Analysis

General Strategy

My general strategy at the beginning of part two was that I would parallelize a small function that was being run many times within a loop. This function simply checked if the ngram had a newline. If it did have a newline, it would return 1, and if it didn't it would return 0. I thought that although this function had very little computational power, it was one of the most-run functions. After I finished threading this function, I ran it through the word list, and it took about 15 *minutes* to complete. After a brief reflection, I concluded that this would not suffice.

I ended up making a function that would handle each line that was returned from the file. My design was fairly intricate. Each thread would be responsible for a single line from the file. A two-dimensional array was needed to hold each file line that would later be used for each respective thread. Each char array in this array, along with the size of each ngram desired, was then copied onto a structure that held the arguments to each thread. This structure was then passed onto each thread's function. This function would then return a two-dimensional array of every ngram in the char array. The returned two-dimensional array then became a three-dimensional array outside of the threads which contained each thread's ngrams. These ngrams were then added to a hash table, which were counted for frequency, then sorted by frequency and alphabetical order, then printed.

I tested this program bit-by-bit because it seemed like any small change to the program would cause it to have a memory leak. In addition to this, if there was an error in the code, I would start commenting out parts of it to see where the error occurred. In regard to changes to part one, most of the code changed. Not only did part 1 not have strings ending in null bytes, the ngrams were corrupted they and weren't sorted alphabetically. This resulted in output that was gibberish and incorrect. I fixed these issues, and in addition to threading the program, I simplified my code by placing a large part of it in a separate function and calling it from the main function.

Benchmarking Results

The benchmarking results were consistent. Each result was no more than 1.5 seconds away from each other. In addition, the speedup across each number of threads was consistent. For each ngram size, I saw speedup continue throughout each thread except the eighth thread. I believe that this might be because the machine I was shelled into has less than 8 cores. This resulted in the 8 threads taking longer to execute. As the ngram size increased, the speedup decreased slightly, only by .03 points at most. Based on my observations throughout the project, the factors that could affect my program's performance were the number of threads, the ngram size, where I thread my function, and the amount of code overhead. The more code overhead in the program, especially inside multiple loops, the more my program would suffer in its performance.

Impact and Personal Reflection

To my knowledge, deep learning models are highly reliance on efficient code. Based on my results, parallel computing could aid in the creation of more sophisticated models based on the speedup that the practice offers. Pthread is limited to only 825 threads, and each *pthread_create()* creates just one thread. Other parallel libraries can have more threads and are scalable to large super computers. One concept I struggled with was malloc. I didn't quite grasp why I needed to touch the heap at all.

During this project however, I learned that it was the only way to be able to make a local variable and return it from inside a threaded function. This is because the function stack is destroyed after it is returned, but the heap does not get destroyed. This was the most valuable concept I learned. Another concept I struggled with was memory leaks. These pesky errors were made only slightly easier by valgrind, and they were difficult to fix. The biggest thing I learned from this problem was that every malloc() needs a free(), unless using realloc(), where only one free is needed.

Benchmarking

N = 2 (sec)	1	2	4	8
Run #1	30.77	21.36	18.70	25.14
Run #2	30.56	21.59	18.76	23.66
Run #3	30.37	21.34	18.63	23.63
Average	30.57	21.43	18.70	24.14

N = 3 (sec)	1	2	4	8
Run #1	30.84	22.34	19.54	24.63
Run #2	31.19	22.08	19.43	26.02
Run #3	31.75	22.28	19.38	24.76
Average	31.26	22.23	19.45	25.14

N = 4 (sec)	1	2	4	8
Run #1	33.07	24.04	21.10	27.39
Run #2	33.36	23.99	20.91	27.61
Run #3	33.03	23.96	21.07	27.02
Average	33.15	24.00	21.03	27.34

Speedup	2	4	8
Bigrams	1.43	1.63	1.27
Trigrams	1.41	1.61	1.24
Quadrams	1.38	1.58	1.21

Sources

injoy. (2012, September 12). *What is the maximum number of threads that pthread_create can create?* Stack Overflow. Retrieved December 6, 2022, from https://stackoverflow.com/questions/12387828/what-is-the-maximum-number-of-threads-that-pthread-create-can-create