FIT3142 Assignment 1

Name: Mah Ying Qi

Student ID: 32796765

Serial string-matching algorithm

Algorithm Chosen: Bloom Filter

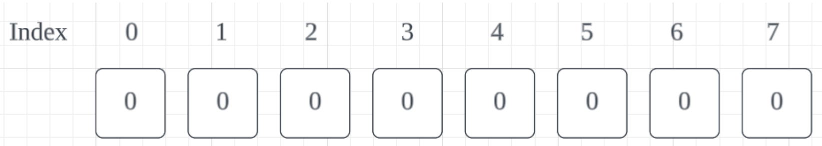
Pseudocode:

```
main() {

wordFile[4] ← list of 4 files
queryFile[4] ← list of 4 files

numberOfWords ← 0

for file in wordFile:
        numberOfWords += numberOfWords in file

bitArray[size calculated using number of words in wordFiles]
wordlist[numberOfWords]

for file in wordFile:
        put all words in file into wordList

for word in input:
        insert(bitArray, word)

for file in queryFile:
        numberOfQueries += numberOfQueries in file

queryList [numberOfQueries]

for file in queryFile:
        put all queries into queryList

truePositiveCount ← 0
falsePositiveCount ← 0

for (query, exist) in queryList:
        found = lookUp(bitArray, query)
        if(found and exist):
                truePostiveCount++
        else if(found and not exist):
                falsePositiveCount++

}
```

```
insert(bitArray, input){
        bitArray[hash1(input)] ← 1
        bitArray[hash2(input)] ← 1
        bitArray[hash3(input)] ← 1
        bitArray[hash4(input)] ← 1
        bitArray[hash5(input)] ← 1
}

lookup(bitArray, input){
        return bitArray[hash1(input)] ← 1
        AND bitArray[hash2(input)] ← 1
        AND bitArray[hash3(input)] ← 1
        AND bitArray[hash4(input)] ← 1
        AND bitArray[hash5(input)] ← 1
        as boolean
}
```

The algorithm counts the number of words and queries in the files. Then constructs a bit array based on the total number of words, an array with length of the number of words, and array with length of the number of queries. After that all the words and queries are saved into their respective array. Next, it inserts all the words into bit array by using 5 different hash functions to hash the input words. With the hashed value “k”, it sets the bit array such that the kth value of the bit array is set as 1. For lookup, it uses the same hash functions to hash the queries. With the hashed value “k”, it checks if the kth value of bit array is 1, if all of the values checked are set to one, it returns true, indicating that the queried word exists in the word file.

Below is an illustration of the bit array, insert and lookup:



Bit Array with length = 8

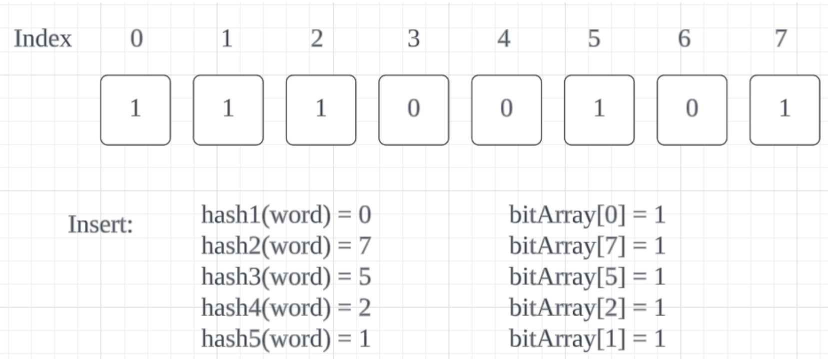


Illustration of word insertion, hashed values are 0,7,5,2 and 1. So the array location 0,1,2,5 and 7 are set to 1.

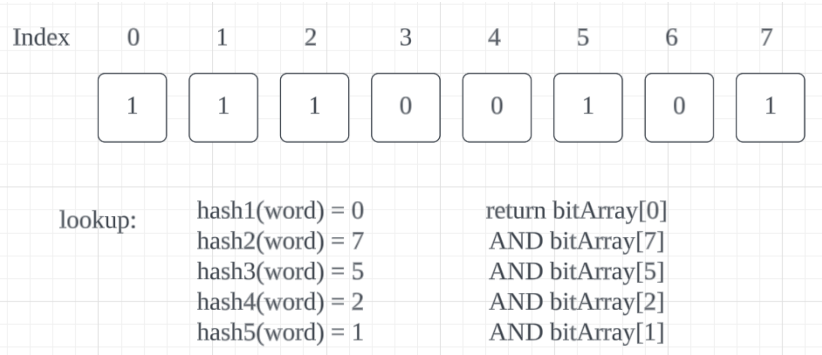


Illustration when lookup returns true, hashed values are 0,7,5,2 and 1. The array locations with index 0,7,5,2 and 1 are 1 so it returns true, the words exists.

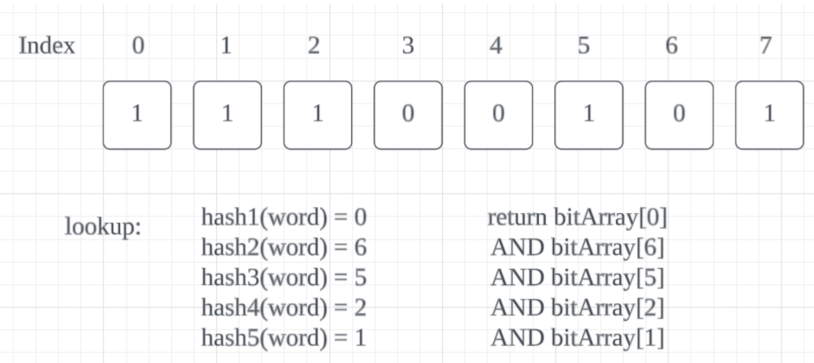


Illustration when lookup returns false, hashed values are 0,6,5,2 and 1. The array location with index 6 is 0, so it returns false, the word does not exist.

The bit array size is calculated using the formula:

$$size = -\frac{n \ln P}{(\ln 2)^2}$$

n = number of words to be inserted
P = false positive possibility

This is to ensure that the false positive possibility of the algorithm can be controlled at our desired level, which is 0.05 for my code.

I have used 5 different hash functions in my algorithm.
Hash 1 multiplies the hash value by 31 to provide a unique hash value as 31 is a prime number
Hash 2 is a basic adding hash function.
Hash 3 is a basic multiplying hash function.
Hash 4 is a djb2 hash function.
Hash 5 is an sdbm hash function

Justification
The reason I chose bloom filter is because in consists of several parts, including inserting, lookup, and file reading, which is ideal for me to analyse and implement parallelization on it.

Details of Input
User is allowed to enter 4 word files, and 4 query files.
Assumptions of the files:

- No large number of duplicate words in the word and query files
- Words and sentences are allowed
- No comma in the middle of word/sentence.
- The algorithm is case sensitive
- The files are in csv format
- Both word file and query file have a header, input file has only one column of values, query file has two columns, the first column is value, second column is an integer, 1 or 0, 1 indicates it exists in the input file, 0 otherwise

Files I used to test my algorithm are generated using python code in the appendix. I have used the code to generate 10 different set of files, from n = 1000000 to 10000000, increase by 1000000 each time. The code will create n number of unique words and insert them into 4 word files and 4 query files, each word file will contain n/8 number of words and query file will contain n/4 number of words, which means half of the words does not exists in word file, which is to test the accuracy of the algorithm.

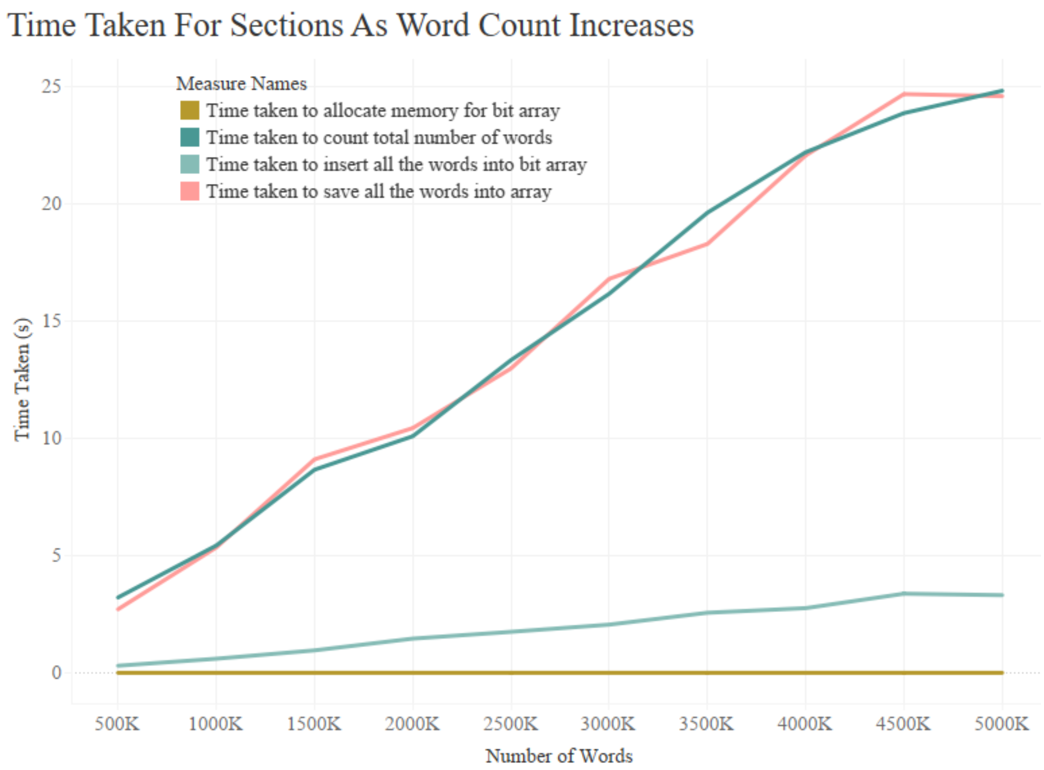
Measure and analyse the performance of the serial algorithm

Specifications of test: 8 Cores CPU, 5.55GB Memory

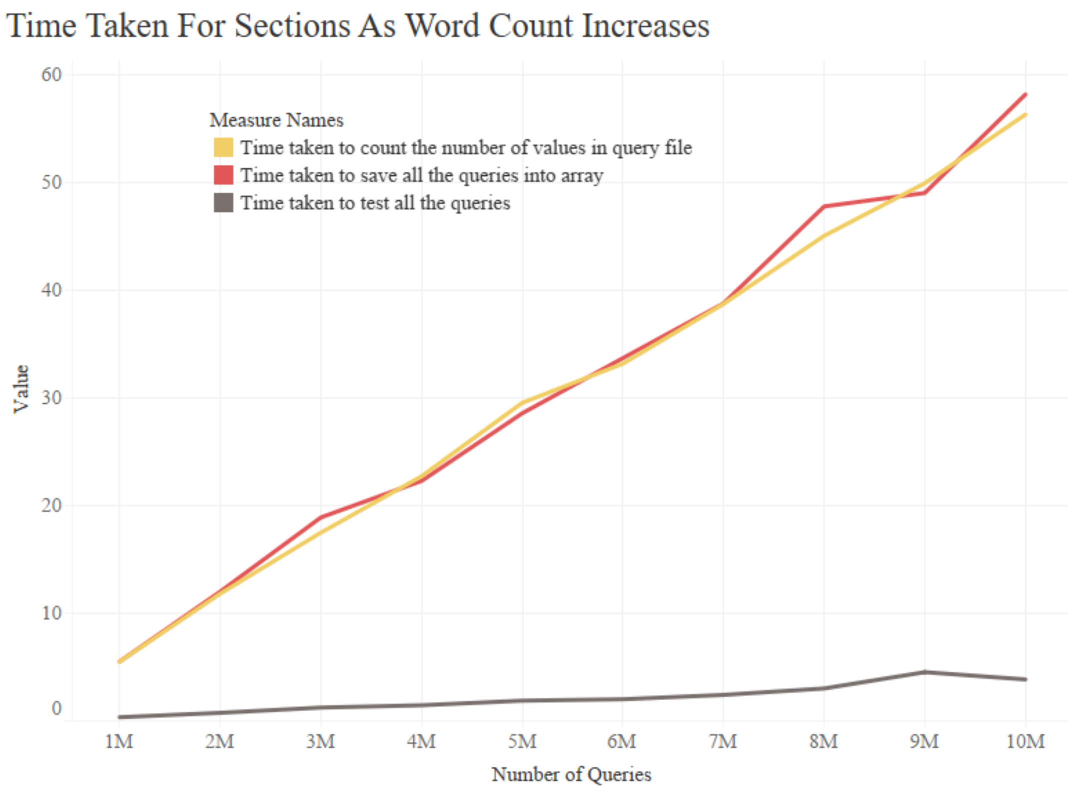
Number of words in word list	Number of words in query list	True Positive Count	False Positive Count	Accuracy
500000	1000000	500000	25080	0.97492
1000000	2000000	1000000	50355	0.974823
1500000	3000000	1500000	75938	0.974687
2000000	4000000	2000000	100675	0.974831
2500000	5000000	2500000	126451	0.97471
3000000	6000000	3000000	151146	0.974809
3500000	7000000	3500000	176763	0.974748
4000000	8000000	4000000	201731	0.974784
4500000	9000000	4500000	226586	0.974824
5000000	10000000	5000000	251609	0.974839

From this table, we can observe that even though the number of words in word list and query list increases, the accuracy remains around 0.975, with 100% the true positive rate, and false positive rate is around 5%.

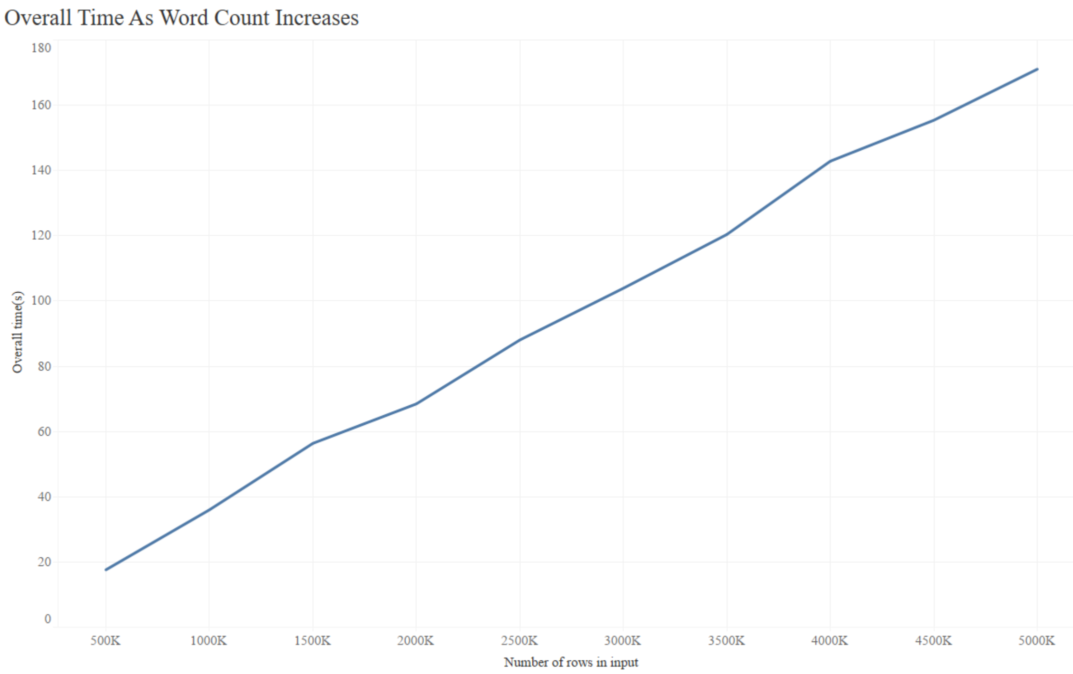
Number of rows in input	Number of rows in query	Time taken to count total number of words	Time taken to allocate memory for bit array	Time taken to save all the words into array	Time taken to insert all the words into bit array	Time taken to count the number of values in query file	Time taken to save all the queries into array	Time taken to test all the queries	Overall time(s)
500000	1000000	3.21573	0.000014	2.71285	0.301817	5.504679	5.518627	0.353343	17.6075
1000000	2000000	5.43308	0.000014	5.349963	0.600559	11.822746	12.01873	0.760296	35.98573
1500000	3000000	8.66885	0.000014	9.112105	0.956814	17.486802	18.895	1.247047	56.36699
2000000	4000000	10.0965	0.000015	10.44647	1.458913	22.718098	22.27925	1.464565	68.46419
2500000	5000000	13.3481	0.000023	12.98969	1.748263	29.528661	28.55704	1.887302	88.0595
3000000	6000000	16.1806	0.000016	16.81747	2.056708	33.157921	33.6772	2.022464	103.9128
3500000	7000000	19.6414	0.000021	18.30532	2.563188	38.709284	38.75252	2.424698	120.3968
4000000	8000000	22.2209	0.000017	22.08338	2.760144	45.025705	47.75314	3.021123	142.8648
4500000	9000000	23.8896	0.000414	24.70281	3.376759	49.901845	49.00322	4.535727	155.4108
5000000	10000000	24.8434	0.000015	24.61129	3.31473	56.28113	58.14598	3.858138	171.055



The most time-consuming part of the inserting algorithm is counting number of rows and saving all the input values into an array.



The most time-consuming part of the lookup algorithm is counting number of rows and saving all the input values into an array.



As the size of word list and query list increases, the time taken to process them increases. The parts that involve file reading are the most time consuming as reading an external file is rather time consuming.

Dependency Analysis

a. Reading rows of one file:

When i = 1; I_1 = {file pointer}; O_1 = {content of row1, file pointer moves to next row}
When i = 2; I_2 = {file pointer from I_1}; O_2 = {content of row after row 1, file pointer moves to next row}
$I_1 \cap O_2 = \emptyset$;
$I_2 \cap O_1 \neq \emptyset$;
$O_1 \cap O_2 = \emptyset$;

This cannot be parallelised as it has a flow dependency, and we can only access one row of the file at a time.

b. Reading of content of multiple files

When i = 1; I_1 = {file 1}; O_1 = {content of file 1}
When i = 2; I_2 = {file 2}; O_2 = {content of file 2}

$I_1 \cap O_2 = \emptyset$;
$I_2 \cap O_1 = \emptyset$;
$O_1 \cap O_2 = \emptyset$;

This can be parallelised as there is no dependency in between 2 executions.

c. Insert values into bit array

When i = 1; I_1 = {word[1], bitarray};

O_1 = {bitArray[hash1(word [1])] = 1, bitArray[hash2(word [1])] = 1, bitArray[hash3(word [1])] = 1, bitArray[hash4(word [1])] = 1, bitArray[hash5(word [1])] = 1 }

When i = 2; I_2 = { word [2], bitArray};
O_2 = {bitArray[hash1(word [2])] = 1, bitArray[hash2(word [2])] = 1, bitArray[hash3(word [2])] = 1, bitArray[hash4(word [2])] = 1, bitArray[hash5(word [2])] = 1 }

$I_1 \cap O_2 = \emptyset$;
$I_2 \cap O_1 = \emptyset$;
$O_1 \cap O_2 \neq \emptyset$;

There could be an output dependency when the hash returns the same value. But this can be resolved by using the pragma omp atomic statement. This might result in some serial execution but in general still parallel.

d. Test all query values

When i = 1; I_1 = {query[1], bitarray};
O_1 = {bitArray[hash1(input[1])] && bitArray[hash2(input[1])] && bitArray[hash3(input[1])]&& bitArray[hash4(input[1])]&& bitArray[hash5(input[1])], truePositiveCount++ or falsePositiveCount++}

When i = 2; I_2 = {query[2], bitArray};
O_2 = {bitArray[hash1(input[2])] && bitArray[hash2(input[2])] && bitArray[hash3(input[2])]&& bitArray[hash4(input[2])]&& bitArray[hash5(input[2])], truePositiveCount++ or falsePositiveCount++}

$I_1 \cap O_2 = \emptyset$;
$I_2 \cap O_1 = \emptyset$;
$O_1 \cap O_2 \neq \emptyset$;

This can be parallalized even though there is a output dependency because the computation of truePositiveCount and falsePositiveCount can be resolved by using reduction in OpenMp

Calculate theoretical speedup

For the calculation of speed up, we use the execution time for input size =500000, and query size = 1000000.

Number of rows in input	Number of rows in query	Time taken to count total number of words	Time taken to allocate memory for bit array	Time taken to save all the words into array	Time taken to insert all the words into bit array	Time taken to count the number of values in query file	Time taken to save all the queries into array	Time taken to test all the queries	Overall time(s)
500000	1000000	3.215726	0.000014	2.71285	0.301817	5.504679	5.518627	0.353343	17.6075

Amdahl's Law
Number of CPU = 8, Memory size = 5GB
Ratio of serial part to overall part = 0.000014/17.6075 = 0.0000007

Speedup for parallelizing multiple file readings. For this portion of code,
parallel ratio/ number of processors = 1-0.0000007= 0.9999993

1/(0.0000007+(0. 0.9999993/8)) = 7.99996

The theoretical speedup of my algorithm is close to 8.

Parallel implementation

I have set the number of threads to 4 in my parallel algorithm, and the pseudocode of my parallel algorithm is as follows:

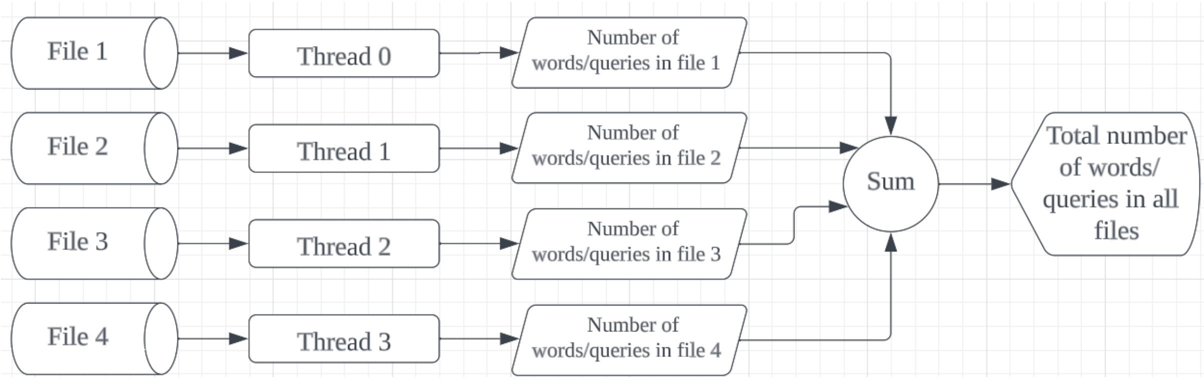
```
Main(){

wordFile[4] ← list of 4 files
queryFile[4] ← list of 4 files

numberOfWords ←0

wordlist[]

forAll file in wordFile and queryfile do in parallel:
        if in wordFile:
                numberOfWords += countRow(file)

        else:
                numberOfQueries+=countRow(file)

forAll file in wordFile and queryfile do in parallel:
        if in wordFile:

                add words in file into wordList
        else:
                add queries in file into queryList

forAll word in wordlist do in parallel:
        insert(bitArray, word)

forAll file in queryFile do in parallel:
        for query in file:
                put query in wordList

truePositiveCount ←0
falsePositiveCount←0

forAll (query, exist) in wordlist do in parallel:
        found = lookUp(bitArray, query)
        if(found and exist):
                truePostiveCount++
        else if(found and not exist):
                falsePositiveCount++
        }
```

```
insert(bitArray, input){
                bitArray[hash1(input)] ← 1
                bitArray[hash2(input)] ← 1
                bitArray[hash3(input)] ← 1
                bitArray[hash4(input)] ← 1
                bitArray[hash5(input)] ← 1
        }

        lookup(bitArray, input){
                return bitArray[hash1(input)] ← 1
                AND bitArray[hash2(input)] ← 1
                AND bitArray[hash3(input)] ← 1
                AND bitArray[hash4(input)] ← 1
                AND bitArray[hash5(input)] ← 1
                as boolean
        }
```

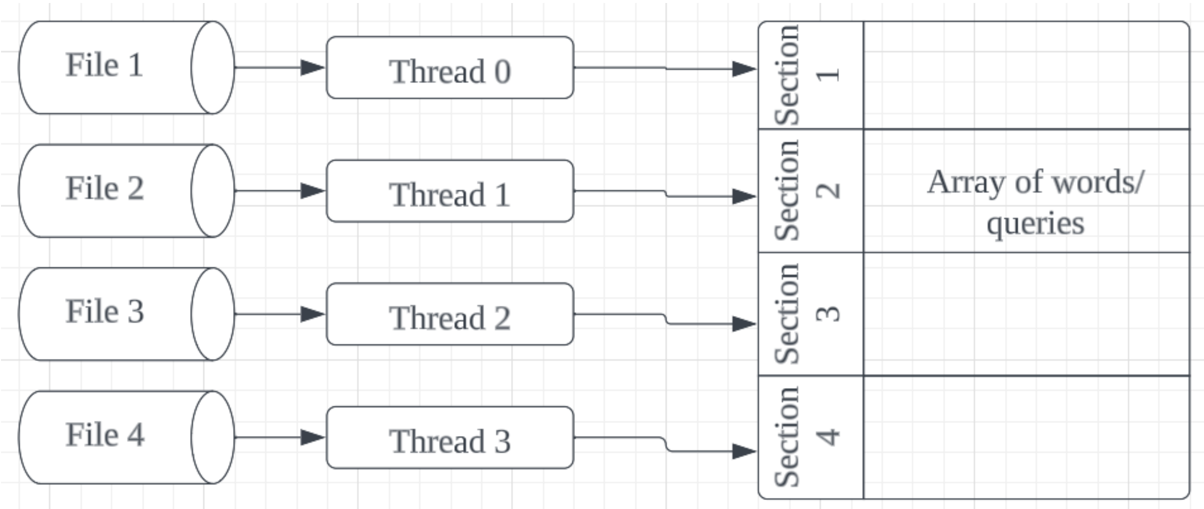
According to the dependency analysis, I have adjuster the code to parallelise the algorithm. The base of the algorithm is not different from the serial code, and it produces the same result, but some of the parts are parallelized. The reading of 8 files, insertion of words and lookup of queries are parallelized as it’s shown that they can be done concurrently in the dependency analysis.

Parallelism of Overall Process

The illustration below is a half of the actual parallelization, as it only includes 4 threads. My actual parallelization includes 8 threads which will read 8 files in parallel.

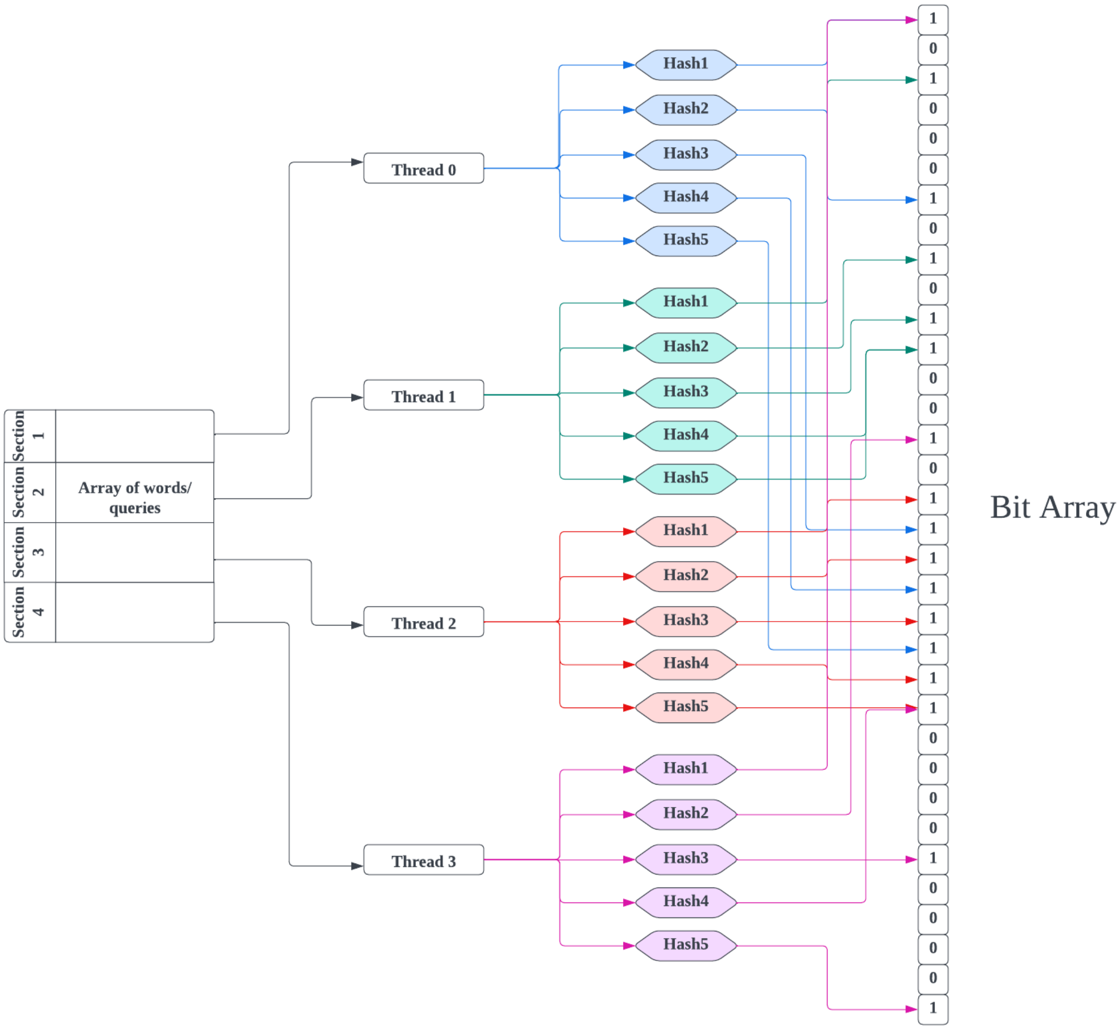


The first parallelized part is counting number of words and queries, the reading of one file cannot be parallelized, but I parallelized the reading of multiple files, which is by reading one file in a thread, and another file in another thread. By parallelizing this, the program can compute the number of words in parallel and combine their results at the end. This is achieved by using "reduction" syntax in OpenMP to allow the combination of number of values at the end of the parallel process. By parallelising this part, the time taken to count words and queries from files can be shortened.

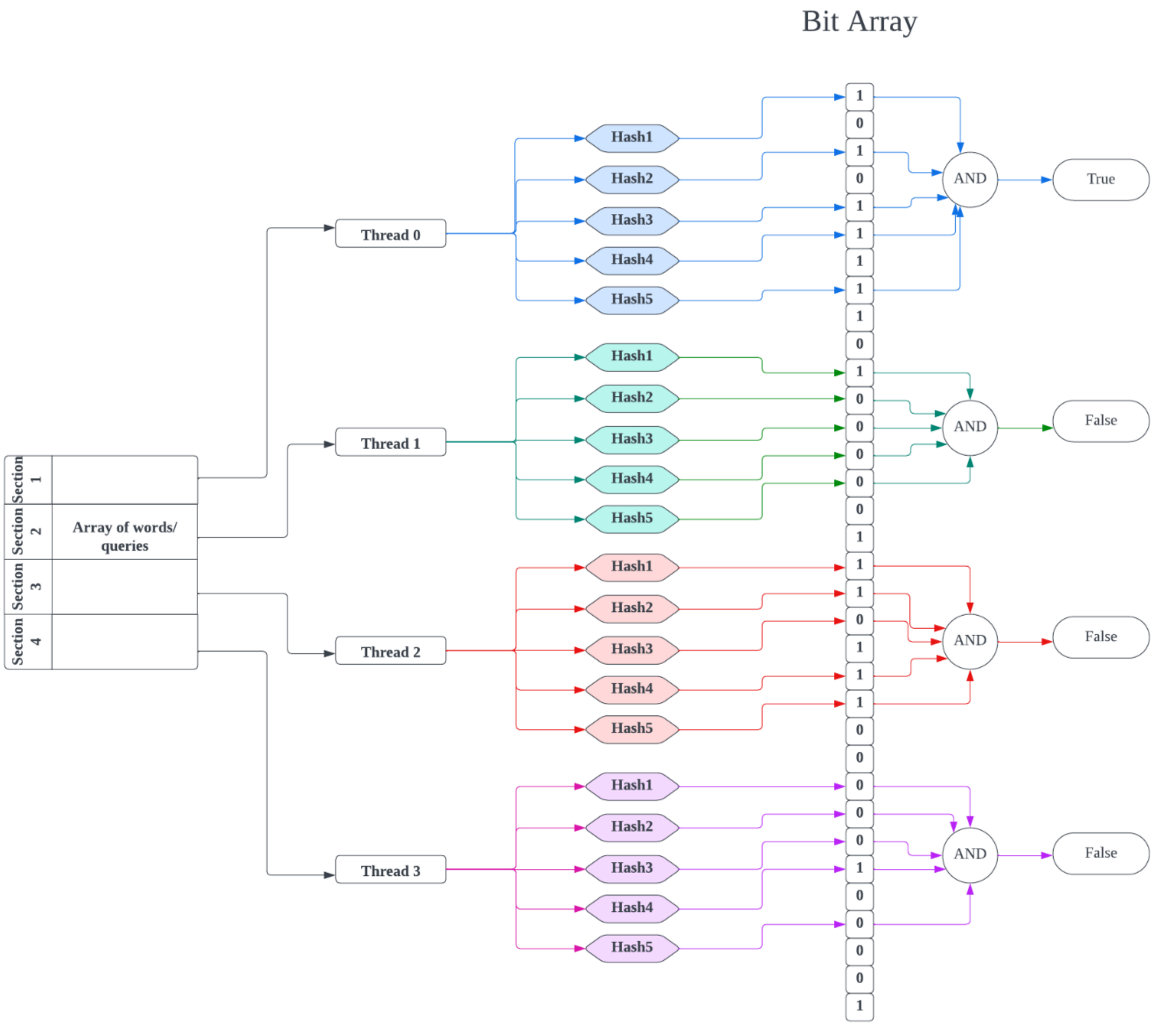


Second part is putting words and queries in their respective array, this is also achieved by parallelizing the reading of multiple files in multiple threads. Each thread takes one file, and as they run parallelly, they insert the values in the files into their preset sections of the array at the same time. This is achieved by using "atomic" syntax of OpenMP so that the process is interrupted as less as possible. By parallelising this part, the time taken to put words and queries from files into their respective array can be shortened.

Parallelism of Inner Workings



The next part parallelized is insertion of words into bit array. This is done by letting multiple threads execute the insert of different words at the same time, when the number of threads is 4, 4 threads will execute insertion of different chunks of words in parallel. By parallelising the process, the time taken to insert all the words can be shortened.



The last part I parallelized is the lookUp of the file, it is parallalized by letting each thread process each chunk of queries and let the threads run at the same time. With this parallelization, the look up of queries in different section can be done in parallel and it will shorten the time to look up through all the queries.

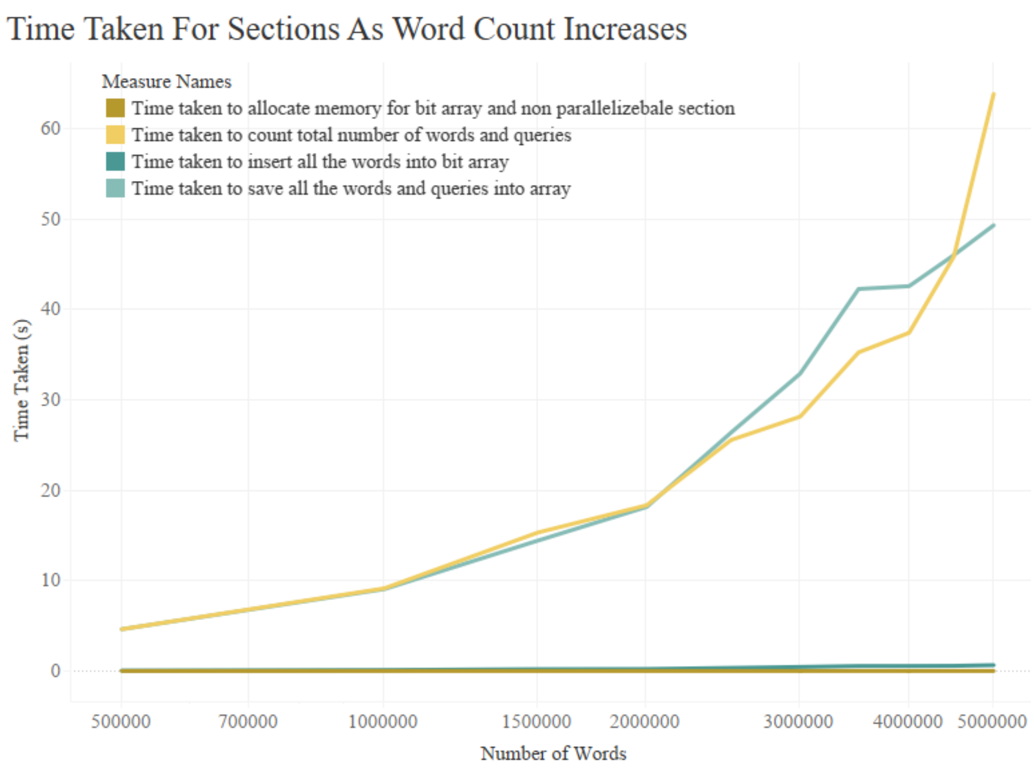
Analyse and Evaluate The Performance of The Parallel Algorithm

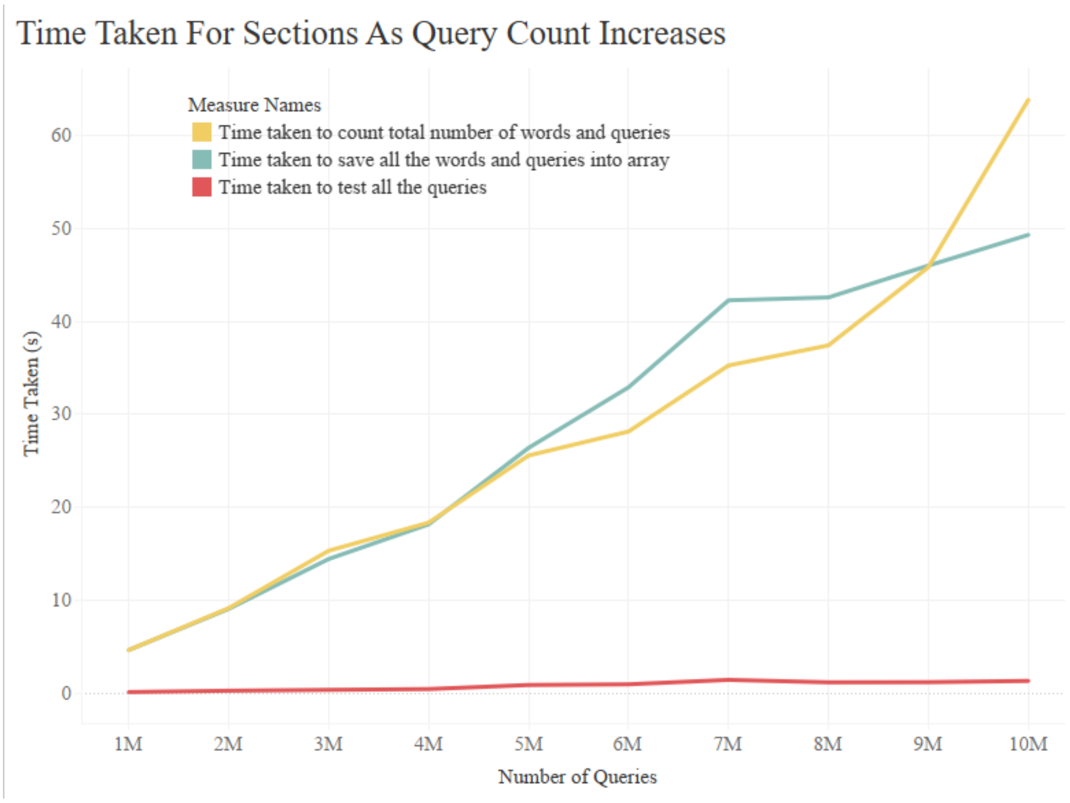
Assessment Environment:
Number of threads = 8
Range of word counts = 500000 to 5000000
Range of query counts = 1000000 to 10000000
Memory = 5GB

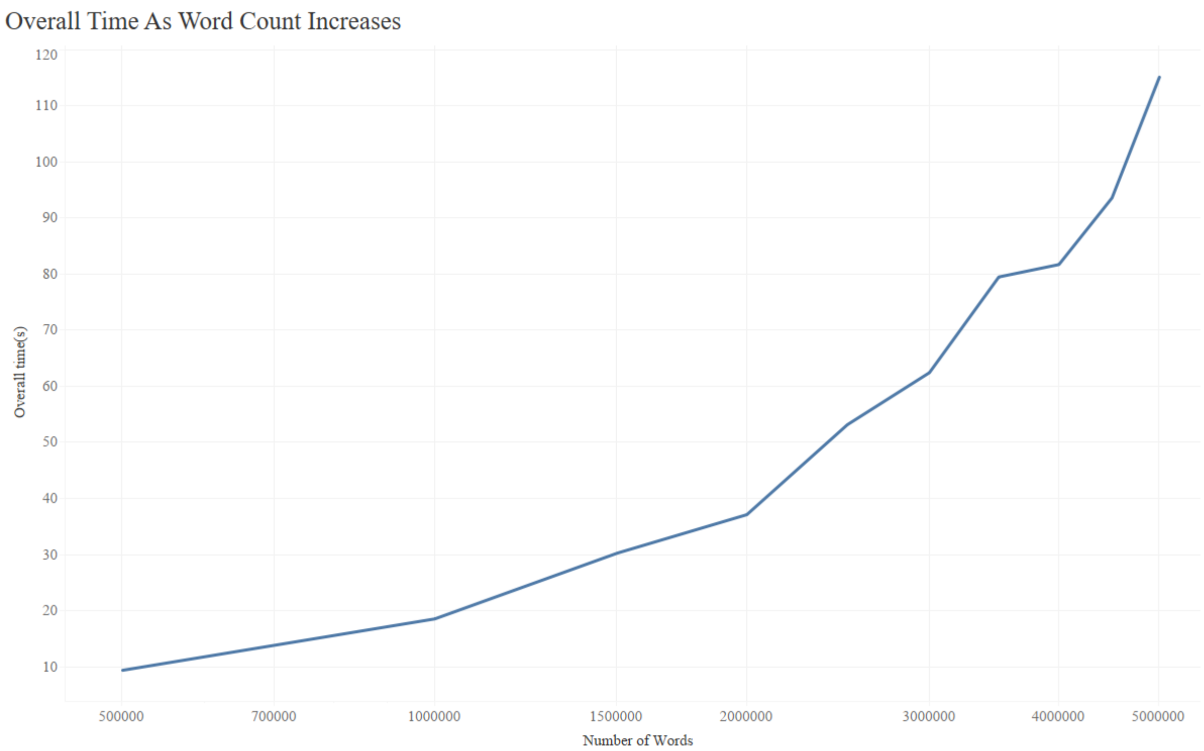
Performance on a Single Computer (My personal laptop)

Number of words in word list	Number of words in query list	True Positive Count	False Positive Count	Accuracy
500000	1000000	500000	25080	0.97492
1000000	2000000	1000000	50355	0.974823
1500000	3000000	1500000	75938	0.974687
2000000	4000000	2000000	100675	0.974831
2500000	5000000	2500000	126451	0.97471
3000000	6000000	3000000	151146	0.974809
3500000	7000000	3500000	176422	0.974797
4000000	8000000	4000000	201731	0.974784
4500000	9000000	4500000	226586	0.974824

Number of rows in input	Number of rows in query	Time taken to count total number of words and queries	Time taken to allocate memory for bit array and non parallelizebale section	Time taken to save all the words and queries into array	Time taken to insert all the words into bit array	Time taken to test all the queries	Overall time(s)	Time taken to count total number of words and queries	Time taken to allocate memory for bit array and non parallelize bale section
500000	1000000	4.623333	0.000019	4.623259	0.044797	0.09161	9.383292	4.623333	0.000019
1000000	2000000	9.138615	0.000016	9.055319	0.107193	0.240204	18.54175	9.138615	0.000016
1500000	3000000	15.312631	0.000017	14.42192	0.197286	0.332313	30.26452	15.312631	0.000017
2000000	4000000	18.334187	0.000016	18.15137	0.215173	0.41746	37.11872	18.334187	0.000016
2500000	5000000	25.561526	0.000018	26.38683	0.340269	0.855167	53.14425	25.561526	0.000018
3000000	6000000	28.142641	0.000019	32.90239	0.445351	0.928936	62.42285	28.142641	0.000019
3500000	7000000	35.253328	0.000017	42.26651	0.550676	1.407535	79.47834	35.253328	0.000017
4000000	8000000	37.415374	0.000019	42.57867	0.553376	1.142626	81.69068	37.415374	0.000019
4500000	9000000	45.829341	0.000017	46.00349	0.569321	1.158546	93.56153	45.829341	0.000017


Time Taken For Sections As Word Count Increases
Measure Names
Time taken to allocate memory for bit array and non parallelizebale section
Time taken to count total number of words and queries
Time taken to insert all the words into bit array
Time taken to save all the words and queries into array
Time Taken (s)
0
10
20
30
40
50
60
500000
700000
1000000
1500000
2000000
3000000
4000000
5000000
Number of Words


Time Taken For Sections As Query Count Increases
Measure Names
Time taken to count total number of words and queries
Time taken to save all the words and queries into array
Time taken to test all the queries
Time Taken (s)
0
10
20
30
40
50
60
1M
2M
3M
4M
5M
6M
7M
8M
9M
10M
Number of Queries

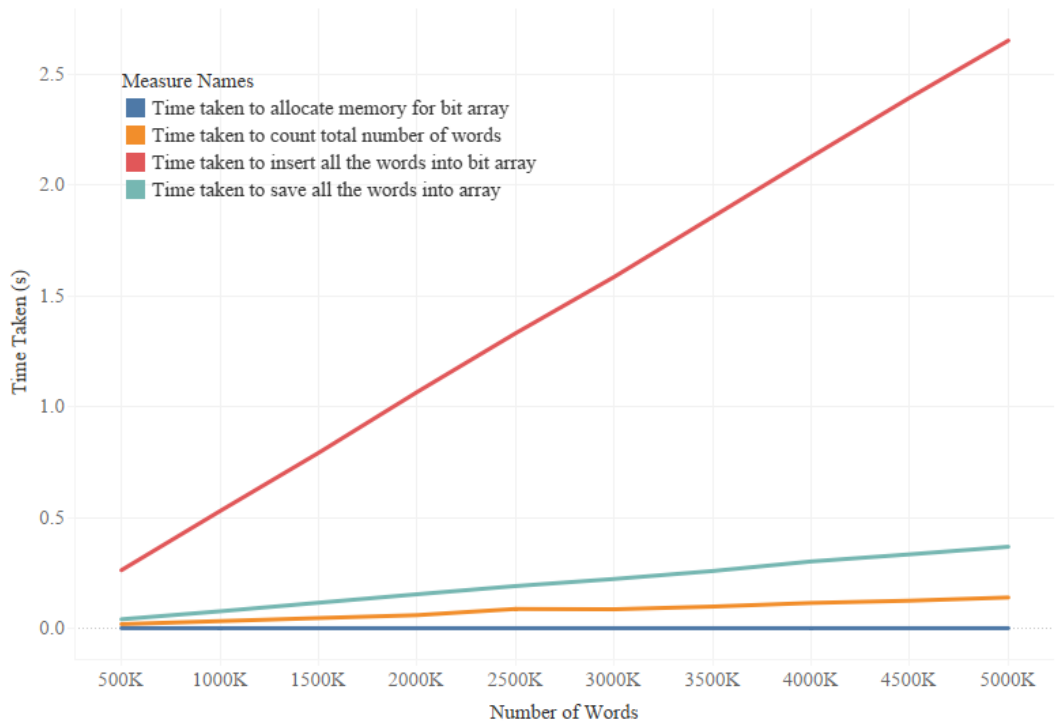

Overall Time As Word Count Increases
Overall time(s)
120
110
100
90
80
70
60
50
40
30
20
10
500000
700000
1000000
1500000
2000000
3000000
4000000
5000000
Number of Words

Performance on a CAAS (Serial)

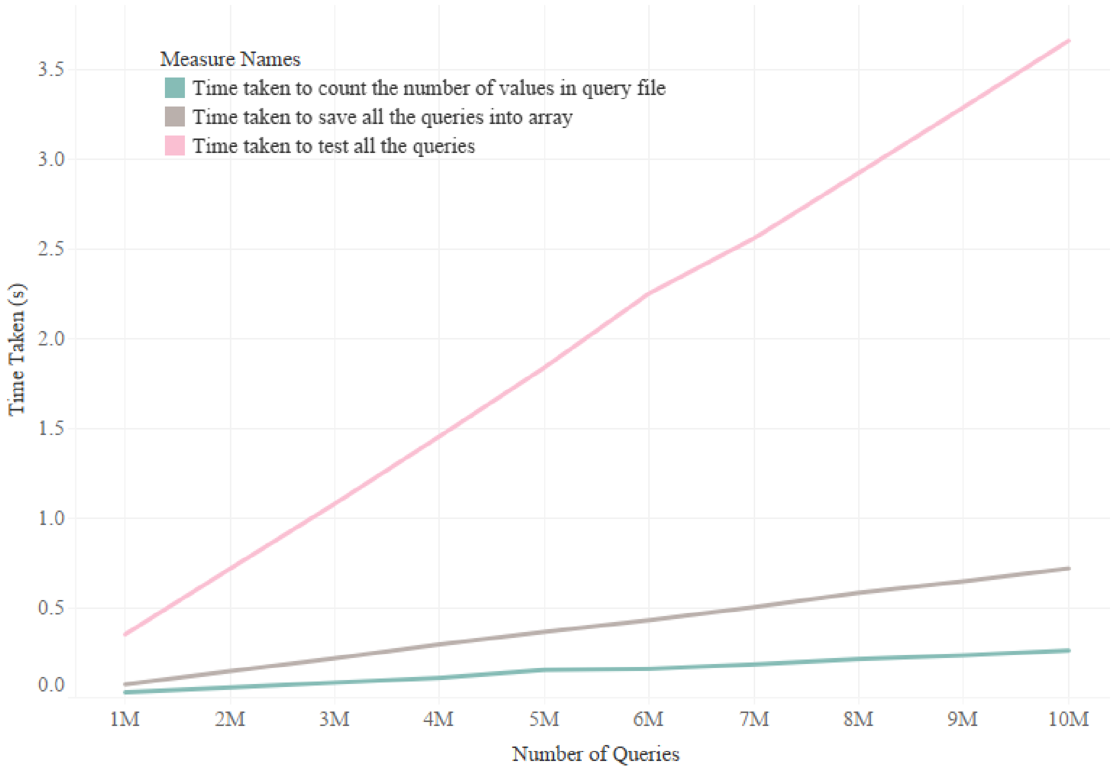
Number of words in word list	Number of words in query list	True Positive Count	False Positive Count	Accuracy
500000	1000000	500000	25080	0.97492
1000000	2000000	1000000	50355	0.974823
1500000	3000000	1500000	75938	0.974687
2000000	4000000	2000000	100675	0.974831
2500000	5000000	2500000	126451	0.97471
3000000	6000000	3000000	151146	0.974809
3500000	7000000	3500000	176422	0.974797
4000000	8000000	4000000	201731	0.974784
4500000	9000000	4500000	226586	0.974824

Number of rows in input	Number of rows in query	Time taken to count total number of words	Time taken to allocate memory for bit array	Time taken to save all the words into array	Time taken to insert all the words into bit array	Time taken to count the number of values in query file	Time taken to save all the queries into array	Time taken to test all the queries	Overall time(s)
500000	1000000	0.018286	0.000005	0.040162	0.261612	0.031706	0.075056	0.353756	0.780614
1000000	2000000	0.031816	0.000009	0.076108	0.528358	0.057969	0.148676	0.71905	1.562024
1500000	3000000	0.045632	0.000005	0.114789	0.791112	0.085257	0.220637	1.081523	2.338991
2000000	4000000	0.058783	0.000005	0.152984	1.065574	0.111737	0.297852	1.457155	3.144128
2500000	5000000	0.086718	0.000005	0.190093	1.331102	0.155796	0.367892	1.840312	3.971957
3000000	6000000	0.085232	0.000006	0.22204	1.585396	0.161933	0.432969	2.253807	4.741418
3500000	7000000	0.097456	0.000007	0.25795	1.856675	0.186727	0.505494	2.560422	5.464768
4000000	8000000	0.113792	0.000009	0.301102	2.127684	0.216589	0.585841	2.925082	6.270144
4500000	9000000	0.123982	0.000007	0.333233	2.394702	0.237418	0.649497	3.291141	7.030017

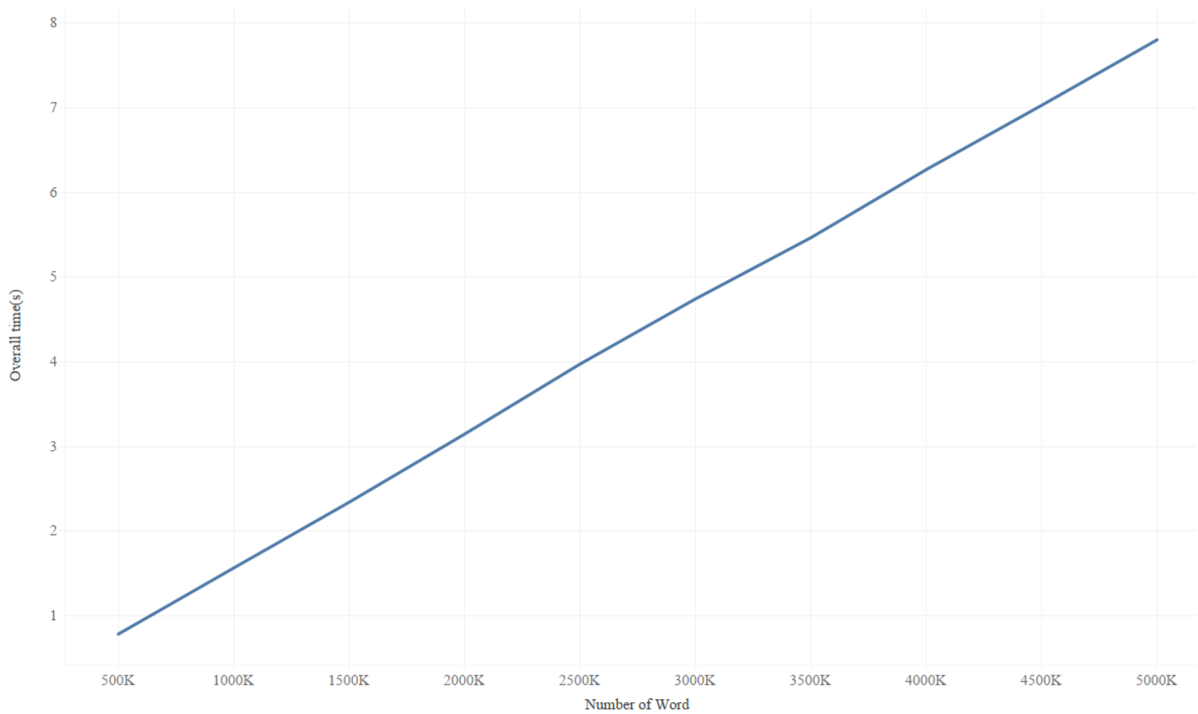
Time Taken For Sections As Word Count Increases



Time Taken For Sections As Query Count Increases



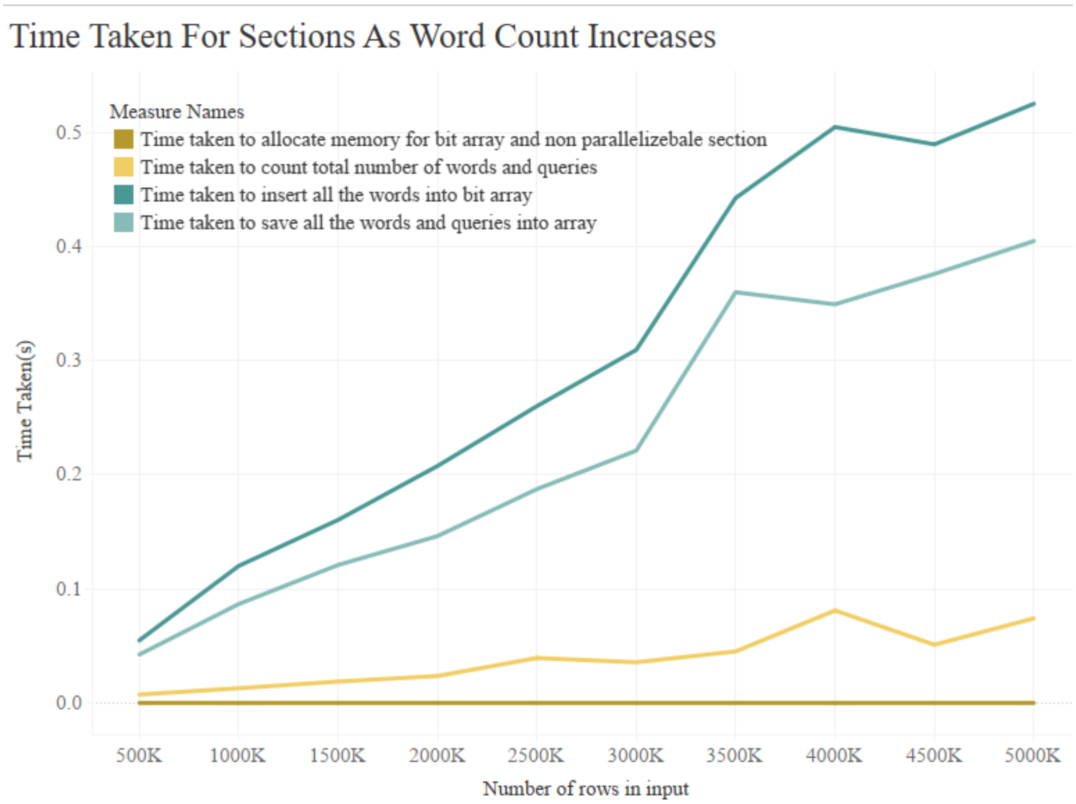
Overall Time As Word Count Increases

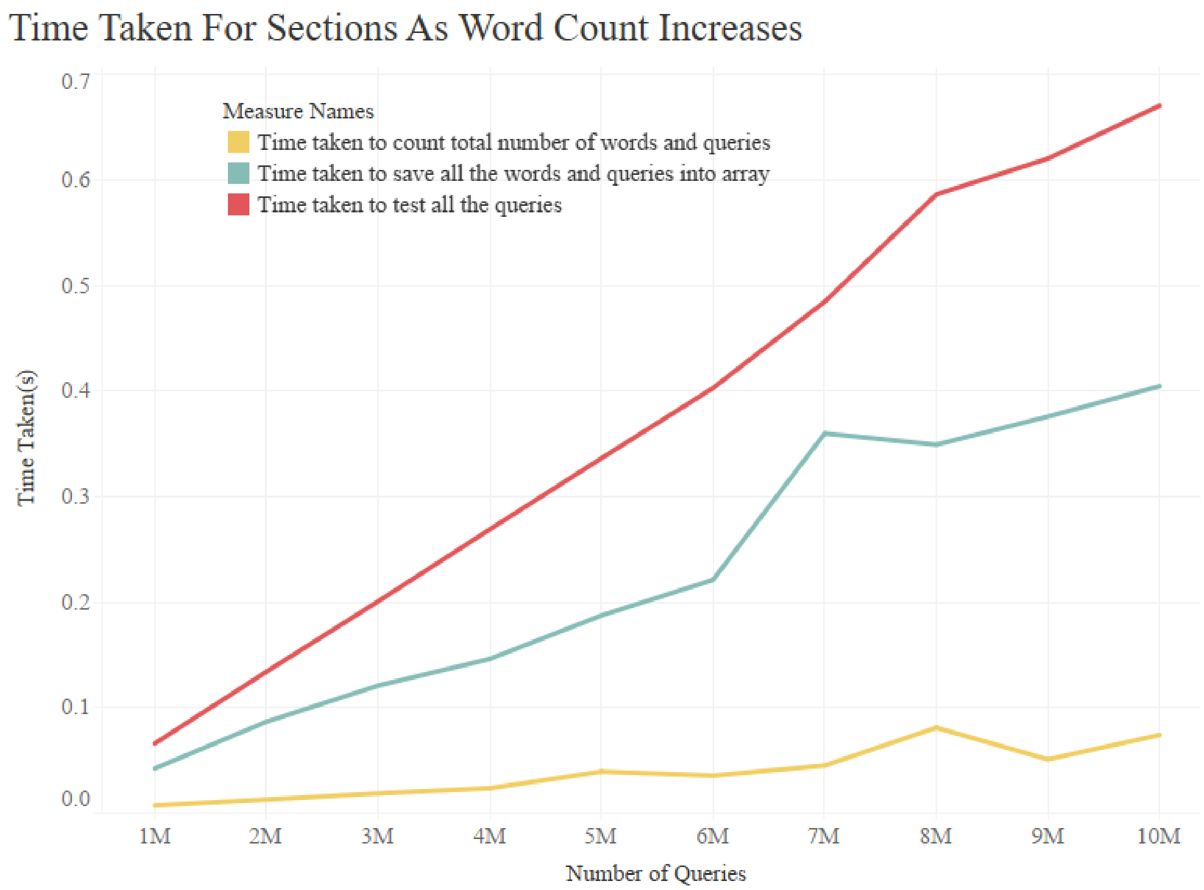


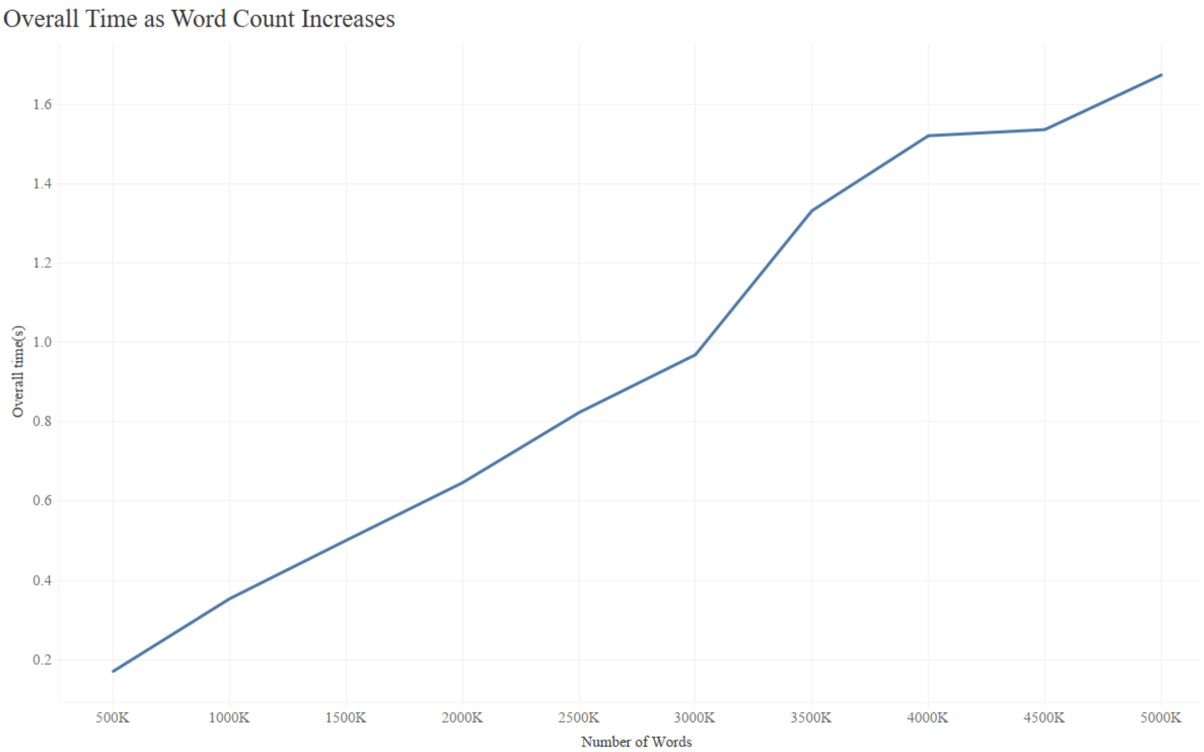
Performance on a CAAS (Parallel)

Number of words in word list	Number of words in query list	True Positive Count	False Positive Count	Accuracy
500000	1000000	500000	25080	0.97492
1000000	2000000	1000000	50355	0.974823
1500000	3000000	1500000	75938	0.974687
2000000	4000000	2000000	100675	0.974831
2500000	5000000	2500000	126451	0.97471
3000000	6000000	3000000	151146	0.974809
3500000	7000000	3500000	176422	0.974797
4000000	8000000	4000000	201731	0.974784
4500000	9000000	4500000	226586	0.974824

Number of rows in input	Number of rows in query	Time taken to count total number of words and queries	Time taken to allocate memory for bit array and non parallelizebale section	Time taken to save all the words and queries into array	Time taken to insert all the words into bit array	Time taken to test all the queries	Overall time(s)	Time taken to count total number of words and queries	Time taken to allocate memory for bit array and non parallelizebale section
500000	1000000	0.007349	0.000004	0.042381	0.054793	0.066028	0.17058	0.007349	0.000004
1000000	2000000	0.012897	0.000005	0.086659	0.120028	0.134056	0.353678	0.012897	0.000005
1500000	3000000	0.018741	0.000004	0.120883	0.16036	0.200711	0.500728	0.018741	0.000004
2000000	4000000	0.023591	0.000004	0.146157	0.207598	0.268937	0.646318	0.023591	0.000004
2500000	5000000	0.03939	0.000005	0.187339	0.260042	0.336339	0.823145	0.03939	0.000005
3000000	6000000	0.035622	0.000006	0.221074	0.309327	0.402918	0.968981	0.035622	0.000006
3500000	7000000	0.04514	0.000015	0.359853	0.44242	0.484656	1.332142	0.04514	0.000015
4000000	8000000	0.081034	0.000016	0.349235	0.504583	0.586382	1.521307	0.081034	0.000016
4500000	9000000	0.051009	0.000008	0.375896	0.489415	0.620363	1.53673	0.051009	0.000008




Time Taken For Sections As Word Count Increases
Measure Names
Time taken to count total number of words and queries
Time taken to save all the words and queries into array
Time taken to test all the queries
Time Taken(s)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1M
2M
3M
4M
5M
6M
7M
8M
9M
10M
Number of Queries


Overall Time as Word Count Increases
Overall time(s)
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
500K
1000K
1500K
2000K
2500K
3000K
3500K
4000K
4500K
5000K
Number of Words

By comparing the performance of parallel and serial algorithm on both single computer and CAAS, we can observe that the accuracy are the same on all of them. This is because the algorithm for both parallel and serial handles all the input the same way without compromising.

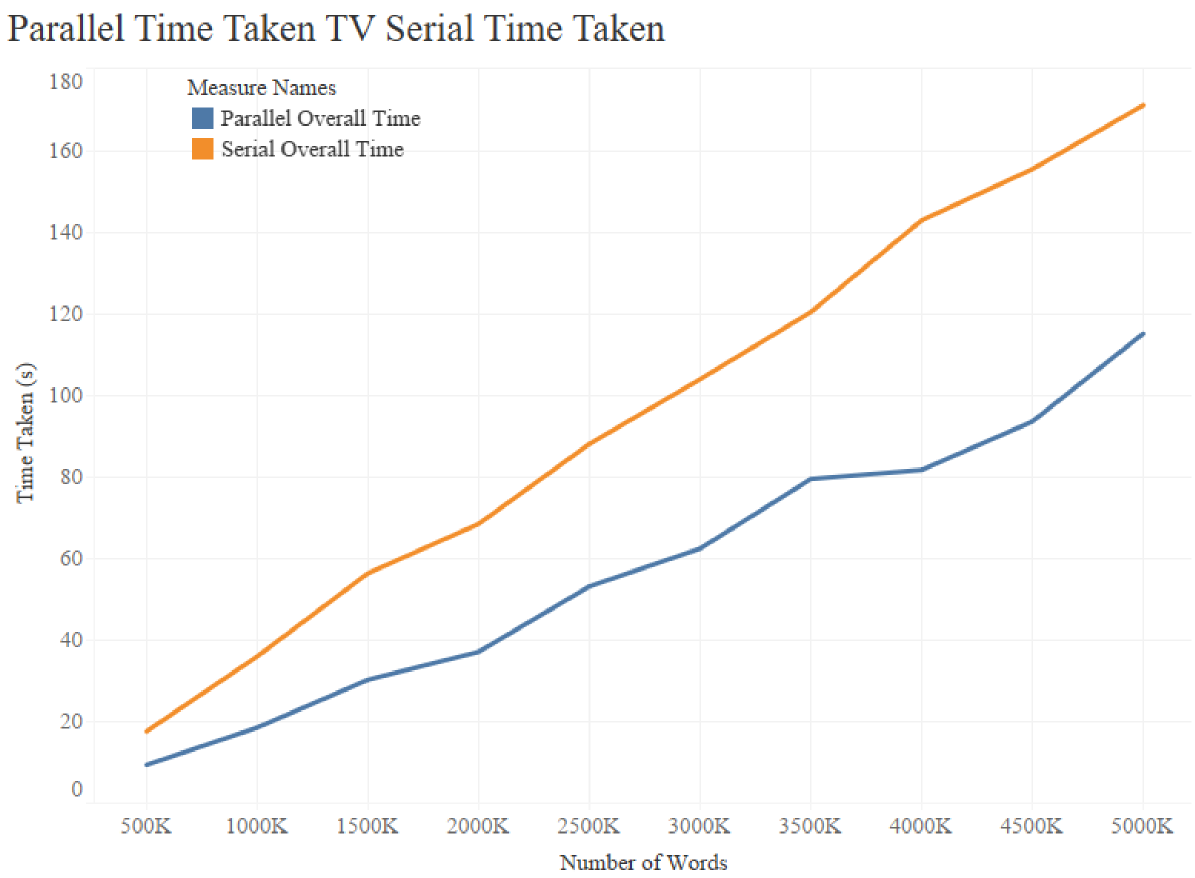
For the time taken, we can observe that CAAS takes significantly shorter time than single computer. And the difference in parallel and serial is also greater for CAAS. For all the executions, sections that involves file reading takes the most amount of time. This includes counting number of words and queries, and saving all the words and queries into array.

For the trend as input size increases, we can observe from the chart that as word count and query count increases, the time taken to process them increases for both parallel and serial algorithm. However, the time taken to allocate memory for bit array, which is the non-parallelizable part remains the same.

Calculation Of Actual Speed Up

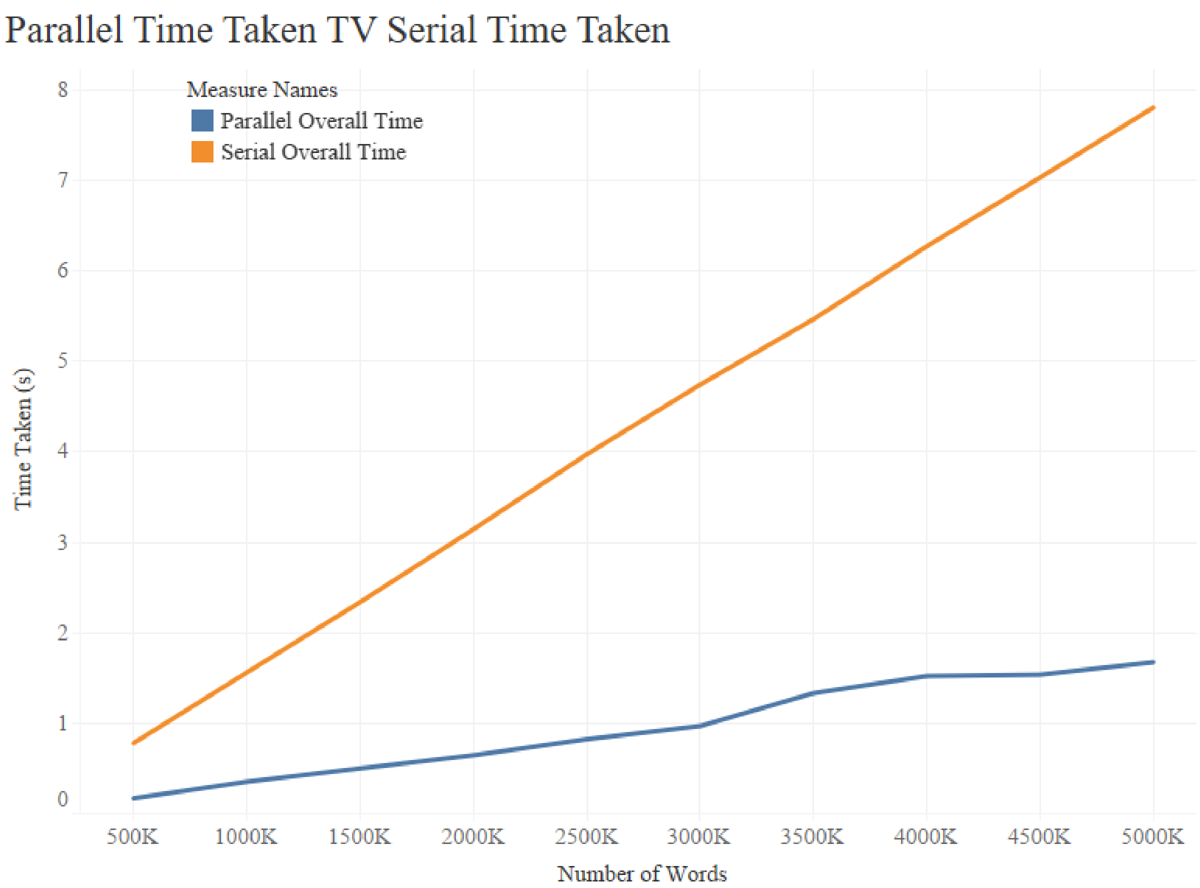
Actual speed up on single computer (My personal laptop)

Number of rows in input	Number of rows in query	Serial Overall Time	Parallel Overall Time	Actual Speed Up
500000	1000000	17.6075	9.383292	1.876474
1000000	2000000	35.98573	18.54175	1.940794
1500000	3000000	56.36699	30.26452	1.862478
2000000	4000000	68.46419	37.11872	1.844465
2500000	5000000	88.0595	53.14425	1.65699
3000000	6000000	103.9128	62.42285	1.664659
3500000	7000000	120.3968	79.47834	1.514838
4000000	8000000	142.8648	81.69068	1.748851
4500000	9000000	155.4108	93.56153	1.661055
5000000	10000000	171.055	115.0913	1.486255
			Average Speedup:	1.725686



Actual Speedup On CAAS

Number of rows in input	Number of rows in query	Serial Overall Time	Parallel Overall Time	Actual SpeedUp
500000	1000000	0.780614	0.17058	4.576234
1000000	2000000	1.562024	0.353678	4.416514
1500000	3000000	2.338991	0.500728	4.671181
2000000	4000000	3.144128	0.646318	4.864677
2500000	5000000	3.971957	0.823145	4.825343
3000000	6000000	4.741418	0.968981	4.8932
3500000	7000000	5.464768	1.332142	4.102241
4000000	8000000	6.270144	1.521307	4.121551
4500000	9000000	7.030017	1.53673	4.57466
5000000	10000000	7.803897	1.674347	4.66086
			Average Speedup	4.570646



There is a difference between the theoretical speedup and actual speedup. The theoretical speedup is 8 and the actual speedup is 1.73 in average for local computer, and 4.57 in average for CAAS. One of the reasons that caused this difference is the theoretical speedup does not consider the overhead of the thread's creation, but the actual speedup is affected by the overhead of the threads.

Second reason is the usage of syntax like atomic and reduction, which will increase the time taken for computation as it prevents other threads from accessing the value which causes serialization of the execution, reduction also added some additional computational time for the extra computation for reduction.

Third reason is the unbalanced workload of the parallel region. Query files are double the size of word files, this could cause the parallelization of reading of the files have unbalanced workload. For example, if word files have length of 100, then query files will have the length of 200. Hence, the other threads will have to wait for the threads that handles query files to end to proceed.

We can observe that the difference of theoretical speedup and actual speedup in local single computer is bigger. This could be due to the limitation of my laptop, for example the computing speed and the speed of input and output. The difference is smaller for the CAAS as it has stronger computing power.

Bibliography

[1] (No date) *Hash Functions*. Available at: http://www.cse.yorku.ca/~oz/hash.html (Accessed: 10 September 2023).

[2] Pieters, M. and Pochmann, S. (2018) *Fastest Way to generate a random-like unique string with random length in python 3*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/48421142/fastest-way-to-generate-a-random-like-unique-string-with-random-length-in-python (Accessed: 10 September 2023).

Appendix

```
import csv
import secrets
import numpy as np
import string
from functools import partial

np.random.seed(0)
def produce_amount_keys(amount_of_keys, _randint=np.random.randint):
# Source: https://stackoverflow.com/questions/48421142/fastest-way-to-generate-a-random-like-
unique-string-with-random-length-in-python
    keys = set()
    pickchar = partial(secrets.choice, string.ascii_uppercase + string.digits)
    while len(keys) < amount_of_keys:
        keys |= {''.join([pickchar() for _ in range(_randint(12, 20))]) for _ in range(amount_of_keys -
len(keys))}
    return keys

queryLength = 1000000
inputLength = queryLength//2
arr = list(produce_amount_keys(queryLength))
inputfile1 = open("word1.csv", "w",newline='')
inputfile2 = open("word2.csv", "w",newline='')
inputfile3 = open("word3.csv", "w",newline='')
inputfile4 = open("word4.csv", "w",newline='')

queryfile1 = open("query1.csv", "w",newline='')
queryfile2 = open("query2.csv", "w",newline='')
queryfile3 = open("query3.csv", "w",newline='')
queryfile4 = open("query4.csv", "w",newline='')

writeInput1 = csv.writer(inputfile1,escapechar = "\\")
writeInput2 = csv.writer(inputfile2,escapechar = "\\")
writeInput3 = csv.writer(inputfile3,escapechar = "\\")
writeInput4 = csv.writer(inputfile4,escapechar = "\\")
writeQuery1 = csv.writer(queryfile1,escapechar = "\\")
writeQuery2 = csv.writer(queryfile2,escapechar = "\\")
writeQuery3 = csv.writer(queryfile3,escapechar = "\\")
writeQuery4 = csv.writer(queryfile4,escapechar = "\\")

writeInput1.writerow(["Value"])
writeInput2.writerow(["Value"])
writeInput3.writerow(["Value"])
writeInput4.writerow(["Value"])

writeQuery1.writerow(["Value"]+["Exist"])
writeQuery2.writerow(["Value"]+["Exist"])
writeQuery3.writerow(["Value"]+["Exist"])
writeQuery4.writerow(["Value"]+["Exist"])
```

```
for item in arr[:inputLength//4]:
    writeInput1.writerow([item])
    writeQuery1.writerow([item]+[1])

for item in arr[inputLength//4:inputLength//2]:
    writeInput2.writerow([item])
    writeQuery2.writerow([item]+[1])

for item in arr[inputLength//2:(inputLength//4)*3]:
    writeInput3.writerow([item])
    writeQuery3.writerow([item]+[1])

for item in arr[(3*inputLength//4):inputLength]:
    writeInput4.writerow([item])
    writeQuery4.writerow([item]+[1])

for item in arr[inputLength:((queryLength-inputLength)//4)+inputLength]:
    writeQuery1.writerow([item]+[0])

for item in arr[((queryLength-inputLength)//4)+inputLength:(queryLength-
inputLength)//2+inputLength]:
    writeQuery2.writerow([item]+[0])

for item in arr[(queryLength-inputLength)//2+inputLength:(3*(queryLength-
inputLength))//4+inputLength]:
    writeQuery3.writerow([item]+[0])


for item in arr[(3*(queryLength-inputLength))//4+inputLength:]:
    writeQuery4.writerow([item]+[0])

inputfile1.close()
inputfile2.close()
inputfile3.close()
inputfile4.close()
queryfile1.close()
queryfile2.close()
queryfile3.close()
queryfile4.close()
```