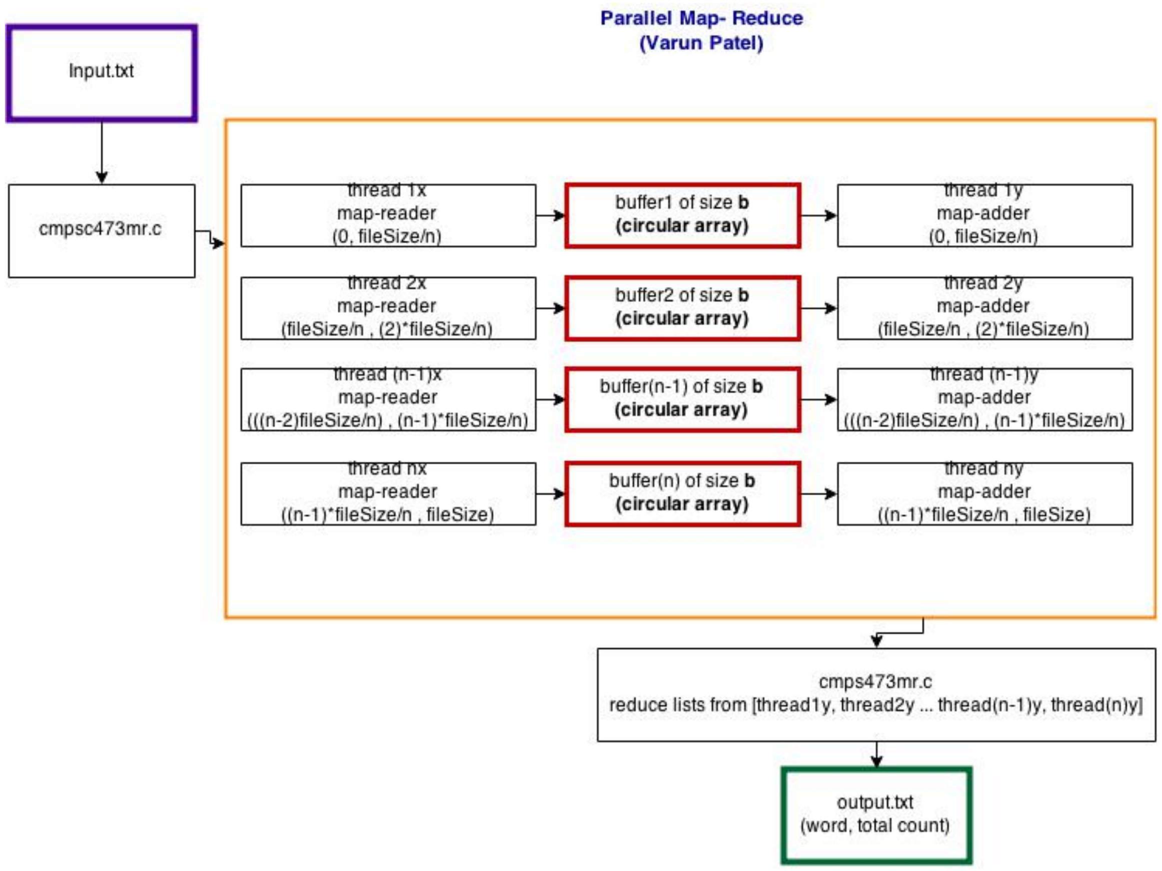
Varun Patel
CMPSC 473 Project 3 report

Parallel Map-Reduce Algorithm:

The below diagram depicts the Parallel Map-Reduce Algorithm that was assigned for us to implement:



A text file, number of threads, and buffer size are given as inputs to the program. Cmpsc473mr.c is the entry point of the parallel map-reduce algorithm that was assigned. In the main function, each input is captured and stored into a variable. A pointer to pthread_t variable is created to denote a variable number of threads that will be utilized to initialize a ‘n’ number of threads (an input to the program). A list is reserved on the heap and a reference to it is retained in the function so that it can be utilized later for the reducing phase.

In the main function, the pivots (ending and starting positions) for each workloads are stored in an array (called positions[]). A file pointer is used to seek to a totalSizeOfFile / n pivot. Then, if the character at the file pointer is a space character (acceptable character) the pivot is stored into the positions array. If the file pointer's position does not contain a space, it increments until the next space (at which point this pivot will be added to the positions array).

Each thread is initialized and passed in the mapper function along with its necessary arguments to carry out a slice of the workload. Each thread will read 'b' amounts of words from the file. These words are then read by the map-adder and inserted into a list of words with final counts. The data structure used to manage the buffer was a circular array. When the index of the buffer resets, the buffer is unlocked for processing by the adder function. Then, as soon as its contents are flushed out by the adder, the mapper function again locks this list to add new content (next set of 'b' amount of strings).

The last step is to take the lists returned by each thread and to reduce it by calling the reducer function. After the lists are reduced into a final output, it is written into an output.txt file. As a debugging measure, the resulting list from each thread is stored into [1,2, … ,(n-1), n].txt file. Each of these files corresponds to the results of each workload.

To measure the time it took for each program to run, before each thread is given its workload to carry out, a variable that hold the current time is calculated. Then, after the lists returned by each thread are processed into a final list, another variable containing the current time in milliseconds is calculated. The difference between the pre and post time variables is then noted down. The following table depicts the different combinations of #-of-threads vs. buffer-size. There were a total of 20 runs used to collect data.

I have included a sample of output files produced by my code in my submission for your reference. A limitation of my program (due to time constraint since it is the end of the semester and I have numerous senior projects due) is that it does not remove punctuation from the input file. As evidenced in the sample output file pertaining to Input File #1, the word ‘The’ has a count of 20 and the word ‘the’ has a count of 1104.

Data:

Input File #1 (each entry in milliseconds: 25th percentile, Average, 75th percentile):

	b = 1	b = 10	b = 100
n = 1	(213,224,284)	(254,267,331)	(270,310,314)
n = 5	(8866,10177,10531)	(6976,8486,8680)	(985,1081,1170)
n = 10	(51363,62568,78703)	(5633,5847,6633)	(822,835,953)

Input File #2 (each entry in milliseconds: 25th percentile, Average, 75th percentile):

	b = 1	b = 10	b = 100
n = 1	(2315,2422,2663)	(2231,3232,3345)	(2369,2271,2502)
n = 5	(13min,13min,15min)	(67150,73463,79432)	(9407,10812,11803)
n = 10	(673269,744296,781119)	(55489,63215,82056)	(6793,7701,9518)

Input File #3 (each entry in milliseconds: 25th percentile, Average, 75th percentile):

	b = 1	b = 10	b = 100
n = 1	(28046,28817,30376)	(24093,30371,33899)	(17883,17943,19104)
n = 5	(25min,25min,25min)	(724341,924571,965816)	(80037,92677,95486)
n = 10	(20min,20min,30min)	(518486,590470,614015)	(78249,84208,86437)

Varun Patel
CMPSC 473 Project 3 report

Examining the Data:

Upon careful review of the data set that was collected, it can be observed that there is a Producer and Consumer Relationship between the Reader thread and the Adder thread. The Reader is at most allowed to produce a certain amount of words per production (reading words from a file) while the Consumer can consumer up to the buffer size at a time (if buffer is full, the consumer can decide to consume the entire buffer during that run before the consumer thread goes into a "holding pattern" or a "wait state" while the producer produces up to a buffer-size amount of words for the Consumer to consume upon waking up- This cycle continues till all of the workload has been processed.). The data set shows that the larger the producing ability (the larger the buffer size), the more the producer can produce during a consumer's "wait" or "sleep" state. Allowing the buffer to fill up makes the algorithm that was implemented (Parallel Word Count Map Reduce) to run with faster run times before the Adder can consume the entire buffer during each of it's awake states. The shared buffer was locked while the producer (mapper/reader) was writing words to the buffer. Upon completion of each write by the mapper/reader, the buffer was unlocked for the consumer to potentially consume the contents written on to the buffer.