Name: Bryan Jun Kit Wong
Student ID: 32882424

Introduction

The Bloom Filter is a widely used, space-efficient probabilistic data structure that quickly checks whether an element belongs to a set. It uses hashing to set bits in a bitarray and, upon query, checks if all corresponding bits are set to one. If so, it belongs to a set. However, false positives can occur where all bits are set to one despite the word not existing in the set. Key features include space efficiency, fast membership testing, and controllable false positives.

The selection of the Bloom Filter for this assignment is based on its efficiency in string matching tasks. It excels in rapid searching, space optimisation and the one downside it has (potential false positives) can be minimised at the expense of more memory usage.

Pseudocode (Serial)

```
Function readFile(filename, fileLength, pUniqueWordsList) {
   Open file for reading
   Allocate memory to store the number of unique words with size fileLength

   While (attempt to read a string from the file and it succeeds) {
      Loop through pUniqueWords array to check if word does not currently exist {
         If exist, isUnique boolean flag will be false to signify it's not unique.
      }
      If unique, duplicate the word and store it in array and increment count by one
   }

   Close the file
   Reallocate memory for pUniqueWords array and resize it
   Set the input pointer to point towards the pUniqueWords array to return it to caller

   Return count (number of unique words)
}
```

This function opens the file for reading, allocating memory to store the number of unique words in the file. It reads each word from the file, checking if it currently exists within the array of unique words. If it doesn't, add it to said array and increment the count of unique words by one. The file is then close, and the counts are return back to the caller.

Name: Bryan Jun Kit Wong
Student ID: 32882424

```
Function initialiseBitArray(int n, int m, int **bitarray) {
    Calculate the fpr based on input parameters
    Calculate optimum m based on fpr, n and m
    Allocate memory for bitarray based on optimum m and fill it with zeroes.
}
```

This function calculates the false positive rate (fpr) based on input parameters, using that to calculate the optimum size of the bit array (m). Then, it allocates memory for the bitarray of size m, initialising all its elements as zeroes.

```
Function insertKey(const char* key, int m, int *bitarray) {
    Get hash value from Jenkin Hash Function
    Get hash value from Fowler Noll Vo Hash Function
    Get hash value from DJ B2 Hash Function


    Modulo the hash values by m to get the index then set all elements at the index to one
}
```

This function is used to insert a word into the bloom filter. It employs 3 hash functions (Jenkin, FNV and DJB2) to transform the word into numerical values. These values are then modulo by m to determine their corresponding index in the bitarray. Said bit at that index will be set to one, indicating the existence of the word.

```
Function lookUpKey(const char* key, int m, int *bitarray) {
    Get hash value from Jenkin Hash Function
    Get hash value from Fowler Noll Vo Hash Function
    Get hash value from DJ B2 Hash Function


    Modulo the hash values by m then check if all the elements at the index are one
    If they are, return true. Else return false as values not in bitarray set
}
```

This function checks if a given word exists within the bloom filter bitarray. It uses the same hash functions used in the insertKey method to generate index values. If all the bits at the corresponding indices are set to one, the function returns true, indicating the word might exists.

Name: Bryan Jun Kit Wong
Student ID: 32882424

```
Function query(const char* filename, int m, int *bitarray) {
    Open query.txt file to read
    Open output.txt file to write

   While reading word from query.txt {
        Call the lookup method and the word as input

        While looping each word in totalUnique array and word does not exist in array {
            If word exist in totalUnique array {
                hasFound True and exit loop.
            }
        }

        If word does not exist in array but exist in bitarray {
                Increment False Positive count by one
        }
        Write the word, followed by 1 if probably found and 0 if not to output.txt
    }
    Close both files
    Remove "query.txt" and renamed "output.txt" to "query.txt"
    Return False Positive count
}
```

This function processes a query file, checking each word within in whether it exists in the bloom filter, keeping track of any false positives found. The results are then outputted into another file, with 1 to indicate it might exist and 0 for the opposite.

Name: Bryan Jun Kit Wong
Student ID: 32882424

```
Function int main() {
    For each file in files {
        Input each file into the readFile method to be read
        Calculate the total unique number of words in all the files
    }

    For unique word in unique words {
        Call the insertKey method to insert each unique word into the bitarray
    }

    Query the dataset by calling the query method
    Calculate accuracy based on number of False Positives
    Free all allocated memory
}
```

This function loops through all dataset files and calls the readFile method on each file to get the total number of unique words. It then initialises the bitarray using the initaliseBitArray method and input the appropriate parameters. Each unique word is then inserted into the bloom filter using the insertKey method. Lastly, the query method is called to query the dataset, getting the number of false positives to determine the accuracy rate.

Datasets Used

Three Datasets used for inserting taken from Project Gutenberg (Moby Dick, Shakespeare and Little Women). Total word count is 1376656 words with 71846 of them being unique. Three query texts are used for querying– small (25000 words), medium (50000 words) and large (75000 words), where each query is a single word of less than 100 characters.

Performance (Serial)

The datasets listed above are used by the serial bloom filter code, and performance is measured via computational time taken. The bloom filter is ran 10 times, with the average time taken being measured.

Name: Bryan Jun Kit Wong
Student ID: 32882424

One dataset (215724 words)

	Reading and Counting	Inserting
Time Taken 1	12.16801	0.008367
Time Taken 2	11.801879	0.007819
Time Taken 3	11.069913	0.007483
Time Taken 4	11.560013	0.007267
Time Taken 5	11.71004	0.007483
Time Taken 6	10.835744	0.007791
Time Taken 7	11.477115	0.006353
Time Taken 8	11.959758	0.007477
Time Taken 9	11.994698	0.007258
Time Taken 10	11.701903	0.007139
Average Time Taken	11.6279073	0.007444

Two datasets (411191 words)

	Reading and Counting	Inserting
Time Taken 1	11.620884	0.008003
Time Taken 2	11.822693	0.008008
Time Taken 3	11.829624	0.007071
Time Taken 4	12.745013	0.007189
Time Taken 5	11.927992	0.008008
Time Taken 6	10.915301	0.007114
Time Taken 7	11.329631	0.009219
Time Taken 8	11.96675	0.007311
Time Taken 9	11.955951	0.007795
Time Taken 10	12.099197	0.008045
Average Time Taken	11.8213036	0.0077763

Three datasets (1376656 words)

	Reading and Counting	Inserting
Time Taken 1	12.993423	0.007917
Time Taken 2	12.007965	0.007637
Time Taken 3	11.815588	0.008992
Time Taken 4	12.491248	0.008148
Time Taken 5	12.488259	0.009177
Time Taken 6	11.115581	0.007763
Time Taken 7	12.101608	0.010812
Time Taken 8	13.263607	0.008163
Time Taken 9	11.730672	0.007345
Time Taken 10	12.06686	0.008045
Average Time Taken	12.2074811	0.0083999

Name: Bryan Jun Kit Wong
Student ID: 32882424

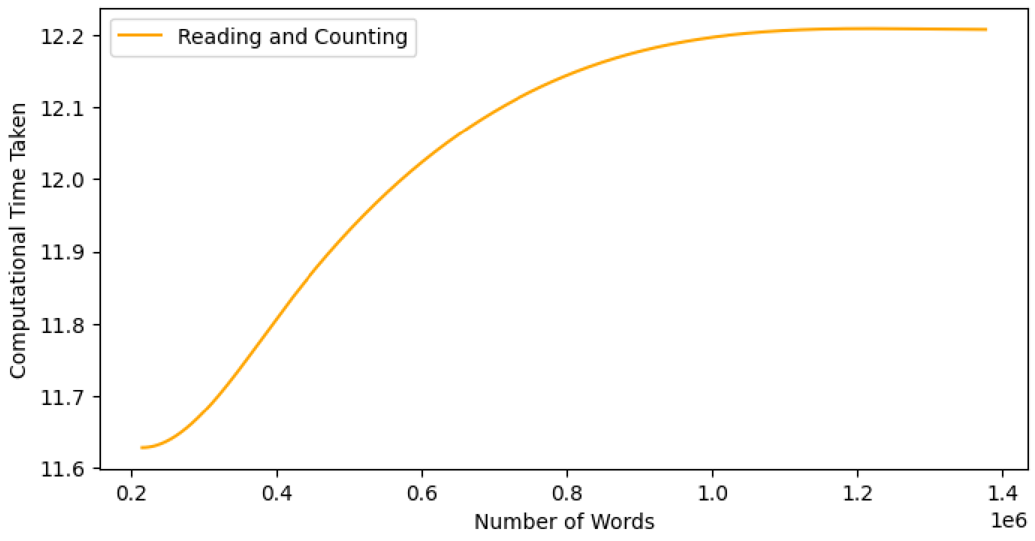

Figure 1.1: Computational Time for Reading and Counting against Number of Words

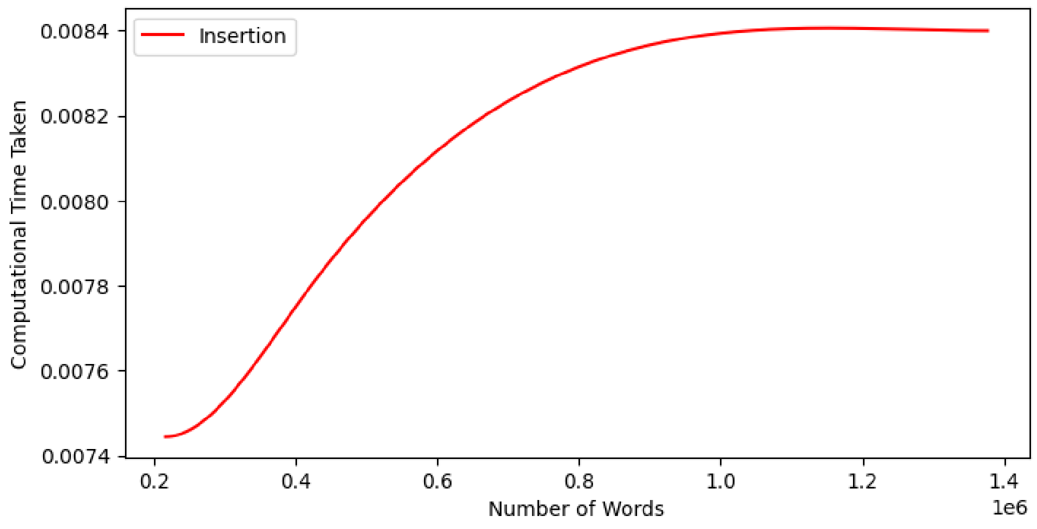

Figure 1.2: Computational Time for Inserting into Bloom Filter against Number of Words

Name: Bryan Jun Kit Wong
Student ID: 32882424

Small query lookup list used (25000 words)

	Querying/Lookup
Time Taken 1	2.422799
Time Taken 2	2.592862
Time Taken 3	2.470639
Time Taken 4	2.560444
Time Taken 5	2.468718
Time Taken 6	2.389193
Time Taken 7	2.671578
Time Taken 8	2.567877
Time Taken 9	2.580404
Time Taken 10	2.404499
Average Time Taken	2.5129013

Medium query lookup list used (50000 words)

	Querying/Lookup
Time Taken 1	2.423919
Time Taken 2	2.773453
Time Taken 3	2.416144
Time Taken 4	2.483036
Time Taken 5	2.421618
Time Taken 6	2.516638
Time Taken 7	2.561567
Time Taken 8	2.7563
Time Taken 9	2.662683
Time Taken 10	2.912589
Average Time Taken	2.5927947

Large query lookup list used (75000 words)

	Querying/Lookup
Time Taken 1	2.465438
Time Taken 2	2.658484
Time Taken 3	2.532034
Time Taken 4	2.683036
Time Taken 5	2.554575
Time Taken 6	2.501917
Time Taken 7	2.900025
Time Taken 8	2.568905
Time Taken 9	2.639285
Time Taken 10	2.740955
Average Time Taken	2.6244654

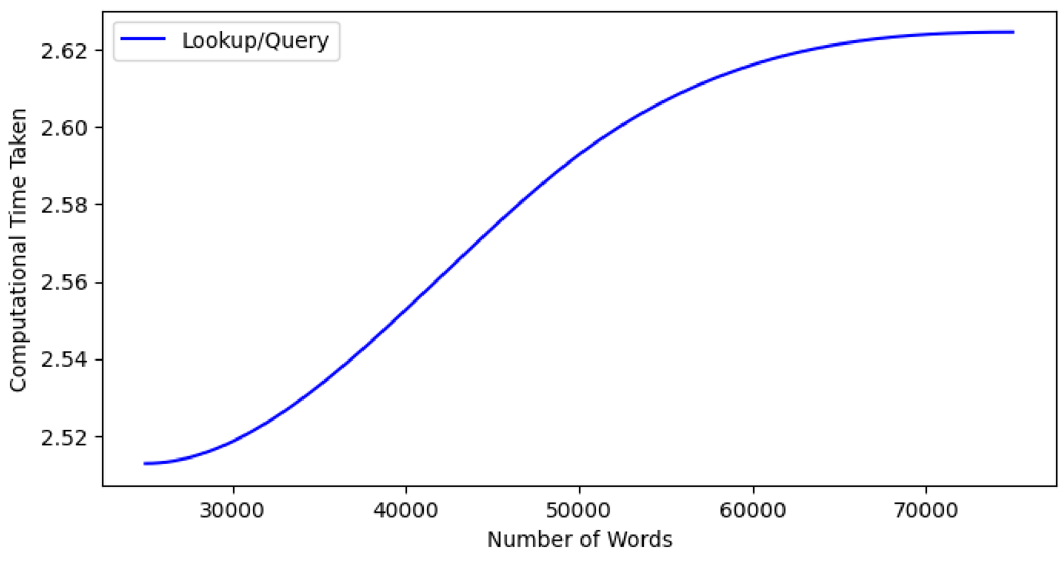

Figure 1.3: Computational Time for querying words from Bloom Filter against Number of Words

The Serial Bloom Filter boast an impressive accuracy of 99.999427 for the three datasets and large query file. On average, the overall processing time is 15.2 seconds with the reading and counting number of unique words (readFile function) taking the longest at an average of 12.2 seconds. This is because the input file is exceptionally large at 1 million words, thus reading and processing it can be time-consuming. Conversely, insertion is remarkably efficient at 0.008 seconds, as it exclusively deals with unique words, which are a small fraction of the total word count. Lastly, the querying took an average of 2.62 seconds as the False Positives have to be identified and counted, requiring additional searching to check if the word actually exist in the dataset or not.

Dependency Analysis

The reading and counting unique number of words operation reads words from a given file, counting the number of unique words, and storing them into a separate array. Using Bernstein's condition, we can determine if this operation is parallelisable. There are no Read-Write conflicts as the reading operations are independent from any writing operations. There are no Write-Write conflicts as each iteration will write the unique words to a independent part of the array and there are no Write-Read conflicts.

For the inserting operation, there are no Read-Write or Write-Read conflicts. For Write-Write, although all threads may write to the same location in the bitarray, the result will always remain the same which is 1 or 0, thus implying that this operation can be parallelised.

For the querying/lookup operation, there should be no Read-Write conflicts due to no medication being made on the original read file. Furthermore, there are no Write-Write conflicts as each iteration will be independent of each other when writing to output file, suggesting that the operation can be parallelised. There are no Write-Read conflicts.

Name: Bryan Jun Kit Wong
Student ID: 32882424

Theoretical Speed Up

To calculate the theoretical speed up of the Parallel Bloom Filter, Amdahl’s Law can be used.

Amdahl’s Law:

$$\frac{1}{r_s + \frac{r_p}{p}}$$

Where r_s is the serial ratio, r_p is the parallel ration and p is the number of processors. The overall average time taken of the Serial Bloom Filter for a fix dataset size of 1376656 words is 15.2 seconds. Therefore,

$$S(8) = \frac{1}{\frac{0.36}{15.2} + \frac{\frac{14.8}{15.2}}{8}} = 6.9 \approx 7$$

Based on the equation above, theoretically, a 7x speedup can be hypothetically observed with 8 processors.

Name: Bryan Jun Kit Wong
Student ID: 32882424

Pseudocode (Parallel)

```
Function readFile(filename, fileLength, pUniqueWordsList) {
    Open file for reading
    Allocate memory to store the number of unique words with size fileLength
    #pragma omp parallel
    {
        Declare private versions of unique word array and unique word count.
        #pragma omp for nowait private(j, isUnique, word)
        For loop to loop through the entire file {
            Loop through private array to check if word does not currently exist {
                If exist, isUnique boolean flag will be false to signify it's not unique.
            }
            If unique, store it in private array and increment private count by one
        }

        #pragma omp critical
        {
            For loop to loop through entire private array {
                Loop through global array to check if private word does not currently exist {
                    If exist, isUnique boolean flag will be false to signify it's not unique.
                }
                If unique, store it in global array and increment global count by one
            }
            Free up memory allocated for  private arrays
        }
    }
    Close the file
    Reallocate memory for pUniqueWords array and resize it
    Set the input pointer to point towards the pUniqueWords array to return it to caller
    Return count (number of unique words)
}
```

Name: Bryan Jun Kit Wong
Student ID: 32882424

This function opens the file for reading, allocating memory to store the number of unique words in the file. OpenMP is used to parallelise the code. Then, it enters a parallel section denoted by #pragma omp parallel. Within it, each thread maintains private word lists and private unique word counts. The for loop has a #pragma omp for nowait private(j, isUnique, word) header to parallelise the loop. The nowait clause is used to optimise and improve efficiency as threads can immediately execute the critical section below without waiting, reducing overhead. The private keyword is used to declare local versions of the listed variables for each thread. Despite being slower due to communication overhead, the #pragma omp critical section is crucial to maintain data consistency, synchronising the unique words in the private array with the global shared array safely, preventing race conditions or data corruption.

```
Function query(const char* filename, int m, int *bitarray) {
   Open query.txt file to read
   Open output.txt file to write


   #pragma omp parallel for private(found, hasFound, j, result, word)
reduction(+:falsePositiveCount)
  While reading word from query.txt {
      Call the lookup method and the word as input
      While looping each word in totalUnique array and word does not exist in array {
        If word exists in totalUnique array, set hasFound to true
      }
      If word does not exist in array but exist in bitarray set, increment False Positive count


      #pragma omp critical
      {
         Write the word, followed by 1 if probably found and 0 if not to output.txt
      }
   }
   Close both files
   Remove “query.txt” and renamed “output.txt” to “query.txt”


   Return False Positive count
}
```

This function processes a query file by opening it and reading it. Within the parallel processing loop, it reads the words from the input file concurrently using OpenMP, performing a lookup using the lookupKey method while simultaneously keeping track of the number of false positives. The parallel loop #pragma omp parallel for private(found, hasFound, j, result, word) reduction(+:falsePositiveCount) is used to read the query and write concurrently. Numerous private variables are declared to prevent race conditions from occurring by keeping the variables independent of each thread. The reduction operation is used to combine the threads' local falsePositiveCount variable into a single one. This is a common technique to efficiently calculate sums or other aggregations in parallel without the need for explicit synchronisation like using a critical block, improving and optimising the performance of the parallel code.

```
Function int main() {
   For each file in files {
      Input each file into the readFile method to be read
      Calculate the total unique number of words in all the files
   }

   #pragma omp parallel for
   For unique word in unique words {
      Call the insertKey method to insert each unique word into the bitarray
   }

   Query the dataset by calling the query method
   Calculate accuracy based on number of False Positives
   Free all allocated memory
}
```

This function is virtually similar to the serial version, with one notable exception. The #pragma omp parallel for header is used to insert all unique words from the totalUnique array into the bitarray in parallel, splitting the task between multiple threads. One key observation is the for loop for readFile is lacking any parallelisation on it. This is because the readFile method itself is already parallelised nd parallelisation of the outer loop reduces performance due to introducing overhead. Lastly, the hash functions are not parallelised as each iteration, the hash values are dependent on the value of the last iteration, making parallelisation virtually impossible as it could lead to race conditions.

Name: Bryan Jun Kit Wong
Student ID: 32882424

Performance (Parallel on Single)

One dataset (215724 words)

	Reading and Counting	Inserting
Time Taken 1	1.45395	0.001694
Time Taken 2	1.387946	0.001979
Time Taken 3	1.447661	0.002739
Time Taken 4	1.422369	0.003075
Time Taken 5	1.462923	0.001034
Time Taken 6	1.447074	0.004664
Time Taken 7	1.447822	0.002252
Time Taken 8	1.444256	0.001261
Time Taken 9	1.444256	0.003656
Time Taken 10	1.432425	0.002937
Average Time Taken	1.4390682	0.002529

Two datasets (411191 words)

	Reading and Counting	Inserting
Time Taken 1	2.055971	0.001236
Time Taken 2	2.019614	0.002752
Time Taken 3	2.021234	0.004376
Time Taken 4	2.06636	0.003403
Time Taken 5	2.054308	0.002241
Time Taken 6	2.059617	0.002484
Time Taken 7	2.06763	0.001105
Time Taken 8	2.071841	0.002067
Time Taken 9	2.032535	0.003782
Time Taken 10	2.086497	0.005286
Average Time Taken	2.0535607	0.0028732

Three datasets (1376656 words)

	Reading and Counting	Inserting
Time Taken 1	6.868233	0.003596
Time Taken 2	6.621404	0.002962
Time Taken 3	6.755877	0.002546
Time Taken 4	6.701287	0.002756
Time Taken 5	6.625117	0.007627
Time Taken 6	6.725362	0.002832
Time Taken 7	6.581827	0.004091
Time Taken 8	6.602132	0.007977
Time Taken 9	6.640983	0.004347
Time Taken 10	6.875076	0.003017
Average Time Taken	6.6997298	0.0041751

Name: Bryan Jun Kit Wong
Student ID: 32882424

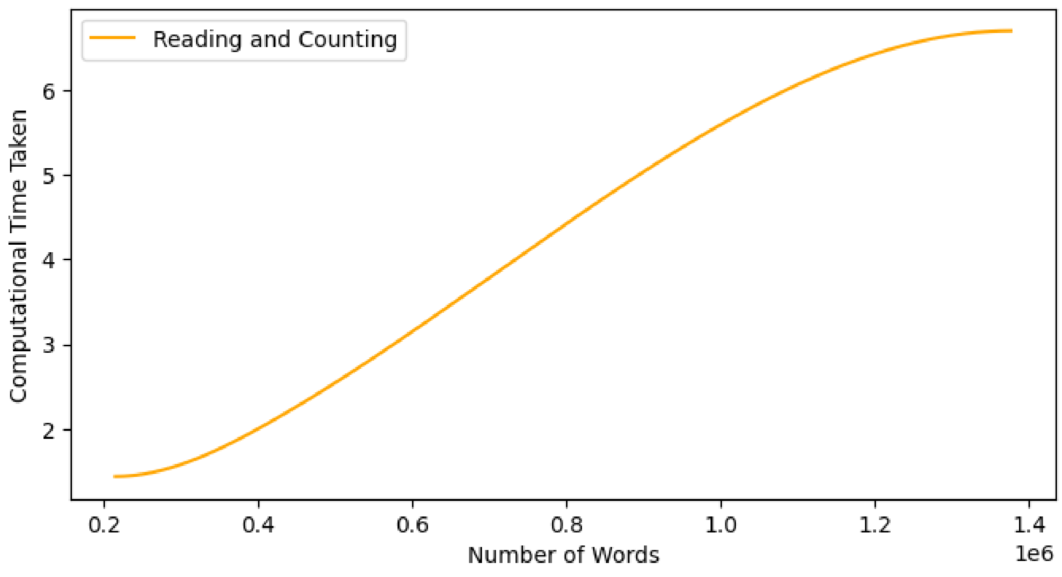

Figure 2.1: Computational Time for Reading and Counting against Number of Words on Single Computer

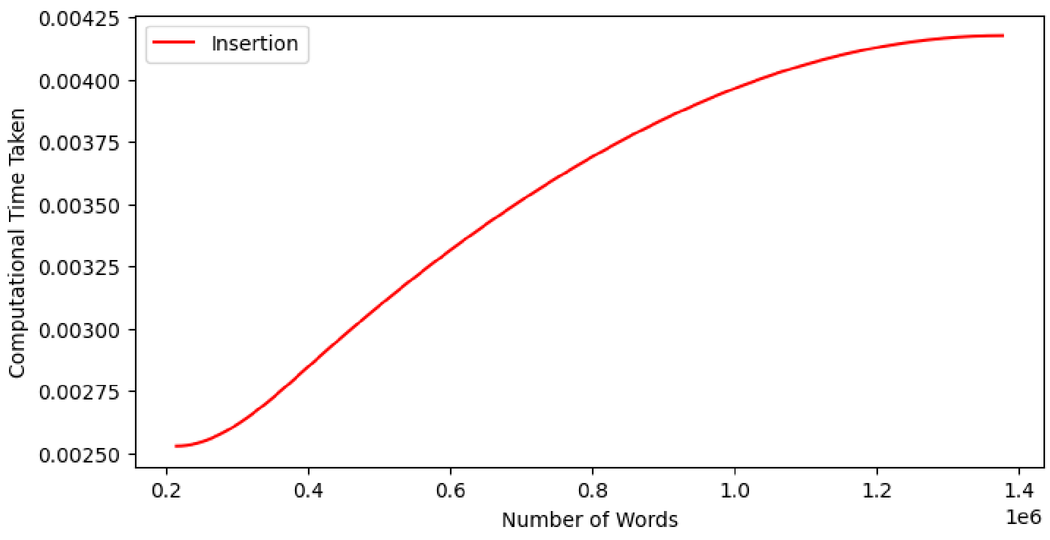

Figure 2.2: Computational Time for Inserting into Bloom Filter against Number of Words on Single Computer

Name: Bryan Jun Kit Wong
Student ID: 32882424

Small query lookup list used (25000 words)

	Querying/Lookup
Time Taken 1	0.172737
Time Taken 2	0.179036
Time Taken 3	0.180802
Time Taken 4	0.173709
Time Taken 5	0.172842
Time Taken 6	0.184141
Time Taken 7	0.152254
Time Taken 8	0.191165
Time Taken 9	0.173374
Time Taken 10	0.174529
Average Time Taken	0.1754589

Medium query lookup list used (50000 words)

	Querying/Lookup
Time Taken 1	0.325317
Time Taken 2	0.31698
Time Taken 3	0.323565
Time Taken 4	0.300942
Time Taken 5	0.321905
Time Taken 6	0.355921
Time Taken 7	0.322035
Time Taken 8	0.338276
Time Taken 9	0.309096
Time Taken 10	0.322624
Average Time Taken	0.3236661

Large query lookup list used (75000 words)

	Querying/Lookup
Time Taken 1	0.441121
Time Taken 2	0.465776
Time Taken 3	0.451962
Time Taken 4	0.5034
Time Taken 5	0.481533
Time Taken 6	0.462769
Time Taken 7	0.458204
Time Taken 8	0.455289
Time Taken 9	0.471127
Time Taken 10	0.475528
Average Time Taken	0.4666709

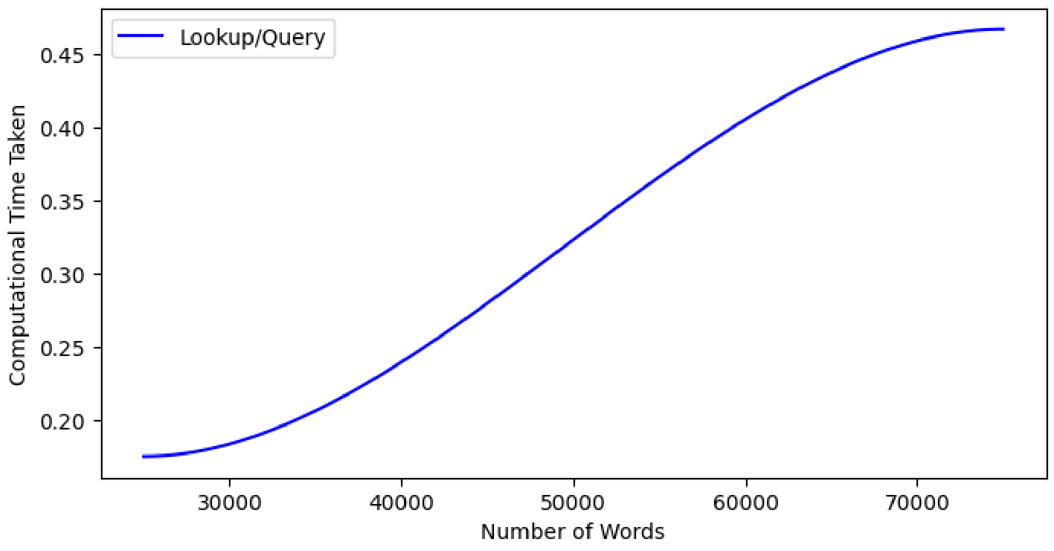

Figure 2.3: Computational Time for querying words from Bloom Filter against Number of Words on Single Computer

The parallel Bloom Filter also has accuracy of 99.999427%, meaning it did not compromise its accuracy in exchange for performance. On average, the overall processing time is reduced to 7.2 seconds The reading and counting of unique words operation (readFile function) takes an average of 6.7 seconds to run. This represents a substantial improvement compared to the serial version, which took twice as long at 15.2 seconds. The parallelised insertion operation demonstrates a 200% increase in performance, with execution time dropping from 0.008 seconds to 0.004 seconds. Likewise, the querying operation exhibits a remarkable 560% performance boost, reducing the time from 2.6 seconds to 0.4 seconds. Just like the serial code, increasing the number of words will result in an increase in time taken for reading, inserting and querying.

Name: Bryan Jun Kit Wong
Student ID: 32882424

Performance (Parallel on CAAS)

One dataset (215724 words)

	Reading and Counting	Inserting
Time Taken 1	2.162262	0.001372
Time Taken 2	2.171581	0.011234
Time Taken 3	2.148132	0.001011
Time Taken 4	2.141738	0.001285
Time Taken 5	2.143799	0.00183
Time Taken 6	2.143267	0.001726
Time Taken 7	2.203228	0.000764
Time Taken 8	2.130773	0.000537
Time Taken 9	2.159519	0.004559
Time Taken 10	2.131867	0.000503
Average Time Taken	2.1536166	0.002482

Two datasets (411191 words)

	Reading and Counting	Inserting
Time Taken 1	3.064093	0.000651
Time Taken 2	3.051282	0.000638
Time Taken 3	3.029616	0.000788
Time Taken 4	3.050996	0.000988
Time Taken 5	3.041932	0.00077
Time Taken 6	3.064187	0.00093
Time Taken 7	3.035553	0.000839
Time Taken 8	3.043511	0.000877
Time Taken 9	3.168518	0.000814
Time Taken 10	3.040681	0.021161
Average Time Taken	3.0590369	0.0028456

Three datasets (1376656 words)

	Reading and Counting	Inserting
Time Taken 1	9.782192	0.008918
Time Taken 2	9.717656	0.007455
Time Taken 3	9.802814	0.001335
Time Taken 4	9.84003	0.001349
Time Taken 5	9.827464	0.001362
Time Taken 6	9.784472	0.013157
Time Taken 7	9.85167	0.001353
Time Taken 8	9.767375	0.008519
Time Taken 9	9.819565	0.001338
Time Taken 10	9.760135	0.001535
Average Time Taken	9.7953373	0.0046321

Name: Bryan Jun Kit Wong
Student ID: 32882424

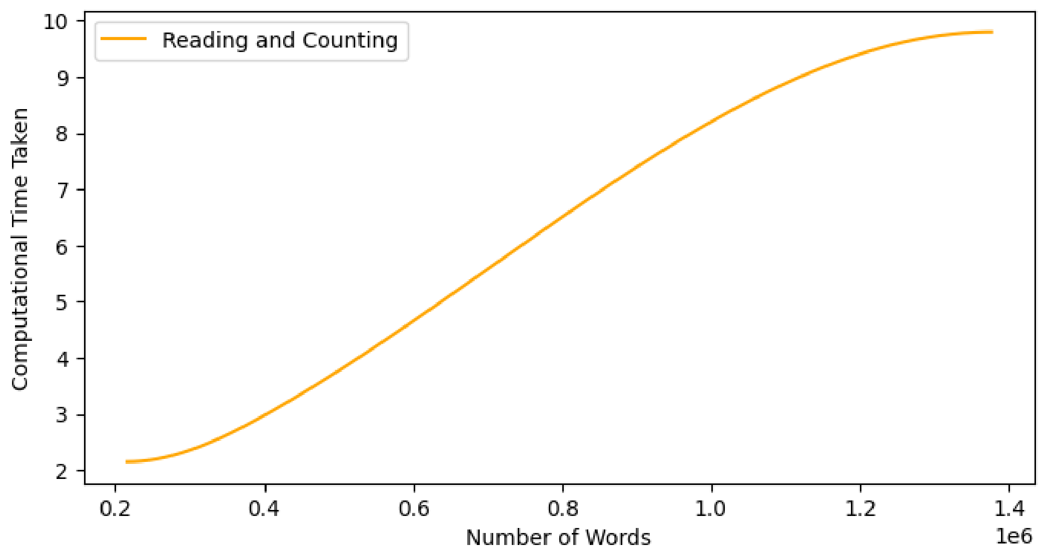

Figure 3.1: Computational Time for Reading and Counting against Number of Words on CAAS

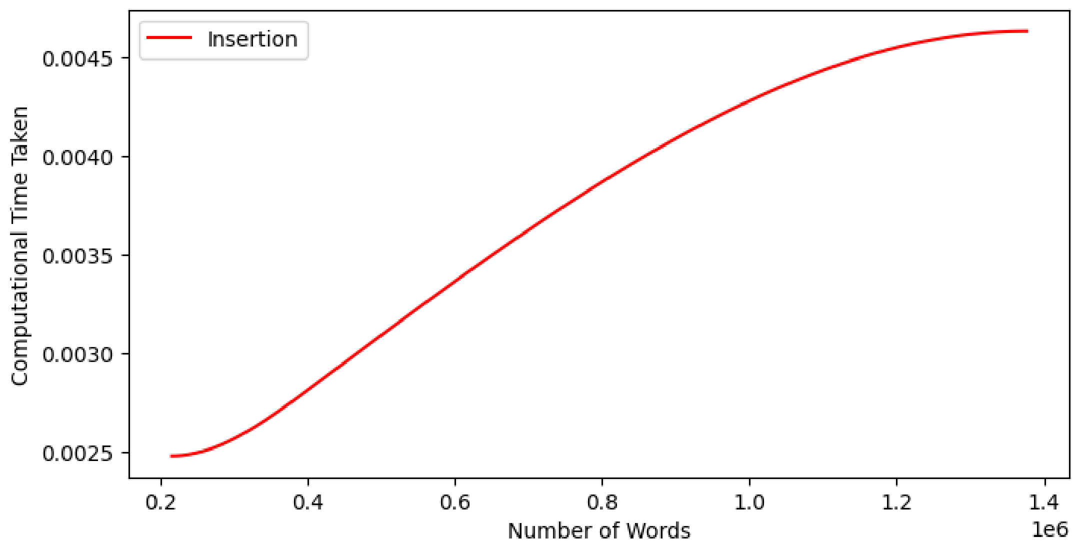

Figure 3.2: Computational Time for Inserting into Bloom Filter against Number of Words on CAAS

Name: Bryan Jun Kit Wong
Student ID: 32882424

Small query lookup list used (25000 words)

	Querying/Lookup
Time Taken 1	0.116807
Time Taken 2	0.147534
Time Taken 3	0.130196
Time Taken 4	0.107263
Time Taken 5	0.155898
Time Taken 6	0.232309
Time Taken 7	0.117259
Time Taken 8	0.116768
Time Taken 9	0.120662
Time Taken 10	0.149756
Average Time Taken	0.1394452

Medium query lookup list used (50000 words)

	Querying/Lookup
Time Taken 1	0.271443
Time Taken 2	0.30237
Time Taken 3	0.256125
Time Taken 4	0.217762
Time Taken 5	0.222323
Time Taken 6	0.326098
Time Taken 7	0.207053
Time Taken 8	0.21895
Time Taken 9	0.206087
Time Taken 10	0.3194
Average Time Taken	0.2547611

Large query lookup list used (75000 words)

	Querying/Lookup
Time Taken 1	0.462298
Time Taken 2	0.294781
Time Taken 3	0.299139
Time Taken 4	0.299347
Time Taken 5	0.306708
Time Taken 6	0.460586
Time Taken 7	0.30283
Time Taken 8	0.300084
Time Taken 9	0.301894
Time Taken 10	0.393611
Average Time Taken	0.3421278

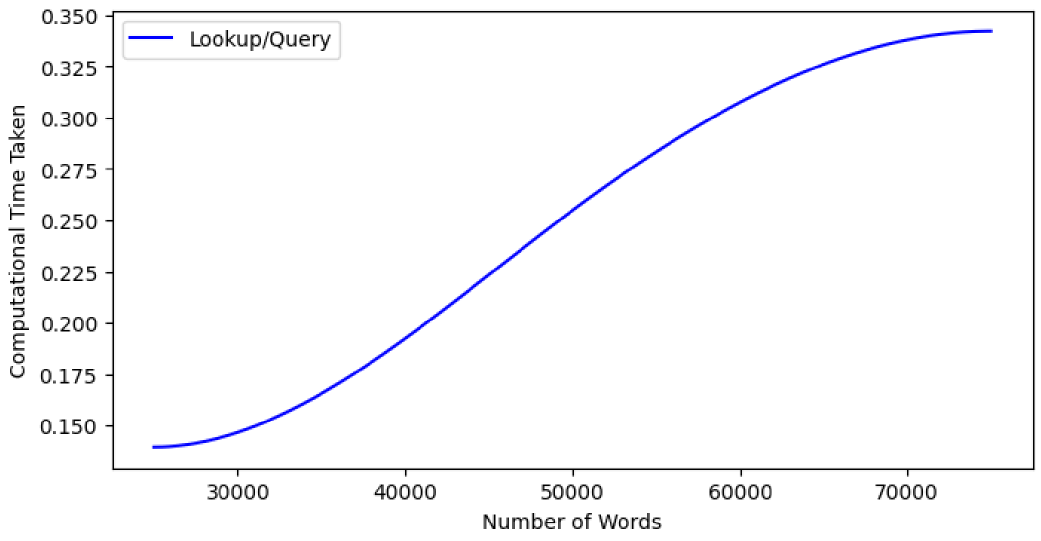

Figure 3.3: Computational Time for querying words from Bloom Filter against Number of Words on CAAS

The parallel Bloom Filter on the CAAS ran slightly slower than the parallel Bloom Filter on a single computer. The reading and counting operation take 9.5 seconds on average, which is 2.8 seconds slower. This difference in performance could be attributed to factors such as I/O latency or network latency when accessing files on the CAAS platform. Network communication overhead and remote file access can introduce delays compared to local file access on a single computer. The insertion operation is also slower by 0.00005 seconds while the querying operation, however, is faster at 0.34 seconds compared to 0.47 seconds for single computer. This could be due to differences in the underlying hardware and resource allocation on the CAAS platform, which might have better suited the parallel querying operation.

Actual Speed Up

To calculate the actual speed up:

$$S = \frac{Overall\ Serial\ Time\ Taken}{Overall\ Parallel\ Time\ Taken}$$

Name: Bryan Jun Kit Wong
Student ID: 32882424

One dataset (215724 words)

Single Computer	Overall
Time Taken 1	1.801654
Time Taken 2	1.738751
Time Taken 3	1.738618
Time Taken 4	1.731744
Time Taken 5	1.744996
Time Taken 6	1.686308
Time Taken 7	1.759041
Time Taken 8	1.743692
Time Taken 9	1.700244
Time Taken 10	1.729661
Average Time Taken	1.7374709

CAAS	Overall
Time Taken 1	2.320199
Time Taken 2	2.138326
Time Taken 3	2.339309
Time Taken 4	2.319074
Time Taken 5	2.329905
Time Taken 6	2.312797
Time Taken 7	2.311837
Time Taken 8	2.329641
Time Taken 9	2.332553
Time Taken 10	2.326116
Average Time Taken	2.3059757

Two datasets (411191 words)

Single Computer	Overall
Time Taken 1	2.398894
Time Taken 2	2.372479
Time Taken 3	2.407599
Time Taken 4	2.356815
Time Taken 5	2.424304
Time Taken 6	2.441817
Time Taken 7	2.500416
Time Taken 8	2.447385
Time Taken 9	2.442262
Time Taken 10	2.432356
Average Time Taken	2.4224327

CAAS	Overall
Time Taken 1	3.324765
Time Taken 2	3.287765
Time Taken 3	3.329279
Time Taken 4	3.268812
Time Taken 5	3.276814
Time Taken 6	3.299502
Time Taken 7	3.279978
Time Taken 8	3.31806
Time Taken 9	3.293536
Time Taken 10	3.285329
Average Time Taken	3.296384

Three datasets (1376656 words)

Single Computer	Overall
Time Taken 1	7.009168
Time Taken 2	7.814635
Time Taken 3	7.259945
Time Taken 4	7.062358
Time Taken 5	7.27675
Time Taken 6	7.161141
Time Taken 7	7.107803
Time Taken 8	7.148081
Time Taken 9	7.03936
Time Taken 10	6.819176
Average Time Taken	7.1698417

CAAS	Overall
Time Taken 1	10.098181
Time Taken 2	10.05232
Time Taken 3	10.106371
Time Taken 4	10.076144
Time Taken 5	10.113385
Time Taken 6	10.079391
Time Taken 7	10.123077
Time Taken 8	10.102404
Time Taken 9	10.095869
Time Taken 10	10.164911
Average Time Taken	10.101205

Name: Bryan Jun Kit Wong
Student ID: 32882424

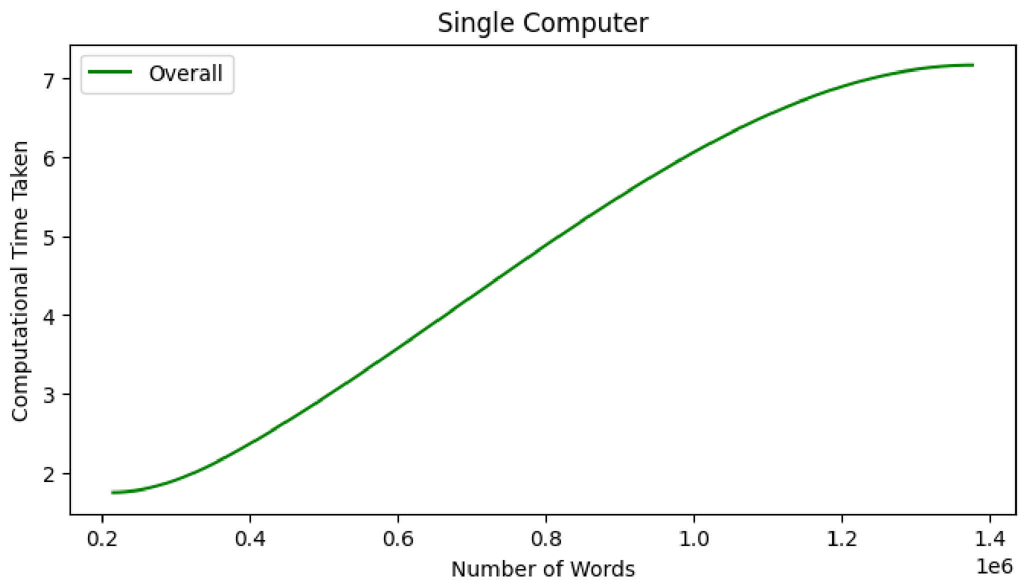

Figure 4.1: Overall Computational Time against Number of Words on Single Computer

Single Computer

$$S = \frac{15.2}{7.2} = 2.11$$

As calculated above, the actual speed up is 2x, which is a far cry from the theoretical speed of 7x. This could be due to communication or synchronisation overhead created by the critical blocks in the reading and querying sections. Additionally factors like load imbalance or contention for shared resources can contribute to the gap between theoretical and actual speedup. Both theoretical and actual speed up are sublinear speed up. Some optimisations were made to improve the actual speed up of the code, like minimising the use of critical sections by exploring alternatives like fine-grained locking, lock-free data structures or transactional memory. Other optimisations are mentioned earlier under the pseudocode section like the nowait and reduction clause.

Name: Bryan Jun Kit Wong
Student ID: 32882424

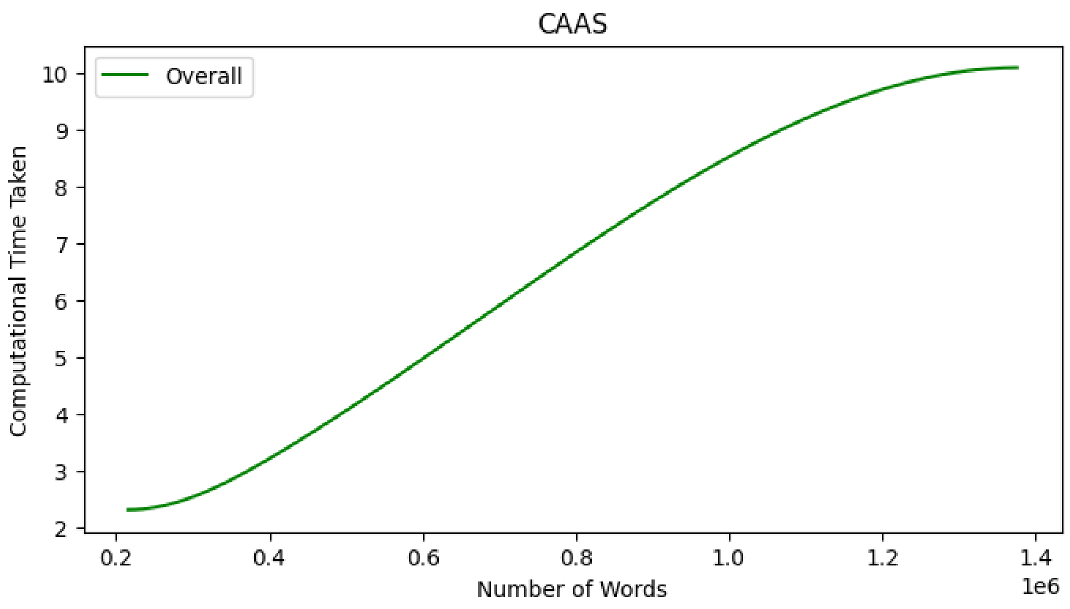

Figure 4.1: Overall Computational Time against Number of Words on CAAS

CAAS

$$S = \frac{15.2}{10.1} = 1.5$$

As calculated above, the actual speed up for CAAS is 1.5x, which is lower than the actual speed up on a single computer which was 2.1x. This could be a result of high network latency, resulting in communication overhead or other reasonings mentioned earlier.

(1736 words)

References

Geekforgeeks. (2020). *Bernstein's Condition in Operating Systems*. Retrieved from https://www.geeksforgeeks.org/bernsteins-conditions-in-operating-system/

Geekforgeeks. (2022). *Bloom Filters Introduction*. Retrieved from https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/

Gutenberg. (n.d.). *Project Gutenberg*. Retrieved from https://www.gutenberg.org/

Wikipedia. (2023). *List of Hash Functions*. Retrieved from https://en.wikipedia.org/wiki/List_of_hash_functions

Wikipedia. (2023). *Jenkins Hash Function*. Retrieved from https://en.wikipedia.org/wiki/Jenkins_hash_function

Wikipedia. (2023). *Fowler–Noll–Vo Hash Function*. Retrieved from https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function

Name: Bryan Jun Kit Wong
Student ID: 32882424

Wikipedia. (2023). *Amdahl Law*. Retrieved from https://en.wikipedia.org/wiki/Amdahl%27s_law

Wikipedia. (2023). *Bloom Filter*. Retrieved from https://en.wikipedia.org/wiki/Bloom_filter