hainguyen200321@gmail.com

PARALLEL STRING SEARCHING ALGORITHM ON SHARED MEMORY

Contents

[Introduction 3](#_Toc145281416)

[Bottleneck Of Bloom Filter 3](#_Toc145281417)

[Pseudocode For Bloom Filter(serial version): 3](#_Toc145281418)

[Pseudocode For UniqueWordSeq 4](#_Toc145281419)

[Bloom Filter Diagram 6](#_Toc145281420)

[Word List & Query List Choose 8](#_Toc145281421)

[Word List 8](#_Toc145281422)

[Query List 8](#_Toc145281423)

[Measure & Analyze The Performance Of The Serial Algorithm/Code 8](#_Toc145281424)

[Measure 8](#_Toc145281425)

[Analyze 9](#_Toc145281426)

[Dependency Analysis For Bloom Filter 9](#_Toc145281427)

[Unique Words 10](#_Toc145281428)

[Hashing all the value 11](#_Toc145281429)

[Querying 12](#_Toc145281430)

[Theoretical Speedup 13](#_Toc145281431)

[Portion of the code that could be paralyzed: 13](#_Toc145281432)

[Percentage of paralyzed portion: 13](#_Toc145281433)

[Unique Word 14](#_Toc145281434)

[Pseudocode 14](#_Toc145281435)

[Diagram 14](#_Toc145281436)

[Hash 15](#_Toc145281437)

[Pseudocode 15](#_Toc145281438)

[Diagram 15](#_Toc145281439)

[Query 16](#_Toc145281440)

[Pseudocode 16](#_Toc145281441)

[Diagram 16](#_Toc145281442)

[Analyse & Evaluate The Performance Bloom Filter 17](#_Toc145281443)

[Actual speed up 17](#_Toc145281444)

[Analyse & Evaluate 17](#_Toc145281445)

# Introduction

In this report, I will discuss the process of parallelizing a string-matching algorithm, detailing everything from the rationale behind the chosen algorithm to the following areas of focus:

a) Investigating the computational bottleneck of a serial string matching algorithm like the Bloom Filter (or an equivalent) for a sufficiently large dataset or word counts and queries.

b) Conducting a dependency analysis to ascertain that the string-matching algorithm is parallelizable.

c) Calculating the theoretical speedup of a parallel implementation of the string-matching algorithm.

d) Designing and developing a parallel string-matching algorithm based on data parallelism in a shared memory parallel computing architecture.

e) Analyzing and evaluating the performance of the parallel algorithm.

For the objectives of this report, I opted for the Bloom Filter algorithm, which is complex yet sequential. This algorithm can be parallelized in many stages due to its independence from the data shared throughout the entire process, a key aspect in achieving true shared memory parallelization. The following components of the algorithm are parallelizable:

- Calculating the hash value for each inserted word (parallelizable).

- Cleaning the data by removing all duplicated words before processing (parallelizable).

- Looking up a list of words (parallelizable).

These factors significantly contribute to the algorithm's efficiency, accounting for up to 80% of its running time.

# Bottleneck Of Bloom Filter

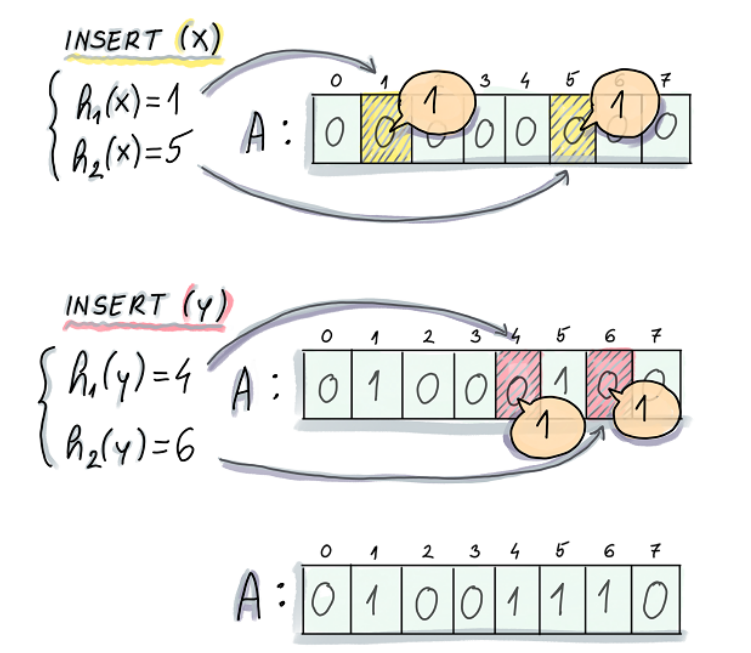
## Pseudocode For Bloom Filter(serial version):



## Pseudocode For UniqueWordSeq



## Bloom Filter Diagram



A whiteboard with math equations and a person standing

Description automatically generated

## Word List & Query List Choose

### Word List

The word list here I chose is designed to be as close to real-world examples as possible. Here, I chose to combine three different books together to increase the dataset but still keep it realistic. These books were chosen to enhance the data set even more but still needed to maintain a sense of realism. As the size increases, the false positive rate of the problem increases, which also tests the efficiency of the algorithm against a sufficiently large data set (≈1.4 million word), assessing how well the algorithm can handle the volume effectively.

These three books are "Moby Dick", "Little Women", and a collection of works by Shakespeare. These books were input as individual words, and the format of the text file was designed so that each word would be on a new line.

### Query List

The query list is a set provided by a third party, the Monash FIT 3143 Team, which includes a list of words that is long enough(91636 words) and contains words both present and absent in the word list. Each of the words present in the query list is accompanied by a number indicating its presence in the word list.

## Measure & Analyze The Performance Of The Serial Algorithm/Code

### Measure

For this Serial algorithm, I run it in a system consisting of:

* The CPU specifications 1 CPU per task (AMD Epyc)
* GPU specifications: NVIDIA RTX
* The memory allocated for the job, which is 16GB here.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Word Combination | Word Size (Words) | Query Size (Words) | Unique Word Time (s) | Word Filtered(words) | Hash(s) | Query(s) | Total(s) |
| LITTLE\_WOMEN | 200000 | 90 000 | 170.238 | 13144 | 0.841051 | 11.0441 | 183.1231 |
| MOBY\_DICK | 215000 | 90 000 | 284.404 | 22321 | 1.4002 | 12.8034 | 291.607 |
| MOBY\_DICK + LITTLE\_WOMEN | 415000 | 90 000 | 292.487 | 35462 | 2.22947 | 14.8081 | 310.524 |
| SHAKESPEARE | 950 000 | 45000 | 48.2463 | 49528 | 2.31395 | 9.1573 | 60.7175 |
| SHAKESPEARE | 950 000 | 90000 | 50.1653 | 49528 | 2.30047 | 10.9886 | 64.4543 |
| SHAKESPEARE + MOBY\_DICK | 1150000 | 90 000 | 58.7165 | 58705 | 3.12692 | 15.0042 | 77.8476 |
| SHAKESPEARE + MOBY\_DICK + LITTLE\_WOMEN | 1400000 | 90000 | 64.3944 | 71846 | 4.52887 | 18.058 | 88.48127 |

Table 1 : The Performance Of The Serial Algorithm/Code

### Analyze

In this algorithm, I divide the word insertion process into two primary steps. Initially, the data undergoes a cleaning process to minimize the number of words inserted into the database, thereby aiming to decrease the overall insertion time. Unfortunately, this UniqueWords Algorithm often increase the insertion time. While it nails it goal in some scenarios (especially when analyzing Shakespearean texts), it generally increased the total runtime in most cases, almost doubling the average time needed compared to merely cleansing the data. It was evident that there was a linear correlation between the number of words inserted and the time taken to complete the insertion. This growth trend seemed linear relative to the volume of words being processed.

However, the algorithm exhibited inconsistent performance during tests involving "Little Women" and "Moby Dick". In these instances, the algorithm seemed to loop to the list's end upon each check, representing the worst-case scenario, in contrast to the Shakespearean dataset where the loop did not always reach the list's end, representing an average case scenario. This resulted in a somewhat unstable performance trend.

Additionally, while the theoretical false positive rate for the Bloom Filter was set using the default formula at approximately 0.01, practical application demonstrated a considerably lower rate of about 0.001, as calculated by the ratio of false positive cases to the total number of cases. This discrepancy between the theoretical and actual false positive rates could potentially highlight the efficiency of the Bloom Filter algorithm in filtering out false positives, even when tested under varying conditions.

# Dependency Analysis For Bloom Filter

## Unique Words



Iteration for i = 1:

* Input set (I1): {FileWordList[1], UniqueWordList}
* Output set (O1): {FileWordList[1], UniqueWordList, UniqueWordListLength}

Iteration for i = 2:

* Input set (2): {FileWordList[2], UniqueWordList}
* Output set (O2): {FileWordList[2], UniqueWordList, UniqueWordListLength}

Applying Bernstein's conditions:

1. I1∩O2≠∅ (Anti-dependency, because I1 reads UniqueWordList and O2 writes to UniqueWordList)
2. I2∩O1≠∅ (Flow dependency, because I2 reads UniqueWordList and O1 writes to UniqueWordList)
3. O1∩O2≠∅ (Output dependency, both iterations write to UniqueWordList and UniqueWordListLength)

**Conclusion:** As per the above analysis, none of Bernstein's conditions are satisfied, meaning these iterations are dependent on each other and cannot be parallelized as they are. But we can sacrifice the accuracy of the algorithm and paralyze although the answer will not be fully correct, but it still correct as there some redundant to it.

Hashing all the value

  
Iteration for i = 0, j = 0, l = 0:

* Input set (I1): {ppWordListArray[0][0], k}
* Output set (O1): {allHashValues[0][0][0]}

Iteration for i = 0, j = 0, l = 1:

* Input set (I2): {ppWordListArray[0][0], k}
* Output set (O2): {allHashValues[0][0][1]}

Applying Bernstein's conditions (for the above iterations):

1. *I1∩*O2*=∅* (Anti-dependency)
2. I2*∩*O1*=∅* (Flow dependency)
3. O1*∩*O2*=∅* (Output dependency)

**Conclusion:** As all the sets are empty sets, it satisfies Bernstein's conditions which means the iterations are independent and the loop is parallelizable. The memory locations being read and written in each iteration are distinct.

Querying

****Iteration for *i*=1:

* Input set *I*1​: {queries[1],ppWordListArray,bloomFilter}{queries[1],ppWordListArray,bloomFilter}
* Output set *O*1​: {result,actualResult}{result,actualResult}

Iteration for *i*=2:

* Input set *I*2​: {queries[2],ppWordListArray,bloomFilter}{queries[2],ppWordListArray,bloomFilter}
* Output set *O*2​: {result,actualResult}{result,actualResult}

Applying Bernstein's conditions:

1. *I*1​∩*O*2​=∅ (Anti-dependency)
2. *I*2​∩*O*1​=∅ (Flow dependency)
3. *O*1​∩*O*2​=∅ (Output dependency)

**Conclusion:** The conditions indicate that these iterations are independent, and the loop is parallelizable. The variables result and actualResult are local to each iteration and do not cause a data dependency between iterations. The only shared data read is from the queries and ppWordListArray vectors, and the bloomFilter object, but since they are not being modified, there is no data dependency here either.

# Theoretical Speedup

## Portion of the code that could be paralyzed:

* Unique Word
* Hash
* Query

## Percentage of paralyzed portion:

Here, I will calculate it based on the result of SHAKESPEARE + MOBY\_DICK + LITTLE\_WOMEN (table:1.)

f = Hash+ Query+ Unique Word /Total = 86.98127 / 88.48127 \* 100 ≈ 98.30%

S(P)= 1/((1−f)+ f/p) - > S(16)= 1 - ((1−0.983)+ 16/0.983) ≈ 15.29

​

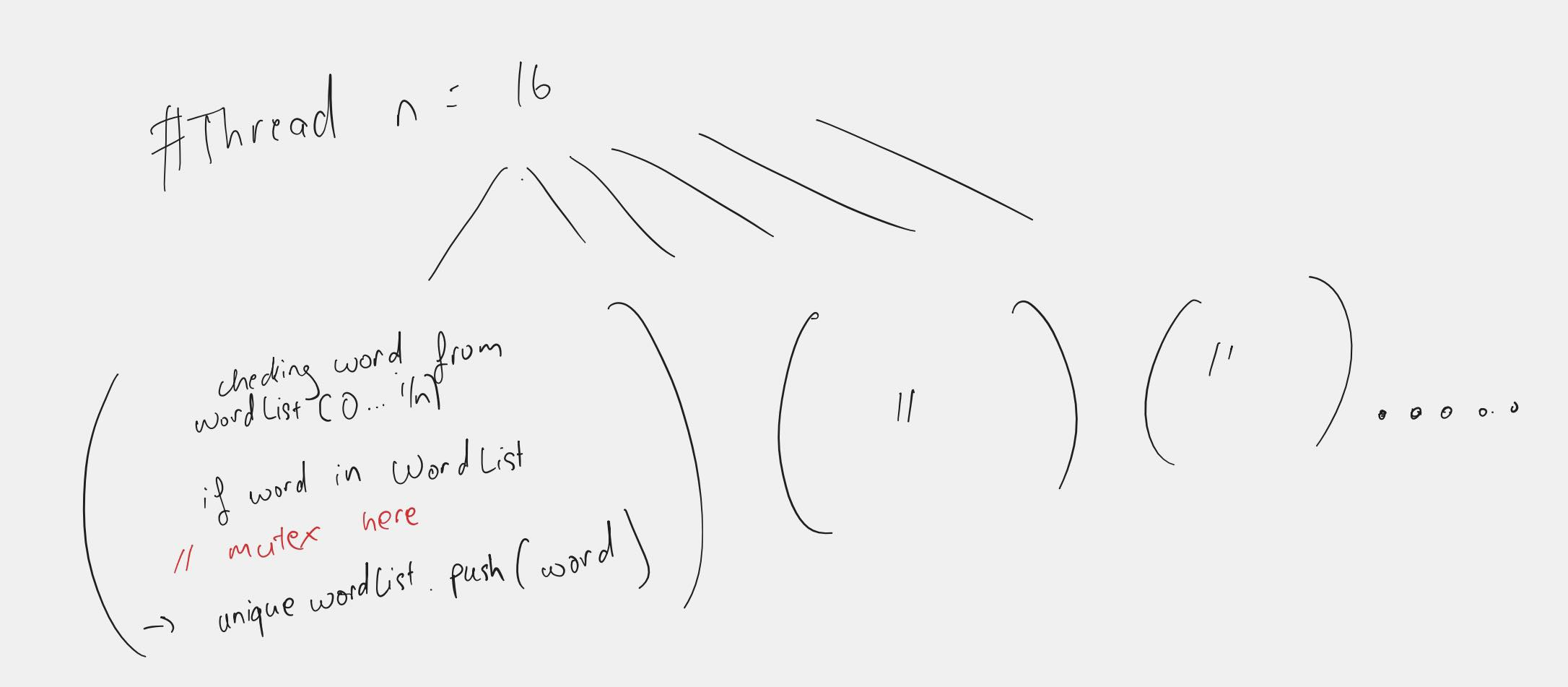
​Design Parallel Bloom Filter

## Unique Word

In this function, we can parallelize the entire for loop, spawning n number of threads. Each thread will be in charge of a set of words (i/n). The result of this function might be partially correct due to the UniqueWordList being read and written at the same time, but the main objective of this loop still remains; it tries to parallelize the process of checking if the word already exists in the array, ultimately reducing the number of words that need to be checked in later stages.

Pseudocode  


### Diagram



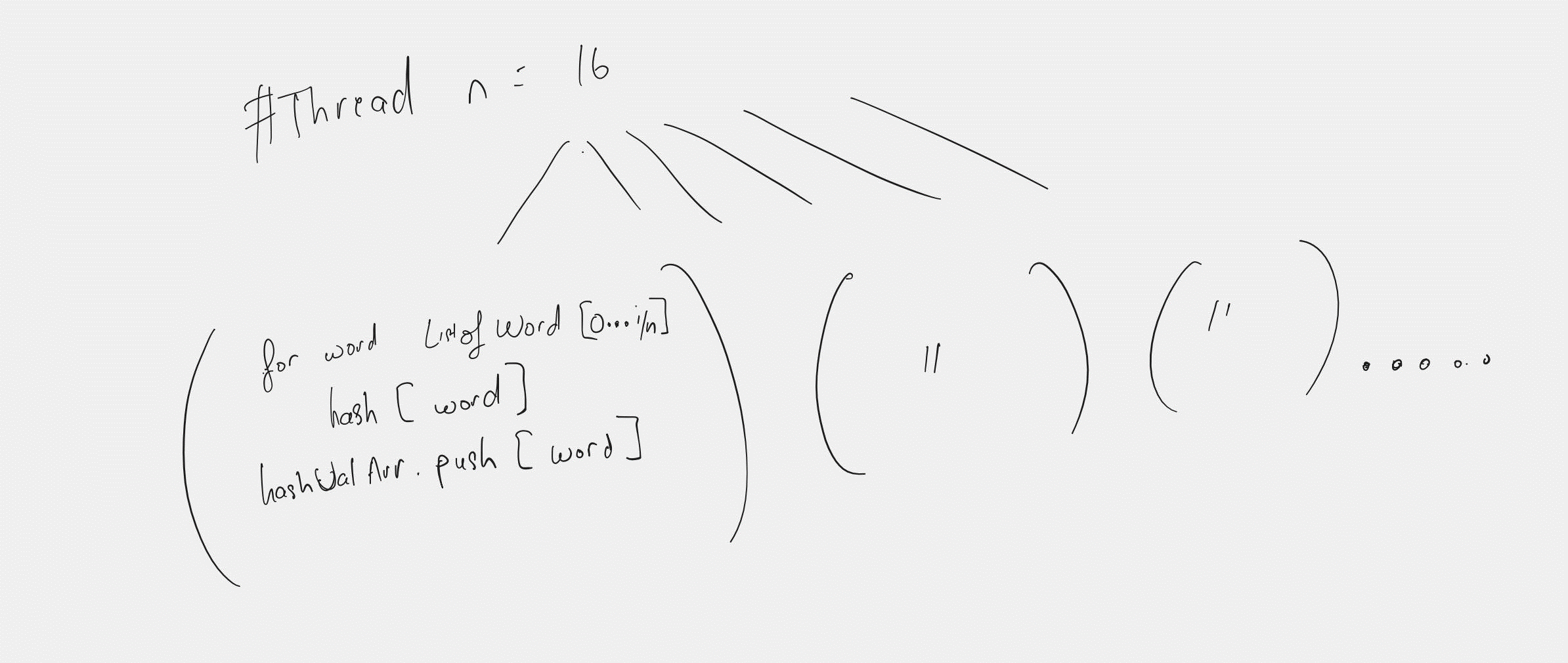
## Hash

We can parallelize the entire process of hashing all the values that need to be inputted into the bloom filter. Here, we assign each thread with a set of variables to hash, and then hash all of them. We can avoid a race condition by redefining the position of the hash value in it. We construct a 3D array where the first dimension will contain words from different books, the second dimension indicating the word, and the third dimension will house all the hash values.

Pseudocode



### Diagram



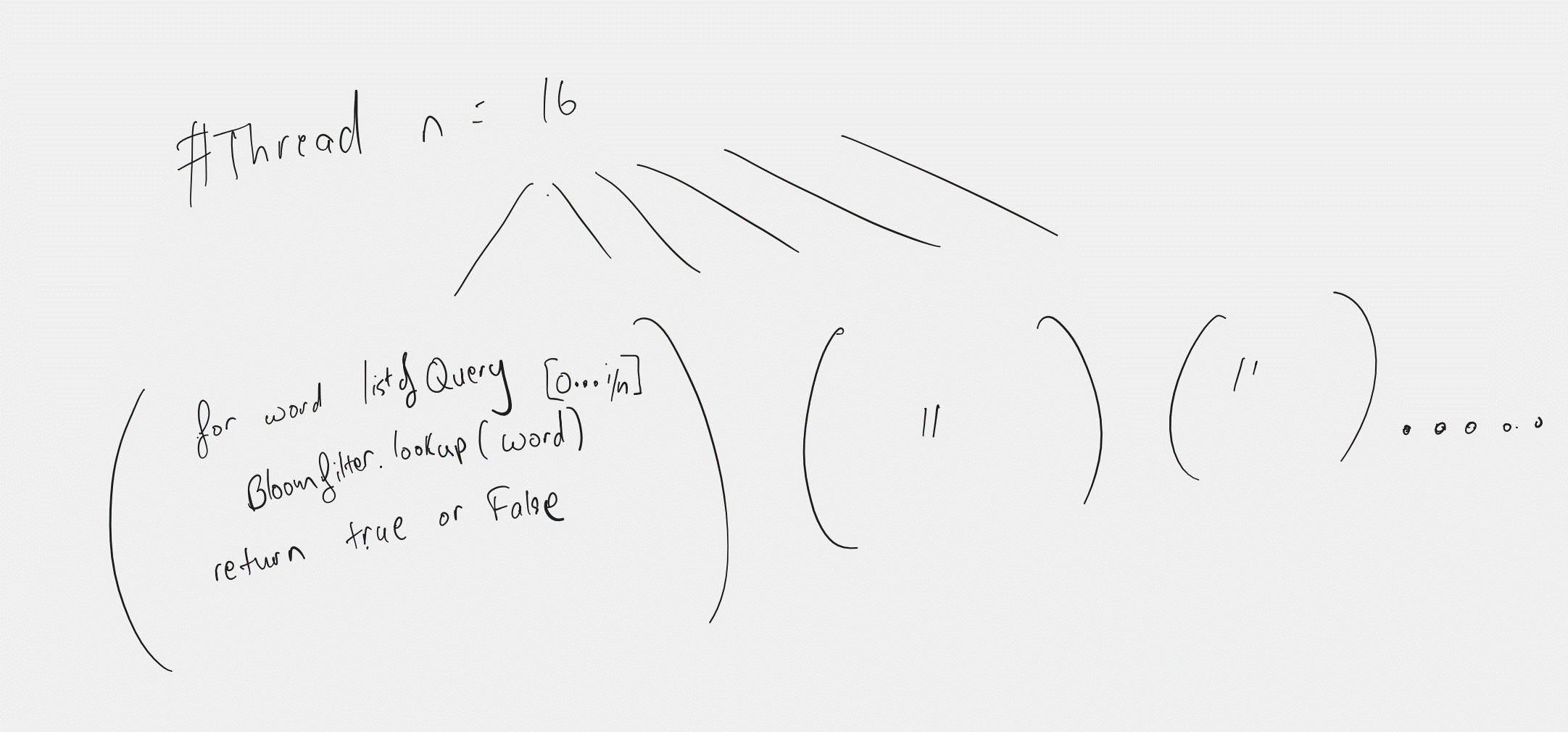
## Query

In the query, we can take all the inputs and put them into one array. From that, we create another array to iterate through the entire array. By doing this, we can parallelize the process of reading the words and querying it, where each thread will be in charge of a set of words to query, and all of these are independent of each other.

Pseudocode



### Diagram



# Analyse & Evaluate The Performance Bloom Filter

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Word Combination | Word Size (Words) | Query Size (Words) | Unique Word Time (s) | Hash(s) | Query(s) | Total(s) | Core |
| LITTLE\_WOMEN | 200000 | 90 000 | 0.405133 | 0.856622 | 0.67686 | 1.938615 | 16 |
| MOBY\_DICK | 215000 | 90 000 | 0.753152 | 1.46439 | 0.823497 | 3.041039 | 16 |
| MOBY\_DICK + LITTLE\_WOMEN | 415000 | 90 000 | 1.04506 | 1.48351 | 1.04862 | 3.57719 | 16 |
| SHAKESPEARE | 950 000 | 45000 | 3.8135 | 2.42594 | 0.503887 | 6.743327 | 16 |
| SHAKESPEARE | 950 000 | 90000 | 3.7999 | 2.52324 | 0.866186 | 7.189326 | 16 |
| SHAKESPEARE + MOBY\_DICK | 1150000 | 90 000 | 4.05771 | 2.41203 | 1.11046 | 7.5802 | 16 |
| SHAKESPEARE + MOBY\_DICK + LITTLE\_WOMEN | 1400000 | 90000 | 4.82748 | 2.55405 | 1.39123 | 8.77276 | 16 |
| SHAKESPEARE + MOBY\_DICK + LITTLE\_WOMEN | 1400000 | 90000 | 8.40882 | 2.38706 | 2.41888 | 13.21476 | 8 |

## Actual speed up

Speed up (88.48127 - 8.77276) / 88.48127 \* 100 90.0851785 %

## Analyse & Evaluate

From the result above, the unique word step with parallel implementation is actually speeding up the process of reducing the number of words, which is unexpectedly increasing the required time significantly instead of reducing the time. There is a close correlation between the theoretical speed up and the actual speed up, with the theoretical being 98.30% and the actual around 90%. This can be understandable since there might be bottlenecks in data transferring.

The results fluctuated considerably but stayed under the 0.01 false positive rate, maintaining high accuracy. Despite the fluctuations, it still managed to cut down the time for reducing the number of words, querying, and hashing, which is unexpectedly equivalent to the serial implementation.