Brandon Buckley
CMPSC 473.2
4/1/2015

Project 3 Report

Function Descriptions

*main(argv, *argc[])*
The main function is a multithreaded program that reads the file and makes an output file containing the number of times an occurrence of a word appeared. It imports the input arguments from *argc[] which contain the filename, number of threads, and size of the buffers. It then initializes NUM_THREADS number of buffers and data structures. These buffers and data structures are passed into the initialization of threads. The reader and adder threads are first initialized and then waited on until finished. It then initializes one more reducer thread to reduce the buffers to one final output file.

map_reader(reader_struct)
The map reader function scans the file and adds words to a shared buffer. The reader_struct contains a filename, start position, end position, and a buffer containing words. The map_reader opens the file with the filename and starts scanning the file adding every word to the buffer until it reaches the end position. The buffer contains a lock so when adding or removing from the shared buffer, threads do not run into data race conditions. Once all of the map_reader threads have returns, there will be NUM_THREADS number of buffers containing all the words read from the file.

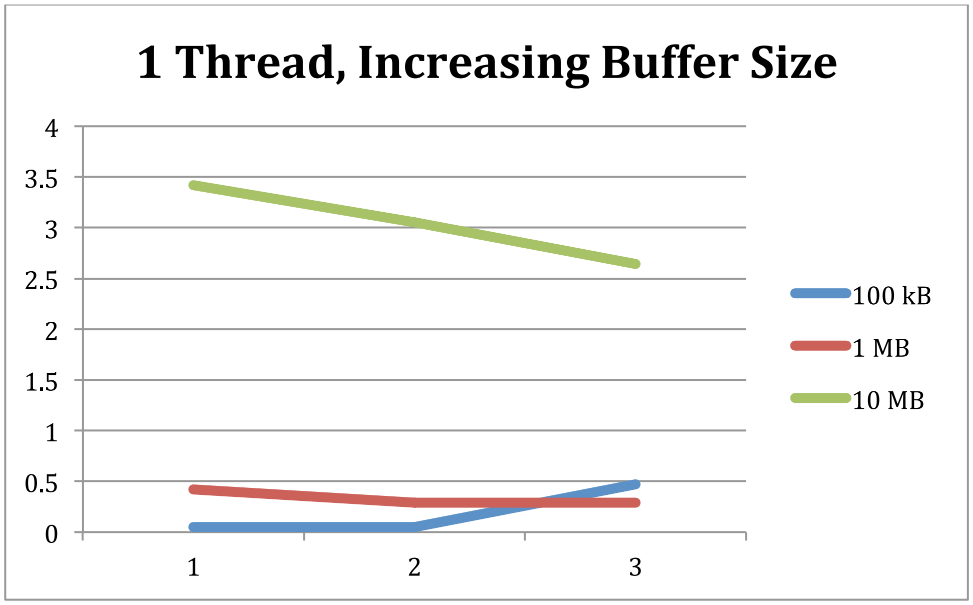
map_adder(buffers)
The map_adder function grabs input from the shared buffer output from the reader and adds it to another buffer adding the count to each word. It scans the second buffer searching for the word received from the first buffer and if found, increments the count of that word. If the word is not found, the word is appended to the end of the buffer with a count of 1. In the case that the second buffer runs out of memory, more memory will be malloc'ed and the old buffer will be copied over to the new, now larger, buffer. There will be NUM_THREADS number of adder buffers once all of the threads return. All of the buffers will have a unique word with the a count.

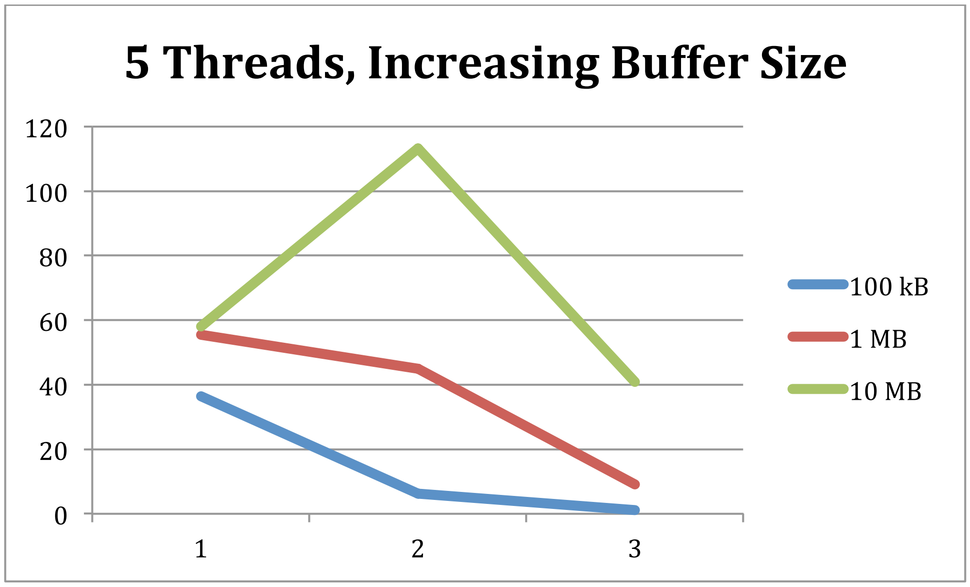
reducer(bufferArray)
The reducer function will search through NUM_THREAD number of adder buffers reducing all the words down to one entry with a final count. The reducer looks at the first word in the first buffer and increments a count corresponding to that word in the buffer. It then searches the rest of the buffers for that word and adds the total count of the occurrences of that word. Once finished, it will output that word to an output file and the count associated with it. Finally, when completed, there will be an output file containing the final list of words and the number of times it appeared in the file.

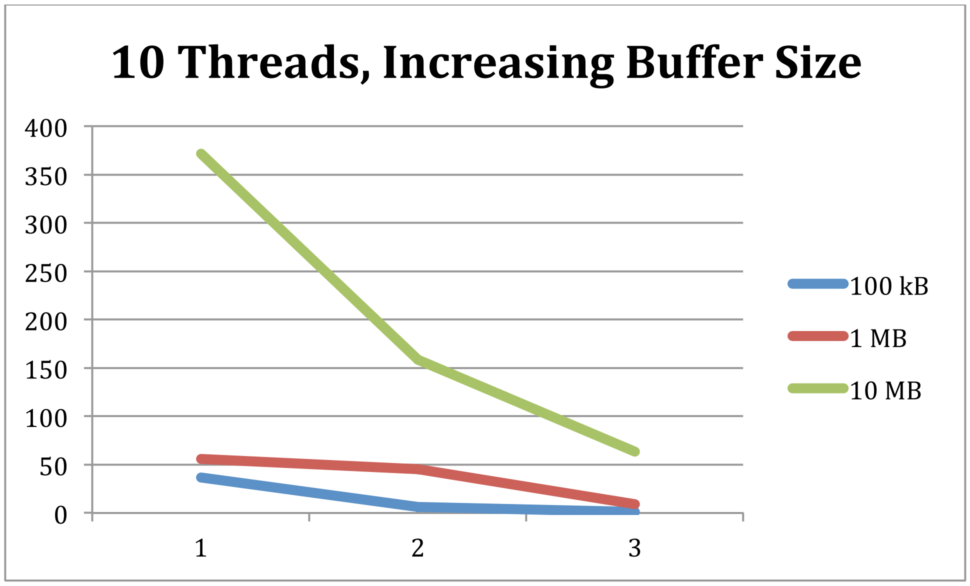
Results

Average Run Times (Running every test 20 times)

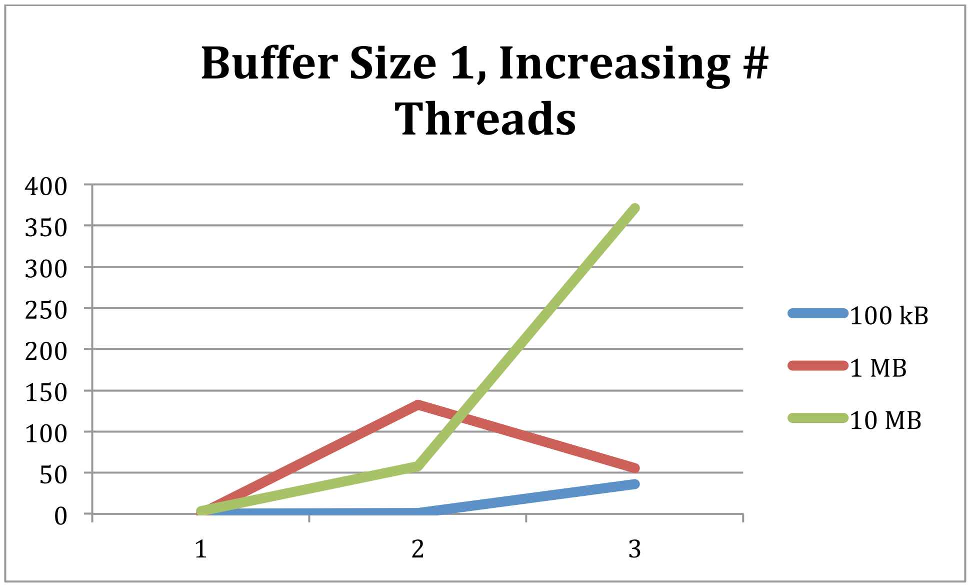
Test File	Num Threads	Buffer Size	Average Execution Time (s)
100 kB	1	1	0.05
100 kB	1	10	0.05
100 kB	1	100	0.47
100 kB	5	1	0.98
100 kB	5	10	0.83
100 kB	5	100	0.69
100 kB	10	1	36.45
100 kB	10	10	6.33
100 kB	10	100	1.14
1 MB	1	1	0.42
1 MB	1	10	0.29
1 MB	1	100	0.29
1 MB	5	1	132.17
1 MB	5	10	21.56
1 MB	5	100	6.51
1 MB	10	1	55.57
1 MB	10	10	45.02
1 MB	10	100	9.11
10 MB	1	1	3.42
10 MB	1	10	3.05
10 MB	1	100	2.64
10 MB	5	1	57.98
10 MB	5	10	113.32
10 MB	5	100	40.88
10 MB	10	1	371.52
10 MB	10	10	158.26
10 MB	10	100	63.43

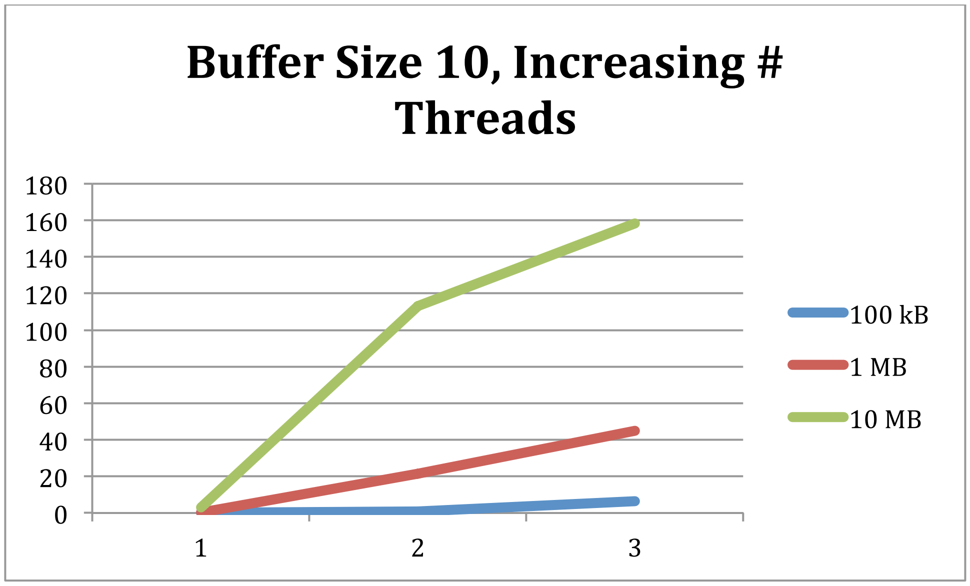

1 Thread, Increasing Buffer Size
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
100 kB
1 MB
10 MB

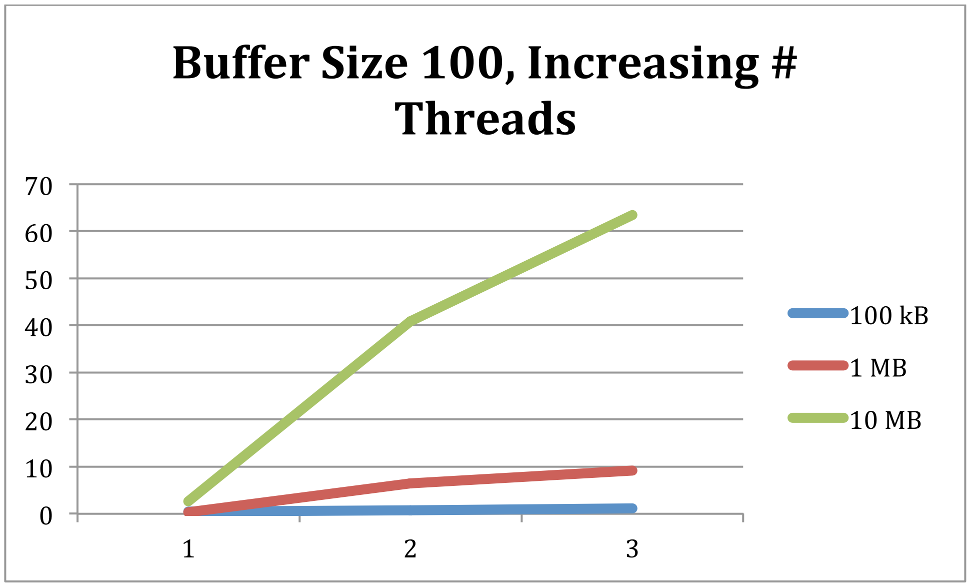

5 Threads, Increasing Buffer Size
120
100
80
60
40
20
0
1
2
3
100 kB
1 MB
10 MB



Overall, it appears as though increasing buffer size decreases the execution time. This makes sense because the buffers do not fill up as fast and do not need to be expanded as often.







Overall, the results show that as the number of threads increase, the execution time goes up. This makes sense because the more threads that are initialized, the more threads that have to join and sync back up. Also, the more threads there are, the more overhead there is initializing them and the more locks that can cause a thread to yield.