FIT3143 - Assignment 1
Parallel String Searching Algorithm on Shared Memory

Unit learning outcomes (LO) for this assignment:

a) Design and develop parallel algorithms for various parallel computing architectures (LO3).
b) Analyse and evaluate the performance of parallel algorithms (LO4).
c) Apply technical writing and presentation to effectively communicate parallel computing to a range of academic and expert audiences (LO5).

I. Preamble

A multicore architecture of a central processing unit (CPU) enables task and data-level parallelism on shared memory architectures. Such parallelism improves performance of an algorithm used in a range of applications such as string searching or string matching. String matching or string searching refers to the process of finding one or more occurrences of a particular pattern or substring within a larger text or string. In other words, it involves searching for a specific sequence of characters (pattern) in a given text. This technique is commonly used in computer science, text processing, and various programming tasks.

String matching and searching have numerous real-world applications, such as finding keywords in documents, text indexing and searching in databases, text editors' "find" functionality, and more complex tasks like bioinformatics and DNA sequence analysis.

The string searching algorithm's primary objective is to determine whether the pattern exists in the given text and, if so, identify the starting position (or positions) of the pattern within the text.

Some of the popular string searching or string matching algorithms are listed below::

a) Naive string matching: This is the simplest and most straightforward method. It involves sliding the pattern over the text character by character and checking for a match at each position.
b) Knuth-Morris-Pratt (KMP) algorithm: KMP is a more efficient algorithm that avoids unnecessary comparisons by utilizing a prefix table (also known as the "failure function") to find partial matches.
c) Boyer-Moore algorithm: This is another efficient algorithm that compares the pattern from right to left, which allows for multiple characters to be skipped in the text when a mismatch occurs.
d) Bloom filter algorithm: Probabilistic data structure used for testing whether an element is a member of a set.

The choice of the algorithm depends on the specific requirements of an application, the length of the text and pattern, and the potential presence of repetitive patterns.

II. Bloom filter algorithm

As aforementioned, Bloom filter is a probabilistic data structure used for testing whether an element is a member of a set. It efficiently checks for set membership by using a compact array of bits. Bloom filters are particularly useful when memory space is a concern and approximate answers are acceptable. They are commonly used in scenarios where false positives (indicating that an element is in the set when it is not) are acceptable but false negatives (indicating that an element is not in the set when it is) are not.

The basic idea behind a Bloom filter involves the following components:

a) Bit Array: A Bloom filter uses a fixed-size bit array, typically initialized with all bits set to 0.

b) Hash Functions: Multiple independent hash functions are employed to map the input element to different positions in the bit array. This is important because a larger number of hash functions used would provide more unique bit positions within an array.

The typical operations supported by a Bloom filter are:

a) Insertion: When an element is added to the set, it undergoes hashing with multiple hash functions, and the corresponding bit positions in the bit array are set to 1. Please refer to Figure 1 for a sample illustration of this process.

b) Membership Query or Lookup: To check if an element is in the set, the element is hashed with the same hash functions used during insertion. If all the corresponding bit positions in the bit array are set to 1, the element is considered "probably in the set." If any of the bits are 0, then the element is definitely not in the set. Please refer to Figure 2 for a sample illustration of this process.

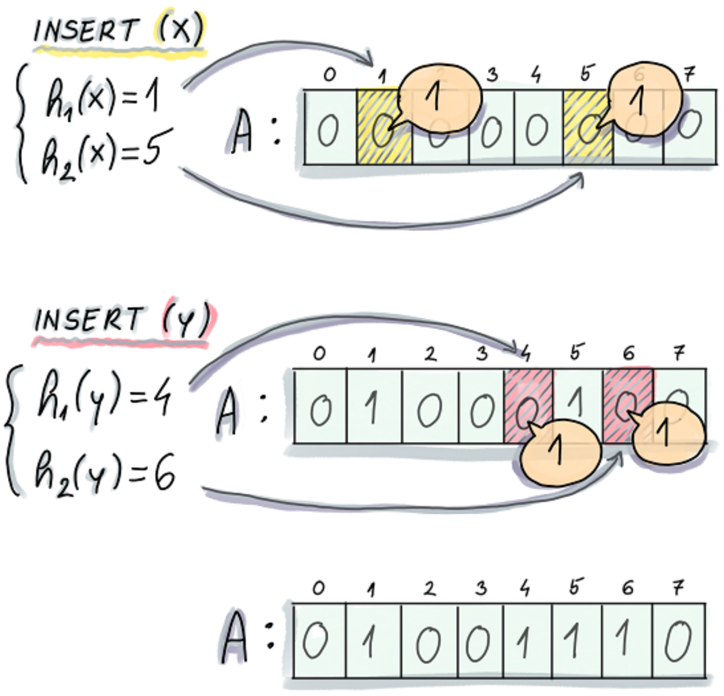

Figure 1: “Example of insert into Bloom filter. In this example, an initially empty Bloom filter has m=8 (array size), and k=2 (two hash functions). To insert an element x, we first compute the two hashes on x, the first one of which generates 1 and the second one generates 5. We proceed to set A[1] and A[5] to 1. To insert y, we

also compute the hashes and similarly, set positions A[4] and A[6] to 1”. Both the figure and caption are from: https://freecontent.manning.com/all-about-bloom-filters/

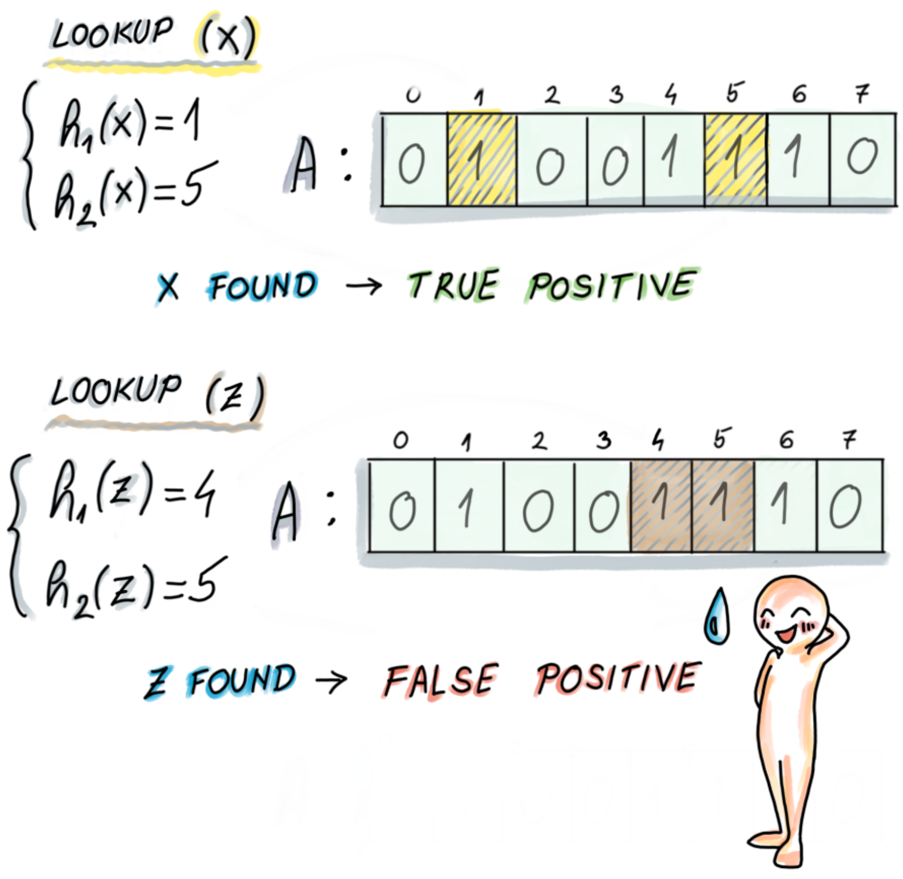

Figure 2: “Example of a lookup on a Bloom filter. We take the resulting Bloom filter from Figure 1, where we inserted elements x and y. To do a lookup on x, we compute the hashes (which are the same as in the case of an insert), and we return Found/Present, as both bits in corresponding locations equal 1. Then we do a lookup of an element z, which we never inserted, and its hashes are respectively 4 and 5, and the bits at locations A[4] and A[5] equal 1, thus we again return Found/Present. This is an example of a false positive, where two other items together set the bits of the third item to 1. An example of a negative (negative is always true), would be if we did a lookup on an element w, whose hashes are 2 and 5 (0 and 1), or 0 and 3 (0 and 0). If the Bloom filter reports an element as Not Found/Not Present, then we can be sure that this element was never inserted into a Bloom filter” Both the figure and caption are from: https://freecontent.manning.com/all-about-bloom-filters/ .

It's important to note that a Bloom filter may produce false positives but not false negatives. False positives occur because multiple elements might share some of the same bit positions, and if those bits are set to 1 by different elements, a collision happens, leading to a false positive result.

The probability of a false positive can be minimized by adjusting the size of the bit array and the number of hash functions used, depending on the expected number of elements in the set and the desired false positive rate.

Bloom filters are widely used in various applications, such as caching, network routers, spell checking, duplicate elimination, and distributed systems for quick filtering before accessing larger data structures like databases or disk storage, thus saving computational resources.

III. Problem statement & Objectives of this assignment

Serial implementation of any string matching algorithms such as Bloom filters on large datasets (i.e., word list and queries) may require costly computational or query time.

The aim of this assignment is to parallelize a string matching algorithm using parallel computing on shared memory with POSIX/OpenMP in C or C++ programming language.

Detailed specifications are detailed in subsequent sections.

Objectives of the assignment:

a) Investigate the computational bottleneck of a serial string matching algorithm such as the Bloom Filter (or an equivalent) algorithm for a sufficiently large dataset or word counts and queries.
b) Carry out a dependency analysis to ascertain that the string matching algorithm is parallelizable.
c) Calculate the theoretical speedup of a parallel implementation of the string matching algorithm.
d) Design and develop a parallel string matching algorithm based on data parallelism on a shared memory parallel computing architecture.
e) Analyse and evaluate the performance of the parallel algorithm.
f) Apply technical writing to effectively communicate parallel computing.

d)

IV. What needs to be done

A) Investigate the computational bottleneck of a serial string matching algorithm such as the Bloom Filter algorithm for large datasets

a) Select a complex serial based string matching algorithm such as Bloom Filters or equivalent (refer to the example in page 1). Simple or naive based string matching algorithms such as sequential or binary search are not allowed. If you have any questions on the choice of algorithm, please consult teaching staff — do not wait till the last day before the deadline.
b) Implement a serial version of the string matching algorithm in C programming language (you may use C++, but it has to be either C or C++)
c) Execute the code for a sufficiently large string search dataset of word list and query list. You can refer to available string matching datasets online.. Examples include obtaining data from DeezyMatch or Toponym (please make sure you cite the dataset in your report).
d) Measure the performance of the code. For instance, if you are using Bloom Filters, you should measure the time required during insertion and lookup using a large number of words. Analyse which part of the algorithm/code that consumes most of the computational time. *Note: Make sure to test with a large dataset(s) to observe the computational time.*

B) Carry out a dependency analysis to ascertain that the string matching algorithm is parallelizable

a) Use Bernstein's condition to carry out a dependency analysis.
b) The dependency analysis is to ensure that the serial algorithm can be parallelized using data parallelism.

C) Calculate the theoretical speedup of a parallel implementation of the string matching algorithm

a) Measure the theoretical speed up based on a fixed dataset size.

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.
b) Implement the parallel algorithm in C / C++ with either POSIX threads or OpenMP

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. You should use the same dataset in the serial code test for the parallel code test. Use your local computer and the CAAS platform to analyze the performance of your parallel and serial code.
b) Calculate the actual speed up.
c) Optimize your algorithm to improve its performance.

F) Report

a) Compile the outcomes of (A) to (E) above into a written report. The report should include the following sections:
b) Introduction - A brief description of the string matching algorithm and the justification for the choice of selecting the algorithm for this assignment.
c) Dependency analysis and theoretical speed up calculations
d) Description of the parallel algorithm design, which includes algorithm pseudocode, proper flowcharts or equivalent.
e) Results tabulation – Tabulation or illustration of the results using tables or graphs.
f) Analysis & discussion - Provide an analysis on the behaviour of the parallel algorithm. Include any additional observations to support the analysis.
g) Maximum number of words in the report: 1,500 words. The word count excludes figures, tables, and references (if any).

G) Code demonstration and interview

a) Demonstrate the program during the lab session (after the assignment submission due date)
b) Interview session with the lab tutor to ascertain the authenticity of the work completed by the student.

V. Submission:

Individual Assignment: Yes. As part of the authentic assessment structure for technical and taught based units, each student is to work individually in this assignment. While you may have discussions with your peers about the assignment, you must produce and submit your own work. Submission of the assignment is made via Moodle.

The following files are required for submission to Moodle:

1) **A compressed file (i.e., zipped) containing the following:**

a) Code file(s) (.h and .c/.cpp file extensions) with adequate comments.
b) Word and query list used to test the algorithm.
c) Any other files (e.g., Makefile) which may support your assignment work.

2) **The Assignment report (in PDF). Please do not compress this file. Drag and drop the PDF report into the submission area.**
Similarity check is enabled for the report submission. Once you upload a report as a draft, Turnitin will check for similarity. Based on the similarity report, you can amend your report to reduce its similarity index and reupload your report. Therefore, it is important that you do not compress your report into a zipped file. Once you submit your report for grading, you are not able to make any amendments to it. Please contact your tutor if you would like your submission to be reverted to draft mode.

The Assignment-1 submission is due on Monday, Week 7 of the semester (09:30 AM Clayton/Malaysia time). All demo and interviews will be scheduled in Week 7 (unless special consideration is granted or an alternative arrangement is made).

VI. Marking Guide:

This assignment is worth 25 marks (or 25 percentage points) of your overall unit score.

PROVISIONAL MARKING GUIDE

Part A: Assignment Code, Demo and Report Assessment (25 marks)

Investigate the computational bottleneck of a serial string matching algorithm such as the Bloom Filter algorithm for large datasets - **Based on the submitted report and serial string matching code**	5 marks
Carry out a dependency analysis to ascertain that the string matching algorithm is parallelizable - **Based on the submitted report**	1 mark
Calculate the theoretical speedup of a parallel implementation of the string matching algorithm - **Based on the submitted report**	1 mark
Design and develop a parallel string matching algorithm based on data parallelism on a shared memory parallel computing architecture - **Based on the submitted report and parallel code**	12 marks
Analyse and evaluate the performance of the parallel algorithm - **Based on the submitted report and parallel code**	6 marks

Part B: Assignment Interview (Negative marking based on obtained marks from Part A)

The student has clearly prepared and understands the code. The student can answer questions correctly and concisely with little to no prompting.	Deduct 0% of Part A
The student is reasonably well prepared and can consistently provide answers that are mostly correct.The student may lack confidence or speed in answering.	Deduct up to 10% of Part A
The student may have prepared the code and can give answers that are partially correct but he/she clearly can't engage in a serious discussion of the parallel algorithm/code	Deduct up to 25% of Part A
The student may have partially prepared the code before and can give some very basic answers. However, the student clearly can't engage in a serious discussion of the code and demonstrates a poor understanding of the parallel algorithm/code.	Deduct up to 50% of Part A
The student has not prepared and cannot answer most/all basic questions, or the student shows any indication of not even seeing/recognizing the code before.	Deduct up to 100% of Part A

Note: Additional penalty (i.e., marks deduction) will be applied for poor grammar, high similarity index for the report and late submission.

VII. Academic Integrity & Generative AI:

Please do not attempt to plagiarise or collude your assignment work.

Please refer to the University's policy on academic integrity.

Generative AI tools cannot be used for any assessments in this unit.

In this unit, you must not use generative artificial intelligence (AI) to generate any materials or content in relation to your assessment.