A) Investigate the computational bottleneck of the serial string-matching algorithm (Z-Algorithm)

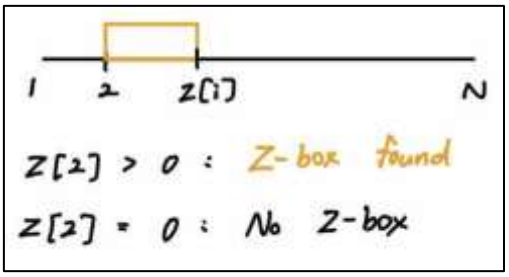
a) Description of Complex Serial Based String Matching Algorithm

The string-matching algorithm of choice is the Z-algorithm, referenced from GeeksforGeeks (Trivedi, 2022). In this algorithm, a Z-array is constructed for the input string *str* of length N, in which each cell (excluding the first element, because any string is also a substring of itself, therefore Z[1] carries no meaning) stores the length of the longest substring starting from *str*[i…N-1] which is also a prefix of *str*.

To compute the Z-array values (a.k.a. Z-values), we continuously reuse previously computed z-values as a reference to minimize character comparisons. We also compute the Z-box at each iteration i to keep track of the largest matched substring ending at *str*[i].

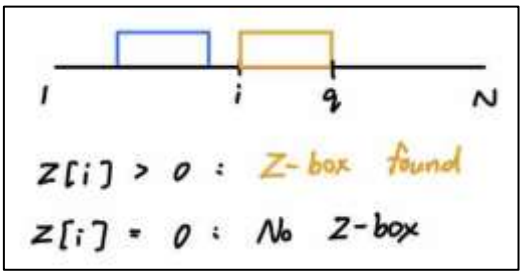
The computation can be broken down into a few cases:

Notation:
L[i] stands for the index of the start of the Z-box before index i;
R[i] stands for the index of the start of the Z-box before index i;
Z[i] stands for the Z-value of the substring str[i…n].



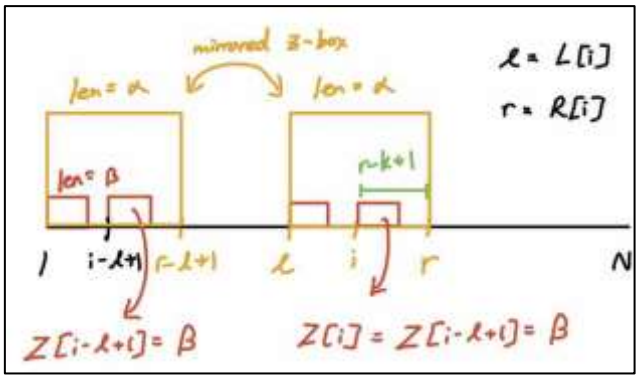
Base case (i = 2): This is the start of the algorithm, therefore no reference Z-value or Z-box is present. The second Z-value Z[2] is explicitly computed by comparing each letter.

For the remaining Z-values and Z-boxes, they are computed using the following rules:

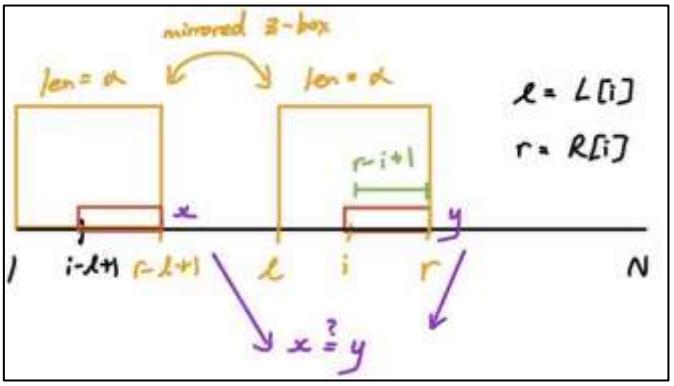


Case 1 - i > R[i]: This means that there is no known matching substring that we can reference, an explicit comparison is needed for *str*[i…q-1] with str[1…q-k] until a mismatch is found at a certain index q ≤ k.

Case 2 - i ≤ R[i]: This means that there is an existing Z-box (i.e. matching substring with the prefix) within the area of interest, therefore we need to consider a few sub-cases, by comparing the substring with its mirrored equivalent, which had its Z-value and Z-box computed earlier in the algorithm.



Case 2a: Z[i – L[i] + 1] < R[i] – i + 1: From the Z-value of the mirror of the current substring and end index of its corresponding Z-box, we can interpret the matching part of the current substring with the prefix of the string falls within the Z-box.



Case 2b: Z[i – L[i] + 1] >= R[i] – i + 1: Since our reference ends at R[i], we need to obtain the remaining information. Therefore, we need to explicitly compare *str*[R[i] + 1…] with *str*[R[i]-k+2] until a mismatch occurs.

This algorithm is a suitable choice because it is able to identify whether strings are matched in linear time. To compare two strings *s1* and *s2*, we can concatenate *s1* and *s2* with a character that is not present in both strings (in this case the '$' character is used), and we supply the resulting string as an input to the Z-algorithm. Upon iterating through the values of the computed Z-array, we can determine that two strings are matching if the length of one is equal to the z-value of the position right after the '$' character.

b) Implementation of the Serial Version of the String-matching Algorithm in C

The implementation of the algorithm is designed to check for all unique words that are present in the given text file.

The pseudocodes of the main function, along with the granular functions are as follows:

```
uniqueWordList[];
wordlist[];
uniqueWordCount = 0;
ReadFromFile(inputfilename, wordList);
for each word in wordlist do
        if isUnique(word, uniqueWordList) do
                uniqueWordList[uniqueWordCount] = word;
                uniqueWordCount++;
        endif
endfor
for each word in uniqueWordList do
        WriteToFile(outputfilename, uniqueWordList);
endfor
```

```
Func stringPatternMatch(word 1, word2)
        string = word1 ++ '$' ++ word2;
        z_arr = [];
        z_algo(string, z_arr);
        if z_arr[strlen(pat) + 1] ≠ strlen(pat) do
                return True;
        else
                return false;
        endif
endFunc
```

```
Func z_algo(string, z_arr)
        for each cell in z_arr do
                Cell = 0;
        endfor
        z_arr[0] = strlen(string);
        L = 0;
        R = 0;
        for each cell in z_arr do
                if case 1:
                        while match do
                                R++;
                        endwhile
                        cell = R – L;
                        R--;
                else if case 2a:
                        cell = z_arr[pos – L];
                else
                        while match do
                                R++;
                        endwhile
                        cell = R – L;
                        R--;
                endif
        endfor
endFunc
```

```
Func isUnique(pat, uniqueWordList)
        for each word in uniqueWordList do
                if stringPatternMatch(pat, word) do
                        return False;
                endif
        endfor
        return True;
endFunc
```

```
Func ReadfromFile(filename, wordList)
        f = fopen(filename)
        for i in range sizeof(f) do
                word = readline(file);
                wordlist[i] = word;
        endfor
endFunc
```

```
Func WriteToFile(filename, wordList)
        f = fopen(filename)
        for word in wordList do
                writeline(file, word);
        endfor
endFunc
```

c) Description of the Word List Used to Test the Serial Code

The word lists I used are named *MOBY_DICK.txt, LITTLE_WOMEN_MOBY_DICK.txt, and SHAKESPEARE.txt*. The files contain 215724, 411191, and 965465 words respectively, where each word in the files is separated by a newline ("\n") character. All of the words are derived from famous English literature books. Since my algorithm is checking for the unique words in the text file, the query list uses the same file as the word list.

All the files are sourced from the text files supplied by the FIT3143 teaching team in Moodle. The file *LITTLE_WOMEN_MOBY_DICK.txt* is made by combining *LITTLE_WOMEN.txt* with *MOBY_DICK.txt* to obtain a file size larger than the first file, but smaller than the last file for computational time comparison purposes.

d) Measure and Analyse the Performance of the Serial Algorithm/Code

The serial code is executed using the three text files as both the word list and query list.

These are the specifications of the hardware of the single computer used to run the code:

CPU : 8-core Intel® Core™ i7-10870H CPU @ 2.20GHz
RAM : 16GB
Disk : 512GB ESR512GTLCG-EAC-4 SSD
Status: Charging (Laptop)

The results obtained by running the code on a single computer and CAAS are:

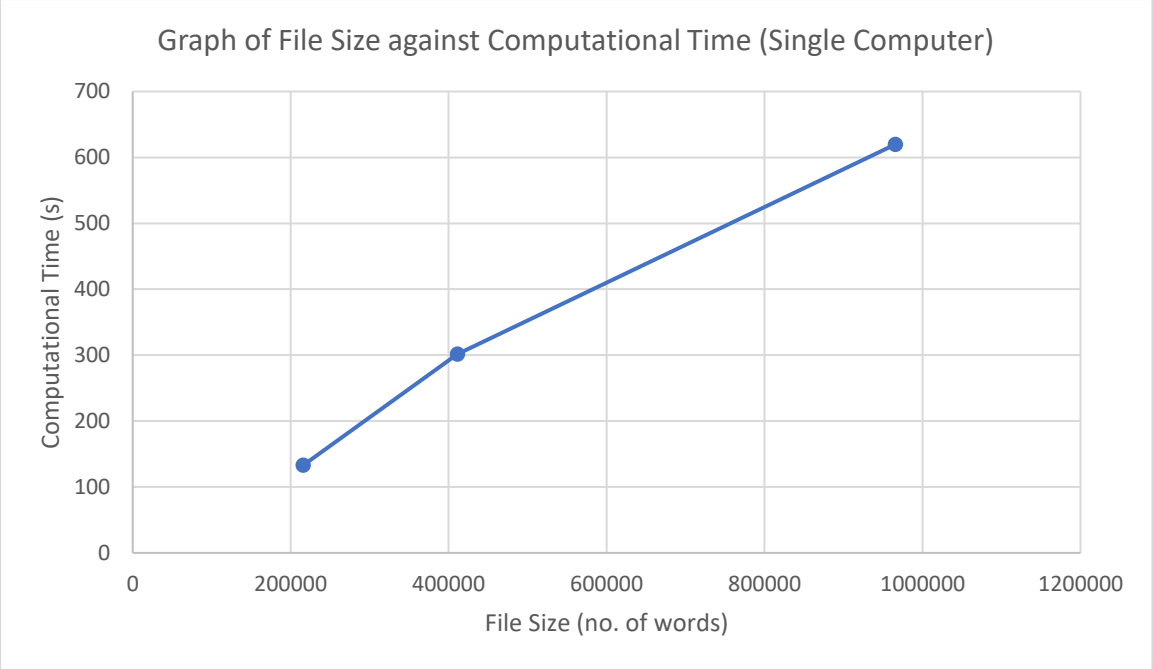
File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	132.859739	132.961368
LITTLE_WOMEN_MOBY_DICK.txt	411191	301.407223	301.525853
SHAKESPEARE.txt	965465	619.713980	619.963919

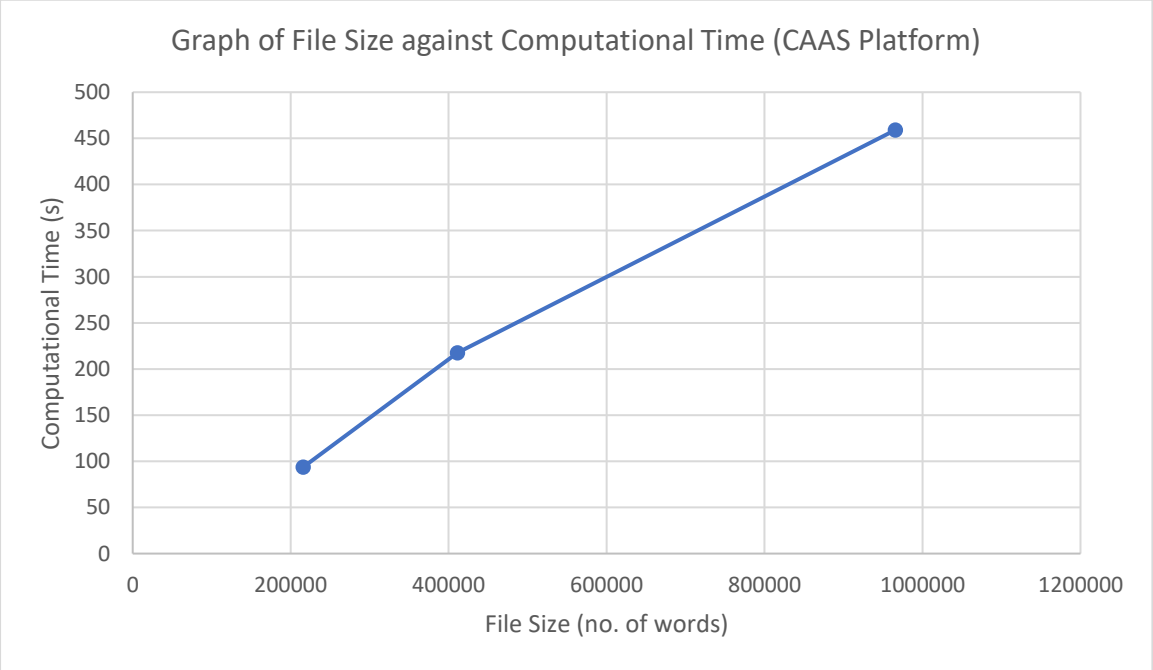
These are the specifications of the hardware of the cluster computing platform used to run the code:

CPU : 256+ cores AMD Epyc server CPUs
RAM : 256+ GB
Storage : 5+ TB
Network : Gigabit Ethernet
Scheduler : SLURM

The results obtained by running the code on the three text files is recorded:

File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	93.634984	93.694676
LITTLE_WOMEN_MOBY_DICK.txt	411191	217.697280	217.823710
SHAKESPEARE.txt	965465	458.963608	459.185315





From the results, we can observe that the code runs approximately O(√n) to O(log(n)) time. Since the Z-algorithm is an exact-string pattern matching algorithm, the accuracy of the results is 100% with no false positives or false negatives.

The difference in computational time and overall time is very little (< 0.5s), therefore we can deduce that the main part of the code that requires large computational times is the section that is checking for unique occurrences and the string pattern matching algorithm. The word comparisons against the unique word list takes the most time, because the size of the word list increases as the program runs, meaning there are more checks required per word to determine its uniqueness.

B) Carry Out a Dependency Analysis to Ascertain that the String-Matching Algorithm is Parallelizable

Bernstein's conditions outline the conditions which are sufficient to determine whether the execution of two or more threads can be performed simultaneously (FIT3143, 2023). Therefore, a dependency analysis using Bernstein's conditions is performed to determine which portions of the above implemented string-matching algorithm can be parallelized.

The three conditions which constitute the Bernstein's conditions are as follows:

- $I_1 \cap O_2 = \emptyset$ (Anti dependency)
- $I_2 \cap O_1 = \emptyset$ (Flow dependency)
- $O_1 \cap O_2 = \emptyset$ (Output dependency)

with I_1 and O_1 representing the input and output for the first thread T_1, whereas I_2 and O_2 represent the input and output for second thread T_2.

Main Program

When i = 0,
I_1 = {inputFileNames[0], inputFileSizes[0]}
O_1 = {wordLists[0], uniqueWordLists[0], uniqueCounts[0]}

When i = 1,
I_2 = {inputFileNames[1], inputFileSizes[1]}
O_2 = {wordLists[1], uniqueWordLists[1], uniqueCounts[1]}

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

There is no violation of the Bernstein's conditions, therefore the main program can be parallelized.

countUnique Function

When i = 0,
I_1 = {wordlist[0], uniqueWordList[0...0]}
O_1 = {uniqueWordList[uniqueCount]}

When i = 1,
I_2 = {wordlist[1], uniqueWordList[0...uniqueCount]}
O_2 = {uniqueWordList[uniqueCount]}

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

There is an Anti Dependency, Flow Dependency, and Output Dependency. All three Bernstein's conditions are violated, therefore this function is not parallelizable.

isUnique function

When i = 0,
I_1 = {uniqueWordList[0]}
O_1 = ∅

When i = 1,
I_2 = {uniqueWordList[1]}
O_2 = ∅

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

There is no violation of the Bernstein's conditions, therefore the function can be parallelized.

stringPatternMatch Function

Section - Populating the Z-array:

When i = 0,
I_1 = ∅
O_1 = {z[1]}

When i = 1,
I_2 = ∅
O_2 = {z[2]}

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

There is no violation of the Bernstein's conditions, therefore the section can be parallelized.

Section - Finding pattern matches:

When i = 0,
I_1 = {z[1]}
O_1 = ∅

When i = 1,
I_2 = {z[2]}
O_2 = ∅

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

There is no violation of the Bernstein's conditions, therefore the section can be parallelized. Since both sections are parallelizable, therefore the function is parallelizable.

z Array Function

When i = 1,
I_1 = {str[0...]}
O_1 = {z[1]}

When i = 2,
I_2 = {z[1], str[0...]}
O_2 = {z[2]}

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

Since there is a Flow Dependency between I_2 and O_1 , if two threads were to read and write simultaneously from z[1], this would lead to a race condition and hence the outcome will not be correct.

C) Calculating the Theoretical Speedup of a Parallel Implementation of the String-Matching Algorithm.

Using the results from subsection d of Part A, the theoretical speedup of the algorithm can be calculated using Amdahl's law where the number of processors is decided to be 4. The formula is:

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

For a file size of 215724 words, the serial and parallel portions are:

$r_s = \frac{(132.961368 - 132.859739)}{132.961368} \approx 0.0008$, $r_p = 1 - r_s = 0.9992$

$S_p = \frac{1}{r_s + \frac{r_p}{p}} = \frac{1}{0.0008 + \frac{0.9992}{4}} \approx 3.990$

For a file size of 411191 words, the serial and parallel portions are:

$r_s = \frac{(301.525853 - 301.407223)}{301.525853} \approx 0.0004$, $r_p = 1 - r_s = 0.9996$

$S_p = \frac{1}{r_s + \frac{r_p}{p}} = \frac{1}{0.0004 + \frac{0.9996}{4}} \approx 3.995$

For a file size of 965465 words, the serial and parallel portions are:

$r_s = \frac{(619.963979 - 619.713980)}{619.963979} \approx 0.0004$, $r_p = 1 - r_s = 0.9996$

$S_p = \frac{1}{r_s + \frac{r_p}{p}} = \frac{1}{0.0004 + \frac{0.9996}{4}} \approx 3.995$

From the calculations using Amdahl's Law, we can estimate that there is a theoretical speedup of close to 4 using 4 processors for varying file sizes, which is a sublinear speedup.

D) Design and Develop a Parallel String-Matching Algorithm Based on Data Parallelism on a Shared Memory Parallel Computing Architecture

a) Design a Data Parallelism Algorithm for the Selected String-Matching Algorithm

Referencing the workshop, I tried to parallelise the main function for string pattern matching using a shared data parallelism approach with OpenMP. Here, each computation of determining whether a word is unique or not is conducted in parallel (i.e. the words/data are split among the threads). If a word is deemed unique, it will be written to the uniqueWordList, and the counter for the number of unique words is incremented. Since the unique count variable is not shared among the threads, I added the reduction clause to combine the results together after all of the threads have finished computation.

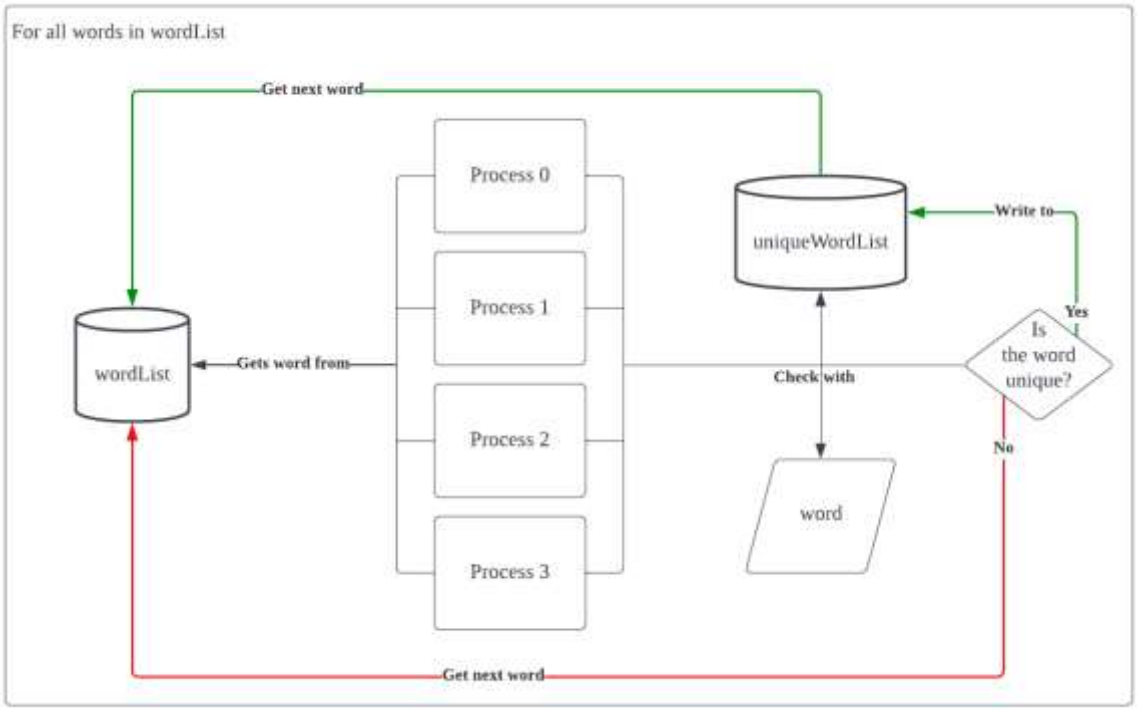
```
#pragma omp parallel for shared(wordList, uniqueWordList) reduction(+:uniqueCount)
for(int i=0; i<inputSize; i++)
{

    if(isUnique(wordList[i], uniqueWordList, uniqueCount))
    {
        uniqueWordList[uniqueCount] = strdup(wordList[i]);
        uniqueCount++;
    }
}
```

This speeds up the overall algorithm because each word is computed individually without interfering with one another, therefore we can divide the computation to multiple processors. Even if they share the same data (the word list), since each word is only checked once, there will be no race condition when reading from it, thus achieving algorithm parallelism.

It is important to mention that this implementation sacrifices a portion of the algorithm's accuracy to achieve much higher speedup. The main focus is to parallelise the process of string pattern matching. Although parallelisation is implemented at the expense of recognising more words as unique (i.e. including multiple instances of the same word), none of the unique appearances are left out. Due to the read/write speed of the Solid-State Drive (SSD), the impact on overall computational time of writing extra words to the text file is negligible.

Flowchart of the parallelisation implementation of the algorithm:

A flowchart for the algorithm implementation over 4 cores/threads is provided for better visualisation.



All processes get a word from the list, computes whether it is present in the unique word list. The word is stored inside the unique word list if it is not existent within the list (i.e. unique). Once a process is finished with the computation of the current word, it will continue to obtain the next word to check from the word list, until the word list is exhausted.

E) Analyse and Evaluate the Performance of the Parallel Algorithm

a) Measure the Performance of the Parallel Algorithm/Code for Different Word Counts and Queries

These are the results running the program in parallel on my local computer, using the same specifications of hardware:

No. of cores (threads) utilised: 4

File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	21.862657	21.946717
LITTLE_WOMEN_MOBY_DICK.txt	411191	56.999011	57.163515
SHAKESPEARE.txt	965465	121.783202	122.110807

No. of cores (threads) utilised: 8

File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	9.700629	9.782299
LITTLE_WOMEN_MOBY_DICK.txt	411191	23.933253	24.063490
SHAKESPEARE.txt	965465	58.737793	59.022134

No. of cores (threads) utilised: 16

File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	5.658463	5.741899
LITTLE_WOMEN_MOBY_DICK.txt	411191	15.560171	15.751695
SHAKESPEARE.txt	965465	39.551791	39.903544

These are the results running the program in parallel on the CAAS platform:

No. of cores (threads) utilised: 4

File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	13.897303	13.937123
LITTLE_WOMEN_MOBY_DICK.txt	411191	37.414639	37.483428
SHAKESPEARE.txt	965465	80.111631	80.249263

No. of cores (threads) utilised: 8

File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	5.516326	5.600352
LITTLE_WOMEN_MOBY_DICK.txt	411191	16.449474	16.552779
SHAKESPEARE.txt	965465	33.809447	34.019818

No. of cores (threads) utilised: 16

File Name	File Size (No. of Words)	Computational Time (s)	Overall Time Including Read & Write (s)
MOBY_DICK.txt	215724	7.237740	7.278568
LITTLE_WOMEN_MOBY_DICK.txt	411191	17.606285	17.130293
SHAKESPEARE.txt	965465	46.111375	46.252796

b) Calculate the Actual Speed Up

Serial and Parallel Times on the 215724-word file:

Number of Cores	Computational Time on Single Computer (s)	Computational Time on CAAS (s)
1 (Serial)	132.859739	93.634984
4	21.862657	13.897303
8	9.700629	5.516326
16	5.658463	7.237740

The formula for calculating the Speed Up over varying number of cores is:

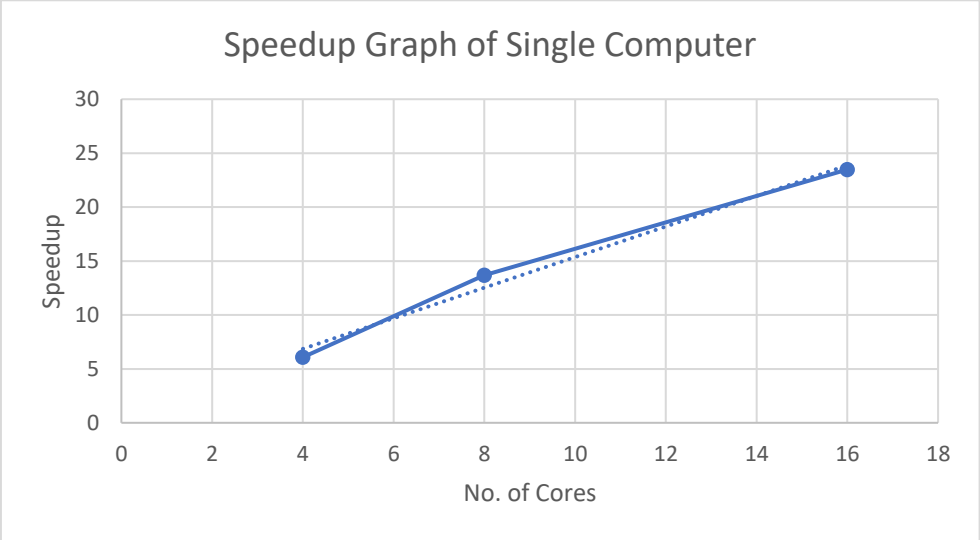
$$S = \frac{T_s}{T_p}$$

For a single computer,

$S_4 = \frac{132.859739}{21.862657} \approx 6.077$

$S_8 = \frac{132.859739}{9.700629} \approx 13.696$

$S_{16} = \frac{132.859739}{5.658463} \approx 23.480$

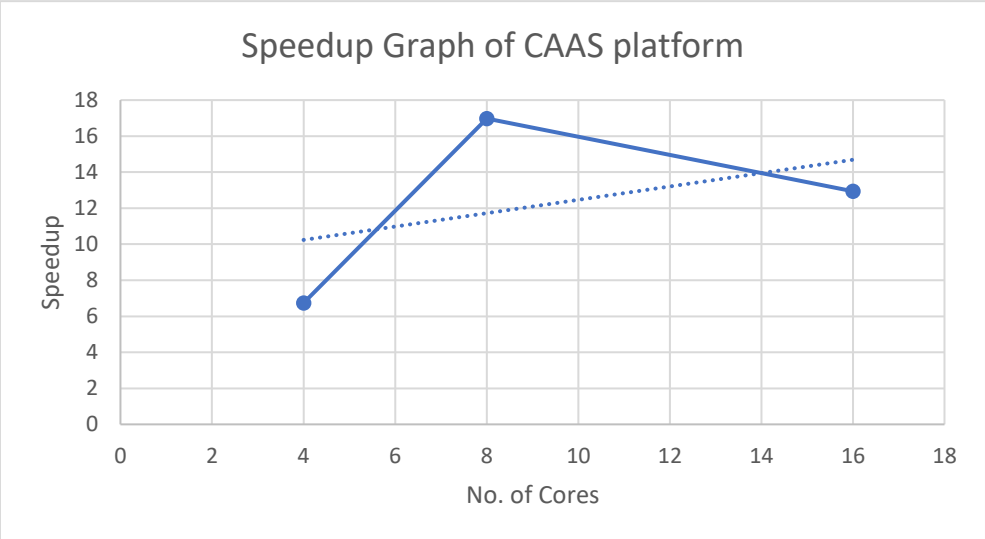


For a cluster computing platform,

$S_4 = \frac{93.634984}{13.897303} \approx 6.738$

$S_8 = \frac{93.634984}{5.516326} \approx 16.974$

$S_{16} = \frac{93.634984}{7.237740} \approx 12.937$



c) Analysis and Discussion of the Performance

There is a similar trend in the computational times of different file sizes running on both a single computer and on CAAS: The larger the file size, the smaller the growth rate of computational time. This is because not all of the words in the text file are unique, and the larger the file size, the more words that are repeated, therefore the growth rate of the unique word list decreases, which explains the trend in the graph.

Running the serial program on a single computer (my local computer) is slower by roughly 40% compared to running on a cluster computing setup (CAAS); whereas running the parallel program on a single computer is slower by roughly 150%. The speed difference in computing the programs may be due to the single computer having a weaker CPU, or it running tasks in the background. The load balancing management in a cluster computing setup may also be more efficient than a single computer, which evens out the workload between nodes/threads, reducing the computational time.

From the calculations of the theoretical speedup and the results of the actual speedup, super linear speedups are observed on both a single computer and on the CAAS platform. This is due to the difference in implementation of the algorithm, whereby accuracy is sacrificed in exchange for speed by introducing false positives into the unique word list.

File name	Unique count (Serial)	Unique count (Parallel)	% Change
MOBY_DICK.txt	19850	34860	75.62%
LITTLE_WOMEN_MOBY_DICK.txt	25380	46157	81.86%
SHAKESPEARE.txt	33031	63631	92.64%

We can observe that by adding less than double the unique word counts, we achieve a super linear speedup in the algorithm.

However, it is noticed that the program is slowed down when running on 16 cores on CAAS. This might be due to the number of cores allocated is less than the number of threads specified, therefore more tasks are possibly delegated to certain threads, increasing its computational time.

Optimization activities:

A few approaches are conducted using the analysis earlier in the report, all utilising the OpenMP library.

1. Parallelising the stringPatternMatch function (unsuccessful)

Parallelising the first section of the function (i.e. populating the Z-array), I found out that the time taken is no better than running the code in serial, if not worse performing. This may be due to the function being called by each thread, creating additional overheads to spawn new threads and affecting the overall performance from nested parallelism.

```
for(i = 0; i<strlen(str); i++)
{
    z[i] = 0;
}
```

Parallelising the second section of the function (i.e. checking the Z-array to find for matches) directly will result in an error message saying "invalid branch to/from OpenMP structured block". This is due to OpenMP's restrictions on how return statements can be used within parallel regions.

```
for (i = 0; i < strlen(str); i++)
{
    if (z[i] == strlen(pat)){
        return true;
    }
}
```

Therefore, the function needs a slight tweak whereby we add in a Boolean flag to represent the detection of a match, then the function returns the flag instead. This results in additional computation because all the threads need to finish computing before the function can be returned to its caller. This does not speed up the algorithm but instead slows it down because we are unable to directly return the function on the event that a match is found.

2. Parallelising the isUnique function (unsuccessful)

The same situation as the second section of the stringPatternMatch function occurs.

```
for(int i=0; i<uniqueCount; i++)
{
    if(stringPatternMatch(word, uniqueWordList[i])){
        return false;
    }
}
```

Again, we are unable to exit the function early due to the restrictions on return statements, resulting in increased computational time for the overall algorithm.

References

Trivedi, U., (2022) “Z algorithm (Linear time pattern searching Algorithm)” GeeksforGeeks Retrieved from: Z algorithm (Linear time pattern searching Algorithm) - GeeksforGeeks

FIT3143, (2023) “Workshop Week 3 Sample Solution” Monash University Malaysia Retrieved from: Workshop Week 3 Sample Solution.docx (d3cgwrxphz0fqu.cloudfront.net)