CSE 330 – Project 2 – Devyash Lodha

Map and Reduce

# Data Structures During Reduce

During the reduce phase of my map-reduce program, I used a 256-ary trie, just like during the map phase. The reduce workers read the data from the file into local tries for each thread. The map workers read the data from incoming map files, and push their data on their local 256-ary tries. The reduce workers push the trie data into files thereafter. The main thread creates a trie from the data from the files, and outputs the histogram of words.

# Status of Program

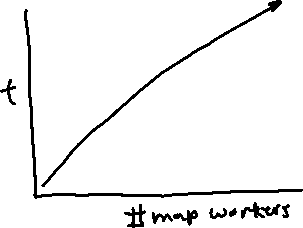
I noticed that when handling very large input with possibly long words, the program crashes. This is likely because I’m handling the words with a recursive algorithm. This should be relatively easy to fix, however. I would need to change the trie generating algorithm from recursive to iterative, and from a recursive tree traversal to iterative depth-first-search or breadth-first-search.

# Strategy of Experimentation

I experimented by running the program on an 8-core system. I played around with the map and reduce worker count. Unfortunately, I’m having the issue that if I turn up my map worker count too high, I encounter segmentation faults.

What I saw is that as I increased the number of cores, the performance of my program reduced. This is because the histogram algorithm I’m using is incredibly efficient (). I’m constantly rebuilding this structure, which puts a significant strain on the memory allocator.

While I couldn’t get hard numbers, here’s what I noticed:



The number of reduce workers did not make a difference because of the round-robin algorithm I used.

# Performance Testing

When , the program required in total for world128.txt.

When the program required in total for world128.txt.

When , the program required in total for world128.txt.

When , the program required in total for world128.txt.

The impact I noticed is that by increasing the number of threads doing work, I reduce the performance significantly. This is very likely because my data structure is multithreaded, and thus, can be written to from multiple threads simultaneously without causing memory corruption.

Very little impact was caused because of equal vs. skewed, because after threads read the files, the data structures were fast enough that input size was a blip compared to the rest of the overheads.

# Interactions & External References

I did this entire project by myself. I used a esoteric data structure that pretty much none of the tutors had even heard of, but which yielded blazing-fast performance.

The main external reference I used was the manual pages from manpages-dev. This included the manual pages for:

* pthreads
* pthread\_create
* sem\_init
* sem\_wait
* sem\_post

I also did look at the C++ reference to learn how to use function references instead of function pointers.