably fix a reasonably large size of K without losing too much performance. (Remember the performance comparisons between van Emde Boas trees and a simple B-tree)

As for parallelism, you will want to use many threads. The best way to do this will probably be to partition the array into N blocks and distribute work across N threads, and then use one thread to merge them into a single sorted array. Note that more threads may not always help, if each thread is requesting too much bandwidth from DRAM such that the DRAM becomes a bottleneck with many threads.

Furthermore, you can try to use AVX for sorting as well, as hinted during the AVX section. This likely non-trivial, especially considering that you will need to sort key-value pairs.

What to Submit

Only submit sort.cpp

Caveats and Grading

This project is new, and I don't (yet) have clear knowledge about the upper limit on the performance we can achieve. It may simply be N*X, where N is the number of threads and X is the throughput of the std::sort implementation. Or, it may become much larger thanks to better cache optimization and (potentially) AVX.

Grading will be done via an anonymized derby, meaning we will compare the performance of all submissions (and some of my reference implementations) with names hidden. But don't worry, as long as there is reasonably effort the grades will be favorable. The derby is less for grading on a curve, and more for comparing the effects of various optimization attempts using concrete implementations.