FIT3143: Parallel Computing

Assignment 1

Parallel String Searching Algorithm on Shared Memory

Lim Zheng Haur

32023952

6th September 2023

Table of Content

Introduction 3

Dependency Analysis and Theoretical Speed-up Calculations 4

Parallel Algorithm Design 7

Results Tabulation 10

Analysis and Discussion 11

Appendix 12

References 14

Introduction

The selected algorithm for this assignment is the Bloom Filter Algorithm. Bloom Filter Algorithm is a space-efficient probabilistic data structure for testing the membership of an element in a set. In this case, we will be using it to conduct string searching to check if a query string is in the dataset. Bloom filters are extremely space efficient because they do not store the string itself in memory, but rather a fixed-size array of bits.

During insertion, Bloom Filter Algorithm hashes the input string through multiple hash functions and changes the corresponding bit positions to 1 in the bit array. In this assignment, I have chosen to use AP Hash, and Jenkins Hash as the hash function for my Bloom Filter Algorithm. The same process is executed for querying a word but instead of changing the bit in the array to 1, the algorithm checks if the bit of the index is set to 1. Due to this hashing method of insertion and query, Bloom Filter Algorithm is a probabilistic data structure as there might be collusions between different inputs where they share the same bit positions. Hence, there might occurs false positives when such collusions occur, but false negatives will never occur in a Bloom Filter.

The Bloom Filter algorithm is chosen for this assignment due to the wide use of it in the industry. The main benefit of this algorithm is the space efficiency and quick lookup time complexity. However, choosing a correct bit array size is important as size too small would lead to more collusions which creates a higher false positive rate, but a size that is too big would lead to wasted space which defeats the purpose to this algorithm. There are two ways to determine the size of the bit array which is static and dynamic. Static determines the suitable size of the algorithm in the creation while dynamic increases the size of the bit array as required during insertion. The method chosen for this assignment is the static method as the number of unique words are counted before the creation of the bit array.

The choice for the hash functions of this assignment is AP Hash and Jenkins Hash. AP Hash is a non-cryptographic hash function typically used for strings. It works by taking an initial hash value, multiplying it with a constant prime number and XOR the hash value with the character, repeating it for every character of the string. Jenkins Hash also starts with an initial hash value, XOR the hash value with the character, multiplying it with a prime constant, and adding bitwise operations such as shifts and XORs, repeating for every character of the string. Both these functions are similar that they are relatively quick to compute and are often used in hash-based data structures.

Dependency analysis and theoretical speed up calculations

Dependency analysis:

Bernstein's conditions are used to verify that each loop iteration is independent and could be parallelized. The Bernstein's conditions are:

1. $I_1 \cap O_2 = \emptyset$ (Anti dependency)
2. $I_2 \cap O_1 = \emptyset$ (Flow dependency)
3. $O_1 \cap O_2 = \emptyset$ (Output dependency)

There are 3 portions of the code that will be verified by Bernstein's conditions. These are the checking of unique words, insertion, and query.

1. Checking of unique words

   $I_1 = word_1$
   $I_2 = word_2$
   $O_1 = unique\ word\ list$
   $O_2 = unique\ word\ list$

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

   O_2 is dependent on I_1 because the O_2 requires O_1, the process of checking if each word is unique is actually not independent and could not be parallelized.

2. Insertion

   $I_1 = word_1$
   $I_2 = word_2$
   $O_1 = bit\ at\ hash\ position_1$
   $O_2 = bit\ at\ hash\ position_2$

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

   The process of insertion is to hash the word and to change the bit of the corresponding hash position to 1. Even in the event where two words have the same hash, the bit in the hash position will still be 1. Thus, the process of insertion is independent according to the Bernstein's conditions.

3. Query

$I_1 = word_1$
$I_2 = word_2$
$O_1 = results_1$
$O_2 = results_2$

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

The process of query is to verify the hash values of the input is 0/1 in the bit array and return true or false. Each input and their corresponding output are independent of other inputs. Therefore, the querying portion of the code is possible to be parallelized according to Bernstein's conditions.

Theoretical speed up calculations:

Amdahl's law: $\psi \leq \frac{1}{f+(1-f)/p}$ where,

- f = sequential fraction of the code.
- $1-f$ = parallelizable fraction of the code.
- p = number of processors which is 8 in the system used.

1. Reading unique words

$$f = \frac{17.370476 - 16.985213}{17.370476} = 0.022179$$

$$1-f = \frac{16.985213}{17.370476} = 0.977821$$

$$\psi \leq \frac{1}{f+\frac{1-f}{p}} = \frac{1}{0.022179+\frac{0.977821}{8}} = 6.924872$$
$$\psi \leq \frac{1}{f+\frac{1-f}{p}} = \frac{1}{0.022179+\frac{0.977821}{\infty}} = 45.087696$$

Using Amdahl's law, we can predict that the potential improvement of this portion of the sequential code will have a speed-up of almost 7 times with 8 processors. When the number of processors approaches infinite, the maximum speed-up will not be more than 45.

2. Insertion

$$f = \frac{17.370476 - 0.006127}{17.370476} = 0.999647$$

$$1 - f = \frac{0.006127}{17.370476} = 0.000353$$

$$\psi \leq \frac{1}{f + \frac{1-f}{p}} = \frac{1}{0.999647 + \frac{0.000353}{8}} = 1.000310$$

$$\psi \leq \frac{1}{f + \frac{1-f}{p}} = \frac{1}{0.999647 + \frac{0.000353}{\infty}} = 1.000353$$

According to Amdahl's law, this portion of the code will have almost no speed up difference even if we have infinite number of processors. However, I assume that despite this portion of the code being a small fraction of the entire code, there will still be a speed-up by running parallel as compared to sequential.

3. Query

$$f = \frac{17.370476 - 0.004807}{17.370476} = 0.999723$$

$$1 - f = \frac{0.004807}{17.370476} = 0.000277$$

$$\psi \leq \frac{1}{f + \frac{1-f}{p}} = \frac{1}{0.999723 + \frac{0.000277}{8}} = 1.000243$$

$$\psi \leq \frac{1}{f + \frac{1-f}{p}} = \frac{1}{0.999723 + \frac{0.000277}{\infty}} = 1.000277$$

By using Amdahl's law, the query portion of the code is similar to the insertion that there is almost no difference in the speed between serial and parallel. However, I am still assuming that there is a small speed-up when converting this serial code into parallel despite being a small portion from the entire code. This is because Amdahl's law calculates the speed-up of the entire code rather than the section itself.

Parallel algorithm design

Algorithm Pseudocode:

```
i.   Read all unique words from the word files into a word list and count number of unique
     words. (Use Open MP to parallelize for loop comparing words where each thread
     compares to check if word exists in word list)
       1. Read word.
       2. If word already exists in word list, read next word, and repeat from 1.
       3. If word does not exist in word list, store word into word list.
       4. Increment unique word count by 1. (Use Open MP critical option to prevent
          race condition)
       5. Repeat from 1 if there is still exists words to be read.
       6. Return unique word count.
ii.  Create a bit array of suitable size for the number of words in word list.
       1. Calculate bit array size by setting a maximum false positive value.
       2. Create bit array of size calculated.
       3. Set all values in bit array into 0.
iii. Insert all words into bit array. (Use Open MP to parallelize the insertion where each
     thread inserts a certain number of words into bit array)
       1. Hash the word.
       2. Change the bit at the hash value index to 1.
       3. Repeat for all words in word list.
iv.  Read all words from query file into a query list. (Use Open
       1. Read word and 0/1 representing the existence of word in word list.
       2. Store word into query list and 0/1 into a truth list
v.   Query all words from query list. (Use Open MP to parallelize the query where each
     thread queries a certain number of words and store results in the results list)
       1. Hash the word.
       2. Check if the bit at the hash value is 1.
       3. Store 0 if false, and 1 if true into a results list
vi.  Compare results list with truth list.
       1. If both value in results list and truth list is 1, add 1 to true positive.
       2. If both value in results list and truth list is 0, add 1 to true negative.
       3. If value in results list is 1 and truth list is 0, add 1 to false positive.
       4. If value in results list is 0 and truth list is 1, add 1 to false negative.
```

Parallelism flowchart:

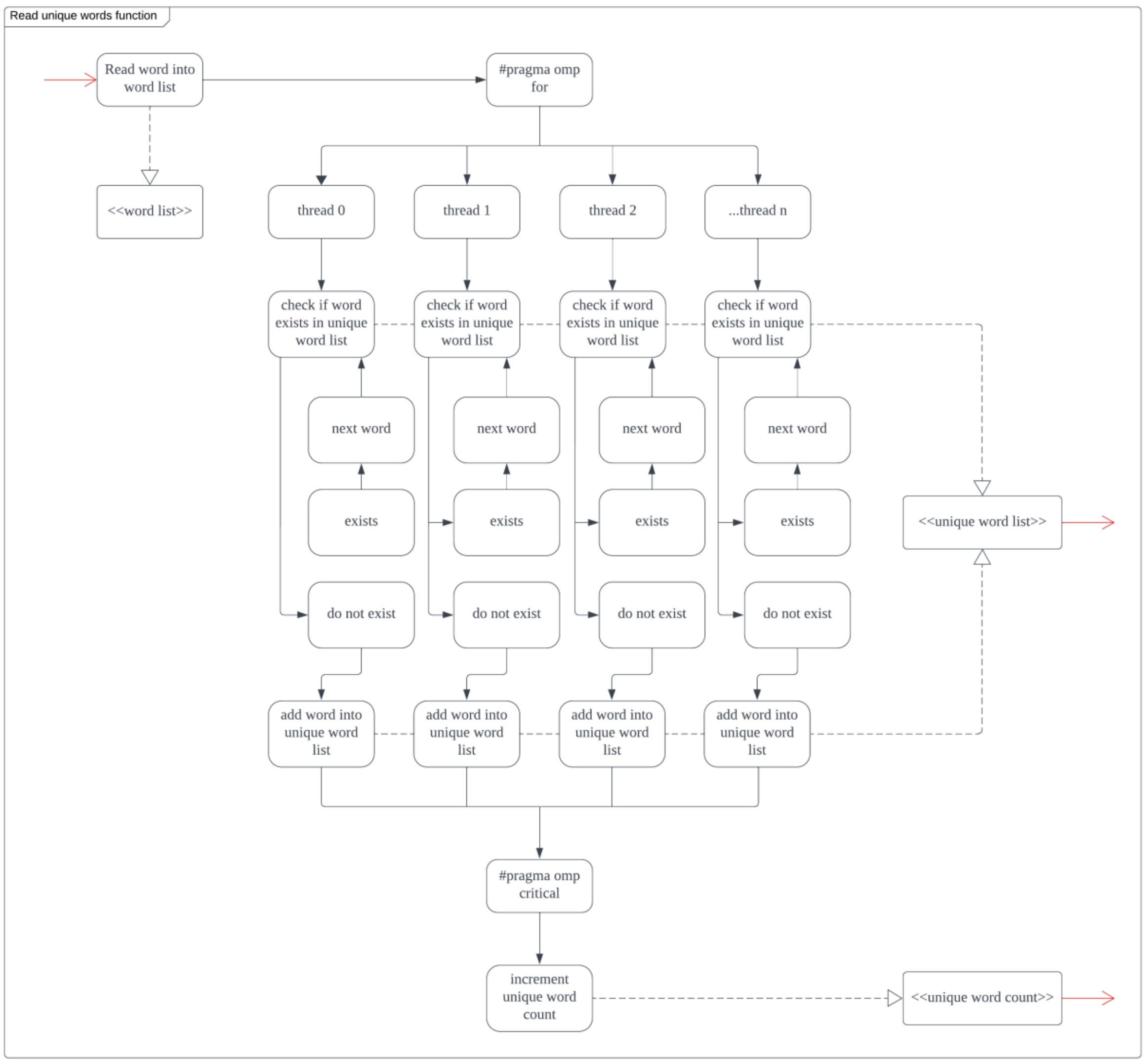


Diagram 1

A flowchart showing parallel OpenMP for loop in the function for reading unique words.

Design justification:

To parallelize the serial code, I have used OpenMP functions to parallelize the “for loops” in certain functions. Diagram 1 shows a flowchart that displays how I have used OpenMP functions to parallelize the comparison of each word in the word list to check if they exist in the unique word list. Diagram 1 also shows the use of the critical function to prevent a race condition among threads for the increment of the unique word count. Each thread will be responsible of the checking for duplicates for a certain section of the word list. This parallelism method is applied to each portion of the codes that could be parallelized.

The choice of the scheduler for each OpenMP for loop is dynamic. This is due to the fact that most process such as comparison, insertion and query are dependent on the length of the word and dynamic scheduling would provide the best results as compared to other options such as static.

The portion of the codes that has been parallelized are the count unique words portion, the insertion portion, and the query portion. The reason that the reading file portion is not parallelized is because of the use of “fscanf” function. Parallelizing I/O bound tasks might not be optimal because multiple threads trying to access the file simultaneously might lead to contention issues which can be relatively slow. Counting unique words require each word to be compared against the entire existing unique word list which is a slow process and parallelizing each comparison will significantly increase the efficiency of the code. On the other hand, insertion is a process of hashing the word with the hash functions and changing the bits of the bit array which could be parallelized for each thread to insert a chunk of words to increase efficiency. Lastly, the query portion is similar to the insertion where words are hashed with the hash function and checked with the bit array if the bits are 0 or 1 which could be parallelized so that each thread queries a chunk of words from the query list.

Results tabulation

Results are tabulated based on ***Appendix 1*** *and* ***Appendix 2****.*

Time Taken (s)	Read file	Insertion	Read query	Query	Total
Serial	16.985213	0.006127	0.371090	0.004807	17.370476
Parallel	8.516365	0.011177	0.342273	0.003216	8.878898

Table 1

Table showing the time taken in seconds for execution.

Count	Unique words	Bit array size	Query words
Serial	71846	447975	91640
Parallel	73367	457459	91640

Table 2

Table showing the number of each items calculated.

Performance	True Positive	True Negative	False Positive	False Negative
Serial	87483	3622	535	0
Parallel	87483	3557	600	0

Table 3

Table showing the performance metrics.

Analysis and discussion

Observing Table 1, we could see that by parallelizing the serial code, we are able to reduce the total run time of the process from 17.37 seconds to 8.88 seconds. The sections of the code that I have parallelized is the checking of unique words, the insertion, and the query sections. Observing the time taken for each of this section, we can see that the unique word comparison which is within the reading file section of the code reduced from 16.98 seconds to 8.51 seconds. This is an approximately 2 times speed-up which does not align with Amdahl's Law which suggests a 7 times speed-up. This is because Amdahl's law generally overestimates the parallelization as it ignores the communication time between threads. Looking at the insertion section shows that by parallelizing the section, the process time has increased instead of decreasing. This is due to the process being a simple calculation process and by parallelizing the section, it instead creates more overhead in communication among threads. The query section of the code instead decreased from 0.0048 seconds to 0.0032 seconds. However, the is still little to no significance of parallelizing this code. This observation aligns with the theoretical speed-up calculations that was conducted prior to this.

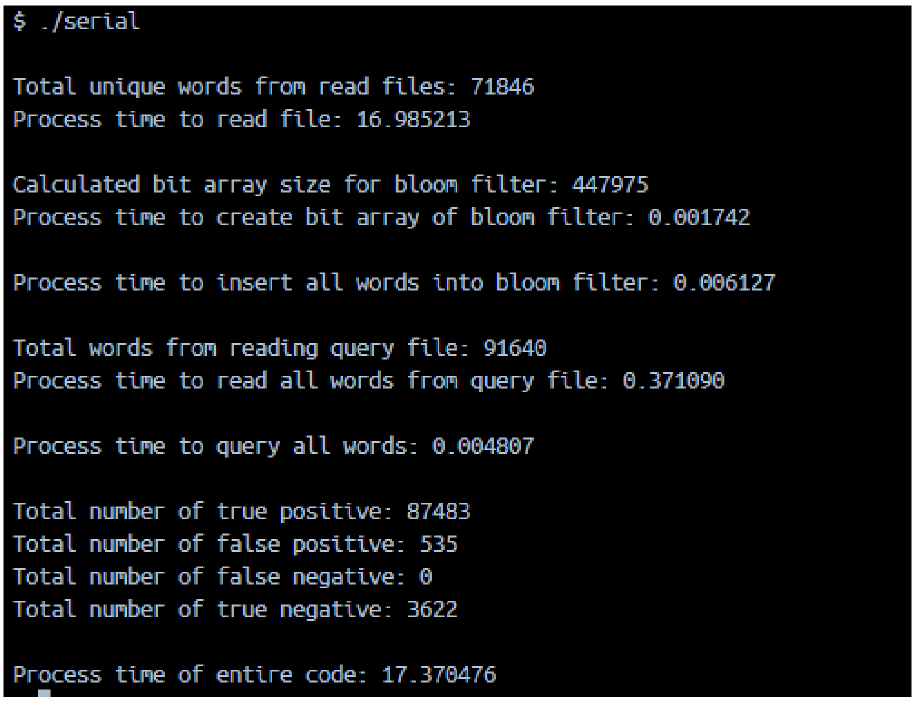
Moving on to Table 2, we could see that the number of unique words has increased from 71846 words to 73367 words when we parallelized our codes. This observation aligns with the verification of Berstein's conditions as well. This code will produce an inaccurate value when being parallelized as there exists dependencies in the loop iterations. However, since these words are used for insertion where a small number of duplicates does not significantly affect the insertion process. This is as observed where the insertion process time is a small fraction of the entire code. Therefore, I believe that a small sacrifice in accuracy in exchange for an almost 2 times speed-up of the code is worth it for the Bloom Filter Algorithm.

As shown in Table 3, the performance of the serial and parallel code is similar where both has a false positive rate of lesser than 5%. This shows the potential of the sequential code to be parallelized and maintain its functionality as intended.

To better demonstrate the ability of parallel computing, I believe that a more complicated hash functions which require more computational asset would be suitable. The larger amount of computing required would better demonstrate the abilities of parallel computing and allow for the insertion section and the query section to emerge as a larger fraction of the code and decrease the communication overheads of parallel computing as the comparison of it is relative.

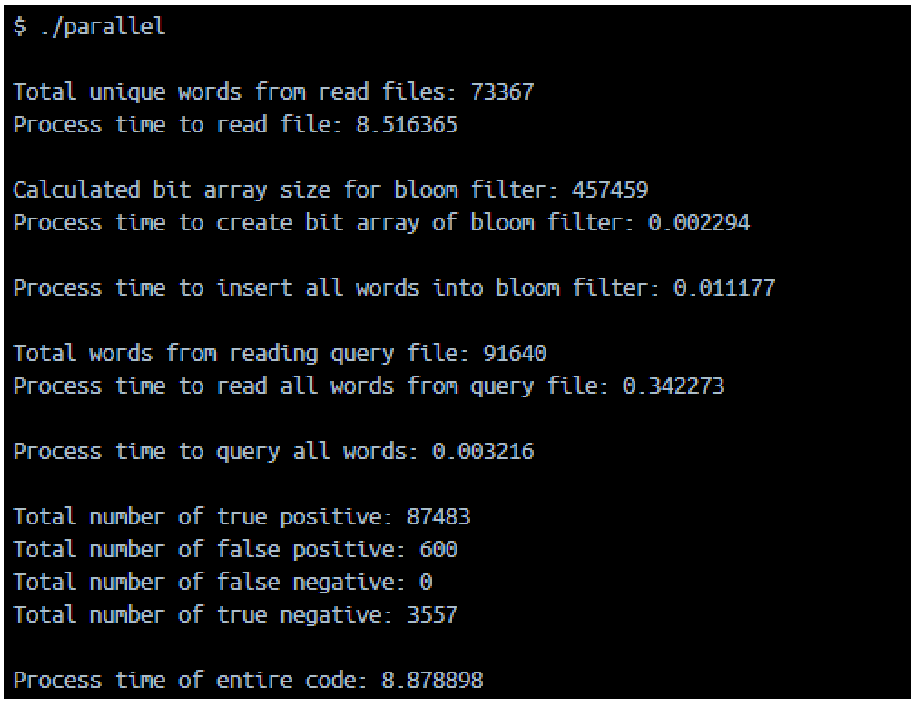
Appendix

Appendix 1: *Output of running serial code.*



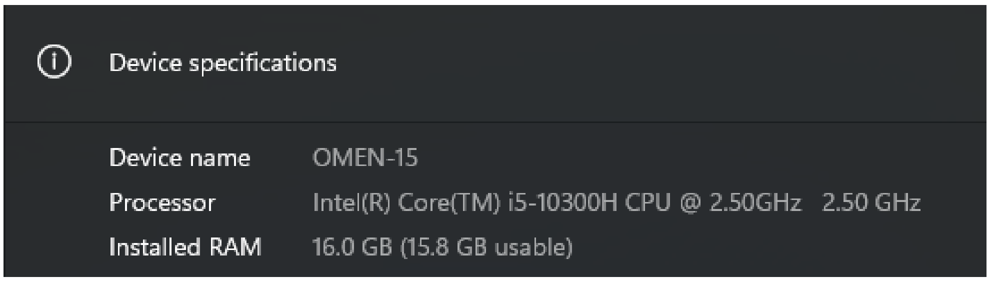
```
$ ./serial

Total unique words from read files: 71846
Process time to read file: 16.985213

Calculated bit array size for bloom filter: 447975
Process time to create bit array of bloom filter: 0.001742

Process time to insert all words into bloom filter: 0.006127

Total words from reading query file: 91640
Process time to read all words from query file: 0.371090

Process time to query all words: 0.004807

Total number of true positive: 87483
Total number of false positive: 535
Total number of false negative: 0
Total number of true negative: 3622

Process time of entire code: 17.370476
```

Appendix 2: Output of running parallel code.



```
$ ./parallel

Total unique words from read files: 73367
Process time to read file: 8.516365

Calculated bit array size for bloom filter: 457459
Process time to create bit array of bloom filter: 0.002294

Process time to insert all words into bloom filter: 0.011177

Total words from reading query file: 91640
Process time to read all words from query file: 0.342273

Process time to query all words: 0.003216

Total number of true positive: 87483
Total number of false positive: 600
Total number of false negative: 0
Total number of true negative: 3557

Process time of entire code: 8.878898
```

Appendix 3: System configuration



Device specifications

Device name	OMEN-15
Processor	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz
Installed RAM	16.0 GB (15.8 GB usable)

References

AP Hash. Programming Algorithms. (n.d.). Retrieved from https://www.programmingalgorithms.com/algorithm/ap-hash/c/

Jenkins, B. (2013, November 3). *A Survey of Hash Functions*. The Hash. Retrieved from http://www.burtleburtle.net/bob/hash/doobs.html